Writing clean and safe UDFs in Delphi

Gregory H. Deatz

Introduction

This article makes some assumptions about the reader's knowledge of Delphi. The reader should understand Delphi well enough to know how to create a DLL--We will demonstrate how to create UDFs, not DLLs. That being said, much of the article depends heavily on the study of FreeUDFLib, a free UDF library (hence the name) distributed with this article, so all source code for a fully functional UDF library is provided!

We will begin by demonstrating a very simple UDF (`Modulo`), and from there we will discuss returning values by reference. Returning values by reference (strings and dates must be returned by reference) leads into a discussion of issues surrounding dynamic allocation of memory and appropriate cleanup.

This presentation will provide the developer with all necessary tools for making good decisions about the design and implementation of solid UDF libraries for InterBase.

Writing a UDF

What is a UDF?

Quite simply put, a UDF in InterBase is a function in a DLL! This simple use of shared libraries provides the developer with virtually unlimited amounts of power and flexibility. Virtually any function that can be exposed through a DLL can be used by InterBase. This comment, however, should be taken with a grain of salt--The intent of a UDF is to perform some small operation that is not available in SQL or InterBase's stored procedure language.

An example of a UDF is

Function: Integer **Modulo** *(Integer Numerator, Integer Denominator)*

> Divide *Numerator* by *Denominator* and return the remainder. This function is essential in many routines, but it is not available in InterBase's DSQL language.

Writing the first UDF

Open the `` `FreeUDFLib/FreeUDFLib.dpr' `` project distributed with this article, and take a look at the `` `FreeUDFLib/MathFncs.pas' `` unit. Among a few other declarations, the reader will find a declaration reading

```
function Modulo(var Numerator, Denominator:
  Integer): Integer; cdecl; export;
```

The first thing to note is that *all* arguments in an InterBase UDF are passed by reference. The second thing to note is that this function uses the `cdecl` calling convention. A discussion of calling conventions is beyond the scope of this article, but the reader can see some documention by looking up `cdecl` in Delphi's on-line help. Beyond that, the reader should take it on a certain degree of faith that this is how InterBase wants its UDFs written. The keyword `export` tells Delphi that this function will be exported.

Further examination of the actual project source (click "View, Project Source") will show code similar to this:

```
exports
  Modulo;
```

The exports clause in the project source tells Delphi that this function *is* exported.

Now, just for fun, we will list the code for `Modulo`:

```
function Modulo(var Numerator, Denominator: Integer): Integer;
begin
  {$ifdef FULDebug}
  WriteDebug('Modulo() - Enter');
  {$endif}
  if (Denominator = 0) then
    result := 0
  else
    result := Numerator mod Denominator;
  {$ifdef FULDebug}
  WriteDebug('Modulo() - Exit');
  {$endif}
end;
```

Inspection of the code will reveal that `Modulo` returns `0` when *Denominator* is `0` (this is better than crashing), and in all other situations it simply returns *Numerator* `mod` *Denominator*.

Using the UDF in InterBase

A precompiled version of FreeUDFLib is distributed with the article (``FreeUDFLib/FreeUDFLib.dll'`). Place this DLL in InterBase's "bin" directory, which is usually

```
c:\Program Files\InterBase Corp\InterBase\Bin.
```

Now, run the ``WISQL32.exe'` utility. It is located in InterBase's "bin" directory. Create a test database ("File", "Create database"), and once attached to the new test database, type the following DDL command:

```
declare external function f_Modulo
  integer, integer
  returns
  integer by value
  entry_point "Modulo" module_name "FreeUDFLib.dll"
```

For a full description of the syntax for declaring UDFs in InterBase, see the `Programmer's`

Guide.

Now, run the above statement and commit. To run this UDF from InterBase, type the following statement:

```
select f_Modulo(4, 3) from rdb$database
```

`RDB$DATABASE` is a one-row table in all InterBase databases, so after running the above command, the following text should be in the results window:

```
select f_modulo(4, 3)
  from rdb$database

   F_MODULO
===========

          1
```

Writing UDFs is easy!

Returning strings

We've successfully written a Delphi DLL that can be used as a UDF library in InterBase. Wouldn't a great function be

Function: PChar **Left** *(PChar sz, Integer Len)*

Return the *Len* leftmost characters of *sz.*

The question is, how do we return a string? Already implied by the above declaration, InterBase doesn't respect Delphi's `String` datatype, so we are forced to use `PChar`.

`PChar`'s versus `String`'s is not a big deal, except that Delphi does not automatically clean up `PChar`'s, and the memory to store a string must be, at some point, explicitly allocated by the developer. Our first UDF returned a scalar value on the function's calling stack, which meant that we did not have to worry about cleanup issues. With any form of string, cleanup issues *must* be addressed.

Here are a few possible solutions to the "returning strings" conundrum. We will explain why certain solutions are desired, and why others will simply not work.

Solution #1: Global static memory

There is an obvious way to accomplish returning strings: Maintain a global `PChar` that has some amount of space allocated to it. Stuff this string with the desired return value, and return the global `PChar`. The only problem with this is that InterBase is multi-threaded, and if a Delphi DLL does this, InterBase will surely crash in a multi-user environment.

It seems that solution #1 is no solution at all...

Solution #2: Thread-local static memory

Instead of maintaining a single global variable, maintain a *thread-local* variable.

Whenever a UDF wishes to return a string, it simply copies the string into the string referenced by the thread-local `PChar`, and returns it.

This solution certainly sounds elegant, and it certainly lends itself to being clean, but... How do we manage thread-local variables in Delphi? Also, this solution certainly sounds like it will work, but will it? InterBase is a multi-threaded environment, surely, but how does it handle the scheduling of its calls to UDFs? (see section A discussion of Solution #2 { anchor link to discussion of Soltuion #2, this document })

Solution #3: Returning dynamically allocated strings

Another obvious solution: Every time a function returning a string is ready to return the string, allocate a bit of memory for the string and return a `PChar` to it. InterBase won't crash--at least not right away. It should be clear that this presents a nasty memory leak. The Delphi function will keep allocating memory, but nobody is cleaning it up.

In InterBase 5.0, a new keyword call `free_it` was introduced. This keyword is used like this:

```
declare external function f_Left
  cstring(254), integer
  returns
  cstring(254) free_it
  entry_point "Left" module_name "FreeUDFLib.dll";
```

If the developer chooses to return strings in the "memory-leaky" fashion, then the UDF should be declared with the `free_it` keyword, just like above. This allows the developer of the UDF to be sloppy, and it forces InterBase to do the housekeeping.

This is a reasonable solution if the developer of the UDF wants to be sloppy; however, it is the author's contention that all functions, *especially third-party functions* should do their own housekeeping, or they should do nothing to "dirty the house" to begin with. It is considered bad form to write a function that is known to be leaky, only to "pass the buck" to the calling application.

Another problem with the `free_it` solution is that it only works with InterBase 5 and up. If the developer intends to write functions for use with InterBase 4.x or lower, this solution simply won't work. (see section A discussion of Solution #3 { anchor link to discussion of Soltuion #3, this document })

Solution #4: Making InterBase do the work

An under documented feature of InterBase allows a UDF declaration to specify a particular parameter as the assumed return value of the function. By implementing a UDF in this way, the UDF developer forces InterBase to pass it valid spaceholders for strings. In other words, the UDF developer won't have to worry about dynamic memory issues because this problem is dealt with entirely in the InterBase engine.

As was indicated before, third-party routines should either do their own housekeeping, or they should do nothing to "dirty the house" to begin with. This method is a simple and elegant way to avoid messing up InterBase's house. (see section A discussion of Solution #4 {anchor link to discussion of Soltuion #4, this document}).

Notes on Delphi's memory manager

Delphi is a derivative product of the old days, when Borland Pascal was called Borland Pascal, and Borland was Borland, and boys were boys, and men were men, and multi-threading was just plain unavailable to the DOS world. When Delphi moved into the Win32 world with version 2.0, Inprise discovered that the memory manager wasn't thread-safe.

To solve the thread-safety concerns, they wrapped their memory management routines in critical sections, thus making the memory manager thread-safe. Critical sections are beyond the scope of this article. Suffice it to say that they ensure orderly access to a shared resource.

The odd trick Inprise played, though, is that the critical sections are used only if a not-so-well-known system variable, *IsMultiThread* is set to `True`. (*IsMultiThread* is defined in the unit ``` `System.pas' ```, which is implicitly used by *all* Delphi units.)

The basic gyst of this story is as follows: Delphi *is* thread-safe, but only when the developer tells it to be. Whenever an application or library knows that it may be dealing with multiple threads it should guarantee that *IsMultiThread* is set to `True`; otherwise, the application or library is *not thread safe*. (Important note: *IsMultiThread* is set implicitly if the developer uses a `TThread` object.)

It cannot be stressed enough that *IsMultiThread must* be set to `True` in multi-threading environments.

A discussion of Solution #2

In our introduction to this solution, we asked the question, "Will thread-locals work?" The answer is a resounding yes! InterBase is a multi-threaded architecture, and any number of different queries can be running in a given thread. InterBase is guaranteed to execute a UDF and process its results within a single *atomic* action, thus thread-locals are perfectly safe for returning strings. (For a more in-depth conversation, visit IB's web site, or talk with the author after the lecture).

Thread-local variable's are extremely easy to work with, and Delphi makes it even easier through the use of the `threadvar` construct. Let's examine how to manage thread-local variables.

Thread-local variables the Delphi way

The simplest way to deal with thread-local variables is through the use of Delphi's keyword `threadvar`. The developer acts as if a global variable is being declared, but instead of using the `var` keyword, the keyword `threadvar` is used. For example, the following code snippet declares a thread-local variable called *szMyString*:

```
threadvar
```

```
  szMyString: PChar;
```

The keyword `threadvar` can only be used at the unit level. In other words, a function cannot have local variables declared as thread-local. The reasoning behind this is clear: Local variables are intrinsically local to the thread in which they were called. The only time a "thread-local" variable needs to be used is when a sharable resource is being discussed, and sharable resources are declared outside the scope of procedures and functions.

Thread local variables the API way

In the section on threading (see section Thread-level initialization and finalization {anchor link to Thread-level initialization and finalization, this document }), we point out some problems with using `threadvar`, so it is important to note how Windows handles thread-local access.

There are four routines involved in managing thread-local variables:

Function: DWORD **TlsAlloc**

> Allocate a thread-local index, this index is used to access a thread-local variable. Allocating an index is basically equivalent to declaring a `threadvar`.
>
> It returns `$FFFFFFFF` when an error occurs; otherwise, it returns a valid thread-local index.

Function: BOOL **TlsFree** *(DWORD dwTlsIndex)*

> Free up a thread-local index. This is used when the thread-local variable referenced by the thread-local index is no longer needed.
>
> It returns `True` when a thread-local index is successfully freed.

Function: Pointer **TlsGetValue(DWORD** *dwTlsIndex)*

> Return the thread-local 32-bit value indexed by *dwTlsIndex*. The value *dwTlsIndex* must have been previously allocated using `TlsAlloc`.

Function: Bool **TlsSetValue(DWORD** *dwTlsIndex, Pointer lpvTlsValue)*

> Set the 32-bit value indicated by this thread-local index to the value specified in *lpvTlsValue.*

The below code snippet shows how these functions work together:

```
var
  hTLSValue: DWORD;

...

hTLSValue = TlsAlloc;
if (hTLSValue = $FFFFFFFF) then
  (* raise an exception or something *)

...

TlsSetValue(hTLSValue, Pointer(100));
```

```
...

ShowMessage(IntToStr(Integer(TlsGetValue(hTLSValue))));

...

TlsFree(hTLSValue);
```

In the next section (see section Thread-level initialization and finalization {anchor link to Thread-level initialization and finalization, this document }), we will show how FreeUDFLib uses the Windows API to manage its thread-local variables, so we will wait until then to illustrate any examples.

Thread-level initialization and finalization

In general, DLLs do not create threads, and in the case of building UDFs, this is no exception; however, InterBase *does* create threads, and it is essential that the DLL knows when a thread is created and when a thread is closed.

Initial inspection of Delphi indicates that the `initialization` and `finalization` sections of a Delphi unit are prime candidates for thread-level initialization and finalization. Further inspection reveals that these sections are only fired when the library is loaded and when it is freed, respectively. Good try, but not good enough.

Delphi defines a variable *DllProc*. *DllProc* is a procedure pointer, and by assigning a procedure to *DllProc*, the DLL can perform actions whenever an attached application creates or destroys a thread.

A DLL entry-point procedure is declared like this:

```
procedure LibEntry(Reason: Integer);
```

The actual name of the procedure is irrelevant. It is merely important to note that a libary entry procedure gets a single argument, *Reason*, which indicates why the procedure is being called.

In Delphi, there are three possible *Reason*'s for a `DllProc` to be called:

1. *Reason* = `DLL_THREAD_ATTACH`. Whenever a thread is created in an attached application, `DllProc` will be called with this reason. This gives the DLL an opportunity to initialize any thread-local variables.
2. *Reason* = `DLL_THREAD_DETACH`. Whenever a thread is being closed in an attached application, `DllProc` will be called with this reason. This gives the DLL an opportunity to free up any resources used by thread-local variables. Take care! Suppose that an application starts some threads; it then loads the DLL. The DLL is never explicitly told that those threads are executing (`DllProc` will never be called with the `DLL_THREAD_ATTACH` argument); however, if those threads exit gracefully, the DLL will be informed that they are closing. This means that the DLL is potentially responsible for cleaning up uninitialized data.

3. *Reason* = `DLL_PROCESS_DETACH`. Whenever the calling application unloads a library, `DllProc` will be called with this reason. This is exactly equivalent to the finalization section of a Delphi unit, so it is irrelevant to our discussions.

Let's study some examples:

Open up the project `` `Playing with threads/Dll1.dpr' ``, and take a look at `` `Playing with threads/Dll1Code.pas' ``. Towards the bottom of the file is the following code:

```
procedure DllEntry(Reason: Integer);
begin
  case Reason of
    DLL_THREAD_ATTACH: begin
      tlObject := TTestObject.Create;
      DllShowMessage(tlObject.ObjectName);
    end;
    DLL_THREAD_DETACH: begin
      (* Uninitialized data is guaranteed to be nil. *)
      if (tlObject = nil) then
        DllShowMessage('Object is nil.')
      else
      (* and we've guaranteed that initialized data has an object *)
        DllShowMessage(tlObject.ObjectName);
      tlObject.Free;
    end;
  end;
end;

initialization

  IsMultiThread := True;
  DllProc := @DllEntry;
  tlObject := TTestObject.Create;

finalization

  (* Uninitialized data is guaranteed to be nil. *)
  if (tlObject = nil) then
    DllShowMessage('Object is nil.')
  else
  (* and we've guaranteed that initialized data has an object *)
    DllShowMessage(tlObject.ObjectName);
  tlObject.Free;

end.
```

As this code snippet shows, Dll1 can easily respond to the creation of threads in a calling application.

To further the reader's understanding of these entry point functions, and to demonstrate a problem with Delphi's `threadvar` construct, the reader should study `` `Playing with threads/Dll1.dpr' ``, `` `Playing with threads/Dll2.dpr' `` and `` `Playing with threads/Example.dpr' ``, all distributed with this article. The two DLL projects are identical, with the exception that `` `Playing with threads/Dll1.dpr' `` demonstrates the use of `threadvar`, and `` `Playing with threads/Dll2.dpr' `` demonstrates the use of the direct Windows API thread-local storage system calls.

The application `` `Playing with threads/Example.dpr' `` allows the user to load either `` `Playing with threads/Dll1.dll' ``, or `` `Playing with threads/Dll2.dll' ``, and fidget with threads.

Try this example:

1. Run `` `Playing with threads/Example.exe,'. ``
2. Click on the "Load library function". Since, by default, the first radio button ("Delphi's threadvar") is checked, `` `Playing with threads/Dll1.dll' `` will load.
3. Click "Create new thread".
4. Click "Create new thread" again.
5. Click on the first thread listed, and click "Close selected thread".
6. Hmmm... Access violation?
7. Exit the application, and reload it. Go through the exercise all over again, but this time, ensure that the second radio button is checked, (ensuring that `` `Playing with threads/Dll2.dll' `` will be loaded). Access violations using the Windows API calls?

This example illustrates two things. First of all, it demonstrates how a DLL can respond automatically to the creation and destruction of threads. Second, it shows that the use of `threadvar` isn't entirely safe, but that directly using Windows API calls resolves the problem.

Before moving much further, the reader should take care that the concepts illustrated in each of these projects (`` `Playing with threads/Dll1.dpr' ``, `` `Playing with threads/Dll2.dpr' ``, and `` `Playing with threads/Example.dpr' ``) are well understood.

A discussion of Solution #3

As was mentioned before, InterBase 5.0 introduces the `free_it` keyword, thus allowing the UDF developer to use dynamically allocated memory for the return of strings and dates.

Aside from the author's contention that this is sloppy, and that it won't work with versions of InterBase previous to 5.0, this is a fully supported and "sponsored" technique for returning strings to InterBase. (see section Solution #3: Returning dynamically allocated strings {anchor link to Solution#3, this document}) So, sloppy or not, we must "face the music", and explore returning dynamically allocated strings to InterBase.

Memory allocation issues

Oddly enough, the Windows version of InterBase is compiled using Microsoft's C-compiler (MSVC). Without getting into a discussion as to *why* they chose this compiler, suffice it to say that InterBase expects dynamically allocated memory to be allocated using MSVC's `malloc` routine.

MSVC's `malloc` routine handles memory allocation in a manner "all its own". That is, we can't rightly infer *how* it manages memory, but it certainly does *not* allocate memory in the same fashion as Delphi. So, a Delphi function that tries to dynamically allocate memory using `GetMem` or the Windows system call `GlobalAlloc` will most certainly cause problems with InterBase if used in conjunction with the `free_it` keyword.

This problem is resolved by making use of the fact that MSVC applications must be distributed with the run-time MSVC library, `` `msvcrt.dll' ``. If InterBase or the InterBase client is installed on a system, then this DLL is installed on your system as well.

By making the following declaration in your Delphi UDF library,

```
function malloc(Size: Integer): Pointer; cdecl; external 'msvcrt.dll';
```

you will allow Delphi to make use of MSVC's `malloc` routine, so that the `free_it` keyword can be used.

Working through an example

Take a look at the project `` `UDF Test 1/UDFTest1.dpr' ``, and open `` `Funcs.pas' ``.

`` `Funcs.pas' `` declares the `malloc` routine, and it implements a very silly function called `CopyString`. Let's take a look at `CopyString`:

```
function CopyString(sz: PChar): PChar; cdecl; export;
var
  szLen: Integer;
begin
  szLen := 0;
  while (sz[szLen] <> #0) do Inc(szLen);
  Inc(szLen);
  result := malloc(szLen);
  Move(sz^, result^, szLen);
end;
```

Quite simple, `CopyString` allocates enough space for the passed string (`sz`) plus the null terminator, and it copies the string.

The declaration for CopyString is as follows:

```
declare external function CopyString
  cstring(64)
  returns cstring(64) free_it
  entry_point 'CopyString' module_name 'UDFTest1.dll';
```

Open InterBase's WISQL tool, and connect to `` `UDF1.gdb' ``. After connecting, do the following:

1. Execute the above "declare external function", and commit.
2. Execute the following query:

   ```
   select CopyString('Hello world') from rdb$database
   ```

A discussion of Solution #4

It is a bit frustrating to think that we can write a UDF library that doesn't support as many versions of InterBase as desired. And, if you agree with the authors that the "sloppy" approach just *won't* do, then this section might be for you.

As was briefly alluded to above, a UDF should do its own housekeeping, and when possible, it should probably also try to avoid "dirtying the house" at all. InterBase's external function declaration syntax includes the ability for InterBase to pass the result buffer to the UDF, so that UDF can take the "high road", and be a gracious guest, providing information only, but not cluttering up InterBase's house at all.

In addition, this method for declaring functions makes it possible to avoid using the `free_it` keyword, so that UDF libraries built in this way can be used in versions of InterBase previous to version 5.0. (see section Solution #4: Making InterBase do the work {anchor link to Solution#4, this document})

How is this done? Clearly, this is best illustrated through an example. Open the project `` `UDF Test 2/UDFTest2.dpr' ``, and open `` `Funcs.pas' ``.

In `` `Funcs.pas' `` we implement the following function:

```
function CopyString(sz, szRes: PChar): PChar; cdecl; export;
begin
  result := szRes;
  while (sz^ <> #0) do begin
    szRes^ := sz^;
    Inc(sz); Inc(szRes);
  end;
  szRes^ := sz^;
end;
```

Now, in InterBase, we declare it as follows:

```
declare external function CopyString
  cstring(64),
  cstring(64)
  returns parameter 2
  entry_point 'CopyString' module_name 'UDFTest2.dll'
```

And finally, in `` `UDF2.gdb' ``, we can test this example by declaring the external function and using it in a silly select statement:

```
select CopyString('Hello World') from rdb$database
```

Conclusions

It is now time to turn our attention to a working example of a UDF library--FreeUDFLib. FreeUDFLib implements its UDFs using the thread-local method described above. With the change of a simple compiler "define", FreeUDFLib can also behave like `free_it` wants it to behave.

The method mentioned above, which allows the UDF developer to avoid all issues of memory allocation and deallocation is also quite elegant.

In conclusion, the `free_it` keyword makes it possible to create fully supportable UDFs that will run cleanly (if declared in InterBase correctly) and safely in InterBase's multi-threaded environment.

FreeUDFLib demonstrates the use of MSVC's `malloc` to dynamically allocate free_it'able memory, and along the way it provides some very convenient functions.

Once again, (and for the last time) the author contends that the free_it approach is sloppy, and given the two proposed solutions of either returning thread-local memory or pushing data into passed parameters, it should be unnecessary.

About the author

Gregory Deatz is a senior programmer/analyst at Hoagland, Longo, Moran, Dunst & Doukas, a law firm in New Brunswick, NJ. He has been working with Delphi and InterBase for approximately two and a half years and has been developing under the Windows API for approximately five years. His current focus is in legal billing and case management applications. He is the author of FreeUDFLib, a free UDF library for InterBase written entirely in Delphi, and FreeIBComponents, a set of native InterBase components for use with Delphi 3.0. Both of these tools can be found at at `http://www.interbase.com/download`. He can be reached via e-mail at `gdeatz@hlmdd.com`, by voice at (732) 545-4717, or by fax at (732) 545-4579.

This document was generated on 6 August 1998 using the texi2html translator version 1.51a.