Run-Time Location Information In Delphi 2

by Vitaly Miryanov

It's happened to us all: the user rings up and says *"Hey, this new program you wrote for us doesn't work! I tried using it and up popped a message that says there's an error in module bigapp.exe at some funny address with letters and numbers in..."*. Aren't these raw addresses a pain? Wouldn't it be much nicer to have your apps provide a source code file name and line number where the error is located? Of course! Well, read on...

Using out-of-the-box tools, there are only two ways to solve this problem. The first approach is to create a detailed map file with full line number information and then manually look up the file name and line number for an error address. The second approach is to keep the unchanged binary unit files (.DCUs) and object files that were used to build the program in order to use the Delphi compiler's `Find Error` facility. This involves storing a large number of files with the potential danger of updating any of them with a recompilation, in which case the `Find Error` mode becomes useless.

Both approaches have obvious deficiencies so it would be nice if the program itself contained the source file name and line number information embedded in the executable, so it could report them when required. This is a similar idea to Run-Time Type Information (RTTI), so I will call it *Run-Time Location Information* (RTLI). It gives a programmer the opportunity to query the source file name and line number information for a particular code address inside the program at run-time. C++ programmers have enjoyed this kind of information for quite a while. Although the Delphi 2 compiler shares the same back end as the Borland C++ compiler, RTLI is still unavailable in Delphi.

Since the compiler does not provide it, let us find out where can we get it ourselves. For an application or a dynamic link library, the Delphi compiler normally generates Delphi compiled unit files (DCUs), an executable file and a map file (if requested). A Delphi compiled unit file contains the entire information on a unit. However, not only has the format of the DCU files never been documented, it is also compiler version dependent and DCU files do not contain all the addresses assigned by the linker. The executable file itself does not provide any line number information, but debugging information intended for a standalone debugger (TD32) does. It looks very suitable at first glance. It is already attached to the executable file, and although the format is version dependent the core of it remains the same. The downside is that it is very big: the size of the debugging information for a minimum program that uses the `Windows` unit is over 300Kb! Another disadvantage is that when you distribute the program executable with the debugging information attached, you give away a lot of information about the executable, which might be undesirable in some cases. Though it is possible to use the debugging information as raw data, transferring it to a more suitable form, the format is rather complicated and is subject to change in future compiler versions. So ruling out these two sources we end up with just one: the map file.

Studying Delphi MAP Files

Few people look at the map files generated for their Delphi application, considering them either too technical or too boring. In fact, map files contain very useful information which cannot be found anywhere else. The format of the map file is standard for all 32-bit Borland compilers and linkers, including Delphi, Borland C++ and TLINK32. Listing 1 (next page) provides the source code for a simple example and Listing 2 shows the format of the detailed map file generated.

The first part of a map file contains a general map of segments, which for a typical Delphi application comprises three segments: `.text`, `.data` and `.bss`. The `.text` segment contains all program code, virtual method tables, run-time type information and constants, including string literals. The `.data` segment contains all initialized data, both writeable typed constants and initialized global variables. The `.bss` segment historically derives its name from the DOS world, meaning *Below Stack Segment*, although this is not true for the 32-bit flat model. It contains all uninitialized data such as uninitialized global variables. Besides a name, each segment has an additional attribute called class. In real mode, the class of the segment is used to combine segments with different names in the same final segment in the executable file. In protected mode, it is mainly used to indicate whether the segment is a code segment or a data one. The class `CODE` is used for code segments, all other class names are normally used for data segments.

The next section of the map file contains a detailed map of segments. It shows smart linking in action: only segments used by the program are listed. This is the only place where you can see which units are linked into the executable. For every non-zero segment that was taken from each unit there is an entry in the detailed map segment. Apart from segment and class names we can see a new

concept: a group of segments. A group is used to combine several segments with different names and/or classes into one single segment in the executable file. The .text segment is not a part of any group so it is separated in its own code segment in the executable file.

The .data and .bbs segments are included in the data group DGROUP so they are combined into one data segment in the executable file. The strange looking ACBP byte value stands for *Alignment Combination Big inPage* and the hard-wired value of A9 means that all segments in Delphi are public double word aligned 32-bit segments.

The order of the code segments in this list identifies the order in which the initialization parts of the units are executed. This order might be quite confusing. For example, consider our Foo example program that uses three units. In which order will the initialization parts of the units be executed? Judging from the Delphi documentation, they will be executed from left to right: Unit1, Unit2 then Unit3. But in fact that is not true if, for example, Unit1 uses Unit2 but not Unit3. In this case the order would be Unit2, Unit1 then Unit3.

For a complex program containing a lot of cross and circular unit references it is almost impossible to guess the order. So wherever the order is important, you can easily find it from the detailed map of segments. You can change the order by rearranging the uses clause of the primary project file and dependent units.

The next two sections of the map file give a list of public symbols with their addresses, sorted by name in the first section and by address in the second one. The only notable exception is the TlsLast variable that is not sorted properly by its address, which has a ridiculously large value. After that there is line number information for each unit compiled in the {$D+} state followed by the program entry point address.

For the purpose of our exercise with RTLI we will be interested in the .text segment which in Delphi has a hard coded index of 1. The most important of all is the line number information, which we will use to generate RTLI. However, in most cases, the line information is not enough. Some units are not compiled in the {$D+} state, for example the System unit. If you don't have the source code for these units you cannot recompile them with line number information enabled. So, in addition to line number information we will use the detailed map of segments which gives us a clue as to the unit in which the address of interest is located. To get a more precise location we will also use the public symbol table for the symbols located in the .text segment. This might give us an indication as to the procedure or function containing the address.

Building An RTLI Generator

First of all we need to set up the main design goals for the RTLI generator. RTLI should be very compact to minimize the size overhead in the executable. RTLI will be retrieved only rarely, so the lookup time is not that significant. Another important requirement is that functions retrieving RTLI should not use large amounts of memory and should be robust and reliable, since they might be called in critical sections of code, such as exception handlers.

It is possible to integrate the RTLI generator with the Delphi IDE using the Tools API. However, for the sake of simplicity I will implement it as a command line utility. This also allows you to use it in conjunction with any other command line utilities such as MAKE, or to invoke it directly from the Tools menu of the Delphi IDE.

The RTLI generator should perform the following steps:

- Make the project with the detailed map file information enabled;
- Parse the MAP file, extracting line number and public symbol information, and then create a binary file containing the extracted information;
- Embed this binary file in the executable file.

➤ *Listing 1*

```
program Foo;
uses Unit1, Unit2, Unit3;
begin
  DoSomething;
end.
```

➤ *Listing 2*

```
 Start         Length     Name                   Class
 0001:00000000 00000C54H .text                  CODE
 0002:00000000 00000070H .data                  DATA
 0002:00000070 0000045CH .bss                   BSS

Detailed map of segments

 0001:00000000 00000BF0 C=CODE     S=.text    G=(none)   M=System   ACBP=A9
 0001:00000BF0 00000010 C=CODE     S=.text    G=(none)   M=Unit2    ACBP=A9
 0001:00000C00 00000014 C=CODE     S=.text    G=(none)   M=Unit1    ACBP=A9
 0001:00000C14 00000010 C=CODE     S=.text    G=(none)   M=Unit3    ACBP=A9
 0001:00000C24 0000002D C=CODE     S=.text    G=(none)   M=Foo      ACBP=A9
 0002:00000000 0000006E C=DATA     S=.data    G=DGROUP   M=System   ACBP=A9
 0002:00001000 0000045C C=BSS      S=.bss     G=DGROUP   M=System   ACBP=A9

 Address             Publics by Name

 0001:000009BD       @Append
 0001:00000751       @Assign
........[ Skipped ].........
 Address             Publics by Value

 0002:FFBFE004       TlsLast
 0001:00000000       TextStart
 0001:000001F0       @_IOTest
........[ Skipped ].........
Line numbers for Unit2(Unit2.pas) segment .text
     8 0001:00000BF0     9 0001:00000BFF

Line numbers for Unit1(Unit1.pas) segment .text
    13 0001:00000C00    16 0001:00000C04    17 0001:00000C13

Line numbers for Unit3(Unit3.pas) segment .text
     8 0001:00000C14     9 0001:00000C23

Line numbers for Foo(foo.pas) segment .text
     5 0001:00000C24     6 0001:00000C43     7 0001:00000C48

Program entry point at 0001:00000C24
```

The next sections describe these steps in more detail.

Recompiling From The Command Line

To make the project we need to use the same settings that were used in the IDE while designing the project. How can we do that? The Delphi 2 IDE creates a <ProjectName>.DOF file containing the compiler/linker options. It is a plain Windows style INI file. The `Compiler`, `Linker` and `Directories` sections contain all the settings necessary to make the project: we just need to translate the INI file settings to the corresponding command line switches used by the Delphi command line compiler DCC32.EXE. There is one exception though: the `ExeDescription` field in the `[Linker]` section does not have any command line equivalent. The obvious solution is to use the `{$DESCRIPTION 'Text'}` compiler directive instead of setting the executable file description in the `Linker` page. Note that the old `{$D Description}` directive from Borland Pascal and Delphi 1 is not present in Delphi 2 or 3, although no compiler error is produced.

The command line compiler uses a configuration file DCC32.CFG to read the initial settings. Although it is possible to override some of the initial settings on the command line, the directory information, such as the unit directory search path, is initially taken from the DCC32.CFG file. For this reason, the RTLI generator creates a new DCC32.CFG file containing the converted options, rather than supplying them on the command line.

The part of the RTLI generator that performs the translations of the compiler/linker settings and runs the command line compiler can be invoked separately. As a side effect, this lets us use the RTLI generator just for recompilation of Delphi projects. It can be very useful if you have lots of projects, since you can automatically recompile them from a batch file, or as part of an advanced MAKE file.

Parsing The MAP File

The parsing stage is straightforward. The RTLI information is placed in three tables. The information from the detailed map of segments is placed in the unit table. The public symbol list sorted by value is used for building the public symbol table. The line number information is used to create the line number table. Each table is optimized to use the least possible space. For example, instead of storing the offset information for every public symbol, the encoded difference between the current and the previous offset is stored. This technique uses 1 or 2 bytes for every offset instead of the 4 bytes needed for the offset itself. A similar approach is used with the line number information, where the encoded line and offset differences are stored in 1, 3, 5 or 7 bytes.

To frustrate any unhealthy desire of a very sad end user to look in the executable file and find out the source file names and symbolic information, all the names are hidden by `XOR`ing them with a variable byte sequence.

There is one important caveat I need to add. I hit some strange problems with the Delphi 2.00 and 2.01 compilers: on some projects they do not produce correct line number information in the map file. The problem occurs in units where a source file is included using the `{$I FileName}` directive. The compiler reports line numbers shifted by a random number. For example, in a unit with 500 source lines and 250 lines in an include file that does not generate any code, line numbers in the map file were reported to start from line 600 and end with line 1000! That is one wierd bug in the Delphi compiler...

So, the message is: do not use the `$I` compiler directive in your code.

Integrating RTLI In The Executable

How can we add the binary RTLI file to the executable file? The easiest solution is just to append it to the end of the executable. However, it is not a good practice to do so, since the format of the executable file is standard and does not allow for user defined data. To avoid that, we will store the RTLI as a resource file and then embed it as a resource in the executable file, so that we can make use of the system functions for loading resources. Fortunately, appending an additional resource does not change any addresses inside the executable file, so our RTLI remains valid.

The RTLI generator produces a standard Win32 .RES file, but the .RLI extension is used to avoid the clash with the Delphi generated .RES file for the project. The .RLI file contains only one resource of type `RC_DATA`.

The Delphi design environment also uses `RC_DATA` resources for storing resource streams in the .DFM file, so to avoid potential problems with assigning the same name to the `RC_DATA` resource as was already used by the designer, the `RC_DATA` resource for RTLI is identified by a magic number, rather than by name.

We can use the Borland Resource Compiler supplied with Delphi to append the resource file to the executable file. Easy? Not quite. Although the resource compiler BRC32.EXE displays in its title that it is a compiler/binder, in fact, it launches TLINK32 to bind binary resource files to the executable. For some obscure reason TLINK32 is not shipped with Delphi, so BRC32 does not work (a *Could not spawn program: TLINK32.EXE* error message is reported). How can we add the resource file to the executable without getting into the huge trouble of parsing and modifying the complex structures of the Win32 executable file? Why not use the Delphi compiler to do it? In the main source file of the project we will include the following lines of code:

```
{$IFDEF BindingRTLI}
  {$R *.RLI}
{$ENDIF}
```

When the RTLI generator makes the project with the detailed map file it does not define the `BindingRTLI` conditional symbol. After parsing the map file, it creates the resource file <ProjectName>.RLI and runs the compiler again with the `BindingRTLI` conditional symbol

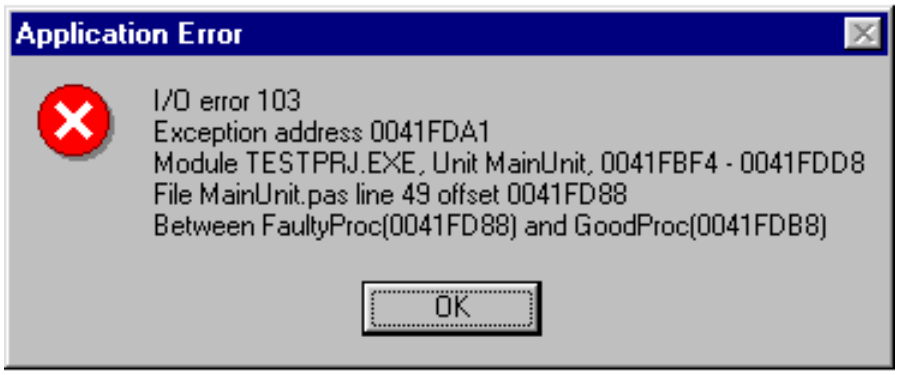
defined, which forces the compiler to bind in the RTLI resource file. Since nothing has changed in the project, the second recompilation leaves all the addresses in the executable file exactly the same as they were reported in the map file.

The Program Interface

The `LIPrgInt` and `LIVCLInt` units provide the interface for accessing RTLI. `LIPrgInt` supplies the generic `GetLocationInfo` function for retrieving RTLI in any program or dynamic link library. It takes a code pointer as a parameter and fills a record containing various information extracted from RTLI, such as a unit name, file name, line number and names of two public symbols between which the specified code address is located. It is worth noting that the first public symbol identifies the procedure or function that contains the requested address *only* when the unit is compiled in the `{$D+}` state. For a unit compiled in the `{$D-}` state, the map file contains a list of procedures and functions declared in the `interface` section only, so the two public symbols returned are the two closest procedures or functions exposed in the `interface` of the unit.

The `LIVCLInt` unit is especially designed for Delphi VCL programs or DLLs. The source for this unit is shown in Listing 3.

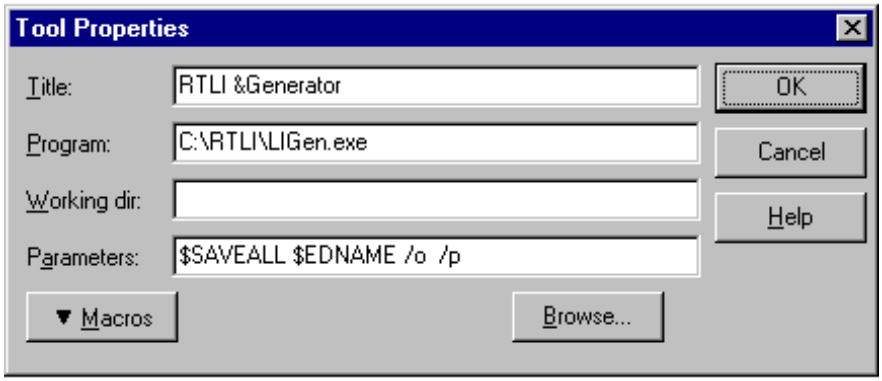
The `RTLIShowMessage` procedure defined in this unit is an equivalent to the `ShowException` procedure found in the Delphi `SysUtils` unit, but displays more detailed information based on RTLI. The initialization section of the `LIVCLInt` unit automatically replaces the default VCL exception message boxes with the RTLI enabled one. In order to do that, the `OnException` property of the `Application` object is set to the RTLI exception handler. Similarly, the `ExceptProc` which is normally initialized in `SysUtils` is hooked by the `RTLIExceptHandler` to display detailed exception information. Since the `LIVCLInt` unit uses the `SysUtils` unit, the `initialization` section of `LIVCLInt` will be executed after the intialization of `SysUtils`, ensuring that the



➤ *Figure 1*

```
unit LIVCLInt;
interface
procedure RTLIShowException(ExceptObject: TObject; ExceptAddr: Pointer);
implementation
uses LIPrgInt, SysUtils, Forms, Windows;
type
  TExceptionHandlerObject = class
    constructor Create;
    procedure HandleExceptions(Sender: TObject; E: Exception);
  end;
var
  PrevInitProc: Pointer;
  ExceptionHandlerObject: TExceptionHandlerObject = nil;
procedure RTLIShowException(ExceptObject: TObject; ExceptAddr: Pointer);
var
  LocInfo: TLocInfo;
  Msg,ModuleName: String;
  Buffer: array[0..259] of Char;
begin
  GetLocationInfo(ExceptAddr, LocInfo);
  if ExceptObject is Exception then
    Msg := Exception(ExceptObject).Message
  else
    Msg := '**Unknown**';
  GetModuleFileName(HInstance, Buffer, SizeOf(Buffer));
  ModuleName := ExtractFileName(Buffer);
  with LocInfo do begin
    if liFileName = '' then
      liFileName := '**Unknown**';
    if liUnitName = '' then
      liUnitName := '**Unknown**';
    Msg := Format('%s'                                    + ^J +
                  'Exception address %8.8x'               + ^J +
                  'Module %s, Unit %s, %8.8x - %8.8x'     + ^J +
                  'File %s line %d offset %8.8x'          + ^J +
                  'Between %s(%8.8x) and %s(%8.8x)',
      [Msg, Integer(ExceptAddr), ModuleName, liUnitName, liUnitBegOfs,
       liUnitEndOfs, liFileName, liLineNo, liLineOfs, liPubSym1Name,
       liPubSym1Ofs, liPubSym2Name, liPubSym2Ofs]);
  end;
  MessageBox(0, PChar(Msg), 'Application Error',
    mb_Ok or mb_IconStop or mb_TaskModal);
end;
constructor TExceptionHandlerObject.Create;
begin
  Application.OnException := HandleExceptions;
end;
procedure TExceptionHandlerObject.HandleExceptions(Sender: TObject; E: Exception);
begin
  RTLIShowException(E, ExceptAddr);
end;
procedure RTLIInitProc;
begin
  ExceptionHandlerObject := TExceptionHandlerObject.Create;
  if Assigned(PrevInitProc) then
    TProcedure(PrevInitProc);
end;
procedure RTLIExceptHandler(ExceptObject: TObject; ExceptAddr: Pointer);
begin
  RTLIShowException(ExceptObject, ExceptAddr);
  Halt(1);
end;
initialization
  PrevInitProc := InitProc;
  InitProc := @RTLIInitProc;
  ExceptProc := @RTLIExceptHandler;
finalization
  ExceptionHandlerObject.Free;
end.
```

➤ *Listing 3*



➤ *Figure 2*

ExceptProc variable will be set to our RTLI exception handler and not to the standard SysUtils one. A typical RTLI enabled exception message box is shown in Figure 1.

Using The RTLI Generator

The RTLI generator (LIGEN.EXE, found in the SOURCE subdirectory when you have unzipped the archive file on this month's disk) can be used either from the command line or from the Tools menu in the Delphi IDE. To display a list of valid command line parameters, run the RTLI generator with the -? command line switch. The RTLI generator takes one command line parameter: the name of the Delphi project file. The path can be also specified. The extension is ignored as the program will use the proper extension for each file: .DPR for the Delphi project file and .DOF for the Delphi option file. To install the RTLI generator on the Tools menu use the options shown in Figure 2.

The only annoyance is that you have to use View|Project Source option first to bring the source code for the project to the top editor window and then select the RTLI Generator option from the Tools menu. This is because the Delphi IDE does not have a macro for getting the name of the project file, one can only ask for the name of the file located in the topmost editor window.

Summary

To use the RTLI generator from the Delphi IDE all you need to do is:

- ➢ Install RTLI Generator on the Delphi 2 Tools with the parameters shown in Figure 2;
- ➢ Add the LIVCLInt unit (or any other unit that you might implement which makes use of RTLI) to the uses clause of your project and make the source file accessible to the compiler (include the path to it in the Project| Options | Directories |Search Directories if this is required);
- ➢ Add the {$IFDEF BindingRTLI} {*.RLI} {$ENDIF} line to the source code for the project;
- ➢ Select View|Project Source;
- ➢ Select Tools|RTLI generator.

It is important to note that after you produce the final executable containing RTLI you should not debug it using the IDE debugger or run it from the IDE. If you do so, the IDE compiler will recompile the project first and as a result RTLI will be lost. As RTLI is intended to help find the exact source of errors when your application is in the hands of the end users, this presents no problem.

And remember that you should not use the $I compiler directive in your code, to prevent the problem with incorrect line numbers mentioned earlier.

Delphi 1 And 3?

Although this article is about Delphi 2, I do not see any serious reasons why RTLI cannot be implemented for Delphi 1. There are a number of difficulties though. In order to implement it properly for Delphi 1, it is necessary to take into consideration the segment part of the address as well as the offset one, so the internal structures and encoding mechanism should be slightly modified. The translation of the physical segment selectors to logical segment numbers can be performed by reading the word at offset 0 in this segment, which gives the logical number of the segment. Also the formats of 16-bit map and resource files are different from their 32-bit counterparts.

On the other hand, the RTLI generator does work with Delphi 3, you just need to include the Delphi 3 BIN directory in the PATH environment variable. The only shortcoming is that new features of the Delphi compiler not available in Delphi 2 are not supported. For this reason RTLI for packages is not available. This might be a subject for a future article...

Vitaly Miryanov is originally from the Ukraine and is the author of the *Virtual Pascal for OS/2* Delphi-alike compiler, now marketed by fPrint (UK) Ltd, for whom he now works on this and other projects. Contact Vitaly as vitaly@ibm.net