BorDebug: Return Of The Giant

Making sense of TD32 debug information

by Hallvard Vassbotn

The Delphi linker has always had the option of including so-called Turbo Debugger (TD32) debug information (on the `Linker` page of the Project Options dialog). The internal IDE Debugger does not normally use this information (Delphi 4 and 5 use it when debugging external DLLs and EXE files), but instead relies on internal compiler structures built during an interactive compile.

External tools, such as Borland's Turbo Debugger[1], do rely on the TD32 information tacked on at the end of the EXE or DLL file to enable symbolic debugging. This information is also used by a number of third party tools, such as Numega's BoundsChecker[2], TurboPower's suite of Sleuth QA[3] tools, Atanas Stoyanov's freeware MemProof memory checking tool[4], AutomatedQA's QTime profiler[5] and Intel's VTune sampling profiler[6].

In this article we will see how we can utilise a relatively new DLL from Borland, BorDebug.DLL, to read and interpret the TD32 debug information in our own applications. This can be used for a number of purposes, although it will be most useful for debuggers and other low-level tools. We will discuss the functionality provided by the BorDebug DLL, present an import unit that gives us access to it from our Delphi applications, look at a set of wrapper classes to simplify the usage and show some simple demonstration programs.

History

Traditionally, it has been very hard to get information on the internal structure of the TD32 format. For many years, it was totally undocumented, not even mentioned in places like Wotsit[7]. There might have been a number of reasons for this: perhaps Borland wanted to protect their own standalone Turbo Debugger from the competition, for example. In any case, this was a major obstacle for people who wanted to develop support for Delphi modules in tools such as debuggers, code analysers, profilers, memory checkers and so on. In contrast, Microsoft have openly documented their own debugging format and even implemented access routines in the IMAGEHLP.DLL (see the Delphi import unit in Source\Rtl\Win\ImageHlp.Pas).

I know that a number of people have been forced to reverse engineer at least parts of the TD32 format to implement their tools. This includes first-rate guys such as Per Larsen of TurboPower (Memory Sleuth, Sleuth QA), Atanas Stoyanov of MemProof and QTime fame, and Stefan Hoffmeister[8], hacker extraordinaire and more recently a Borland employee. After a long period of lobbying Borland, they finally released a long and detailed document to people that specifically requested it in a file called Giant.Txt. I got my copy from Charlie Calvert in the middle of 1998. I later found that mostly the same information had been released in 1993 as part of a product called *Borland Open Architecture Handbook for BC 4.0* (in a file named BC32.TXT).

Internally, the TD32 format is known as the Giant format, presumably because it can handle larger amounts of information than the older 16-bit format used in the old Turbo/Borland Pascal compilers. Some might say it got its name because it makes your EXE files giant-sized ☺.

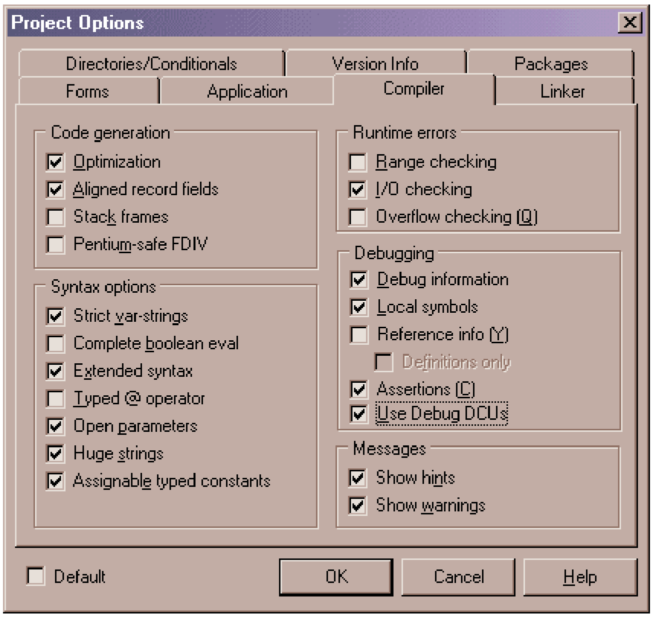
The release of this document was literally a giant step in the right direction. However, the information in the document was very terse, contained no example code and not even C `structs` defining the structure of the information. It would still require hours and hours of studying, trial and error to turn the document into working code that could read the TD32 debug information from a given EXE file.

Then, in March 1999, Borland's Keimpe Bronkhorst contacted a number of interested people, including myself, asking us if we were willing to informally test this new DLL they were developing. The DLL, called BorDebug.DLL, would give relatively easy, albeit low-level, access to TD32 information in any executable module. This was what we had been waiting for! A private mailing list was established and we set out to test and start using this new DLL. Initially, the only interface was through a C header file, but I took on the task of converting it to a Delphi import unit. The result is the BorDebug.Pas unit presented in this article.

In fact, I started writing this article about a year ago, trying to get it published in the August 1999 'Gold' issue of *The Delphi Magazine*. However, at that time, Borland was not ready to publish the BorDebug.DLL, so the article was put on hold, and I had to write a new one[9] in a hurry to fill the reserved pages! Then, in March 2000, Borland's John Thomas released the BorDebug.DLL in a CodeCentral submission[10]. Now the DLL is freely available, I can finally finish off this article. Phew!

Project settings

There are a number of compiler and linker options which can be set in the IDE, the DCC32.CFG file, or on the DCC32 command line, that influence the level of detail of the debug information included. Bring up the Project Options dialog and select the Compiler page, see Figure 1.



Project Options

Code generation
☑ Optimization
☑ Aligned record fields
□ Stack frames
□ Pentium-safe FDIV

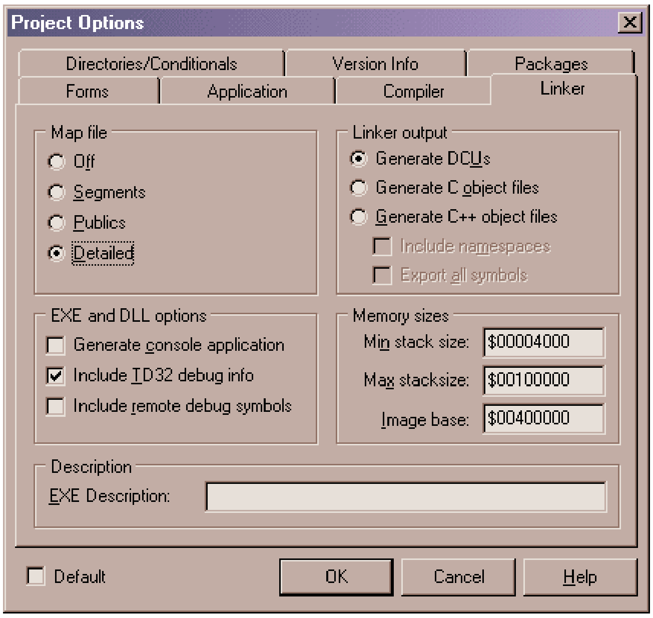
Syntax options
☑ Strict var-strings
□ Complete boolean eval
☑ Extended syntax
□ Typed @ operator
☑ Open parameters
☑ Huge strings
☑ Assignable typed constants

Runtime errors
□ Range checking
☑ I/O checking
□ Overflow checking (Q)

Debugging
☑ Debug information
☑ Local symbols
□ Reference info (Y)
□ Definitions only
☑ Assertions (C)
☑ Use Debug DCUs

Messages
☑ Show hints
☑ Show warnings

□ Default



Project Options

Map file
○ Off
○ Segments
○ Publics
● Detailed

Linker output
● Generate DCUs
○ Generate C object files
○ Generate C++ object files
□ Include namespaces
□ Export all symbols

EXE and DLL options
□ Generate console application
☑ Include TD32 debug info
□ Include remote debug symbols

Memory sizes
Min stack size: $00004000
Max stacksize: $00100000
Image base: $00400000

Description
EXE Description:

□ Default

➤ *Above: Figure 1*

➤ *Right: Figure 2*

Although they are not strictly debug options, the code generation option settings can greatly influence how easy it is to debug the resulting code. Turning off `Optimization` ($O-) makes it easier to follow the code (particularly in the CPU View) and to examine variable contents in after-the-fact situations. Turning on `Stack frames` ($W+) can help the IDE and external tools to find correct return address and parameter information from the stack, helping you show how you ended up in the current routine.

There are also a number of debugging settings. The `Debug information` setting ($D+) is a master setting, determining if the compiled DCU file contains any debug information or not. `Local symbols` ($L+) turns on information for local variables within routines and symbols only visible in the `implementation` part of the unit.

`Reference info` ($Y+) and the related `Definitions only` ($YD) settings affect the level of information available to the browser windows in the IDE. These settings include extra information about where symbols were declared and referenced. The BorDebug API has not implemented access to this information in the current version. To get at it, you would have to glean the information yourself.

`Assertions` ($C+) include the code for any `Assert` calls in the compiled code, this is a very useful technique for catching problems early. Finally, the `Use Debug DCUs` setting instructs the linker to use the debug version of the RTL and VCL pre-compiled DCUs (in your Delphi\Lib\Debug directory). This can make it easier to track problems that interact closely with the RTL and VCL. Note that this is really a linker setting, although it appears in the `Compiler` page. Unless you plan to distribute your DCU files (for a shareware product, for instance), you probably want to keep both `Debug information` and `Local symbols` checked at all times.

Next, select the `Linker` page of the Project Options dialog. Normally all the DCUs of your project contain full debug information. That debug information can be included in the final executable file by checking the `Include TD32 debug info` setting. The `/V` option of the command line compiler (DCC32.EXE) has the same effect. The BorDebug.DLL, and thus all the example code presented in this article, presume that the EXE files have been compiled with this setting. Note that this will typically increase the size of your executable considerably. You probably don't want to ship with debug information to your clients. Also, there is a tool shipped with C++Builder called TDStrp32.EXE[11] that can take an EXE file with debug information and strip the debug info out to a separate .TDS file. This way you don't have to recompile the executable and most tools will still be able to load the debug information from the .TDS file.

The `Linker` page also contains options to generate a text .MAP file. The generated file contains most of the debug information that will be linked into the executable when you select the `Detailed` option. However, it does not contain information about parameters or local variables. Some tools (such as HVEST[9]) that don't know how to deal with the native TD32 format use this .MAP file instead.

Files

Included on the disk you will find BorDebug.DLL and the original C header file, BorDebug.h. The header file contains additional information on how to use each API call. I decided not to duplicate this documentation in the Delphi import file, BorDebug.Pas (it would be too much extra work to keep the two files in sync with each other). So if you need documentation for a certain BorDebug API call, search for the name in BorDebug.h. I will include the most relevant information and a higher-level overview in this article.

BorDebug Concepts

To get a proper understanding of the TD32 format and the BorDebug API, we must first establish the definition of some basic concepts and relate those concepts to what we know as Delphi developers.

They say an image is worth more than a thousand words, so to keep this article short (☺), I've included an overview image of how the Pascal, DCU and EXE-files are related with the different types of debug information in Figure 3.

This figure is most easily interpreted from the right to the left. At the bottom left, we have the source files for two units in the project, `Unit1` and `Unit2`. `Unit1` is a plain, single file, unit, while `Unit2` consists of an include file and an assembly file in addition to the mandatory Pascal source file. A standalone assembler (such as TASM[12]) is used to turn the .ASM file into an .OBJ file. This OBJ file is sucked into the generated .DCU file by using the `$L` directive. From a debug info point of view, all we will see is that the unit has three source files contributing to its contents: Unit2.Pas, Unit2.Inc and Core.ASM.

We compile the source files and, provided we have used the right settings (`$D+,L+`), the generated .DCU files have debug info tacked onto the end. You can see that the orange and green DCU files both have a grey box representing the debug information for each unit. The red System.DCU unit at the top doesn't contain any source-level debug information, so we will not be able to single step into it.

Because the Delphi compiler and linker are so fast, and there is no simple way to just compile the units without linking the final executable, many Delphi developers are unaware that there is a separate link step. The linker takes the generated .DCU files, does some magic, and then constructs a new .EXE file. The yellow and cyan areas in the figure represent the code and data segments respectively.

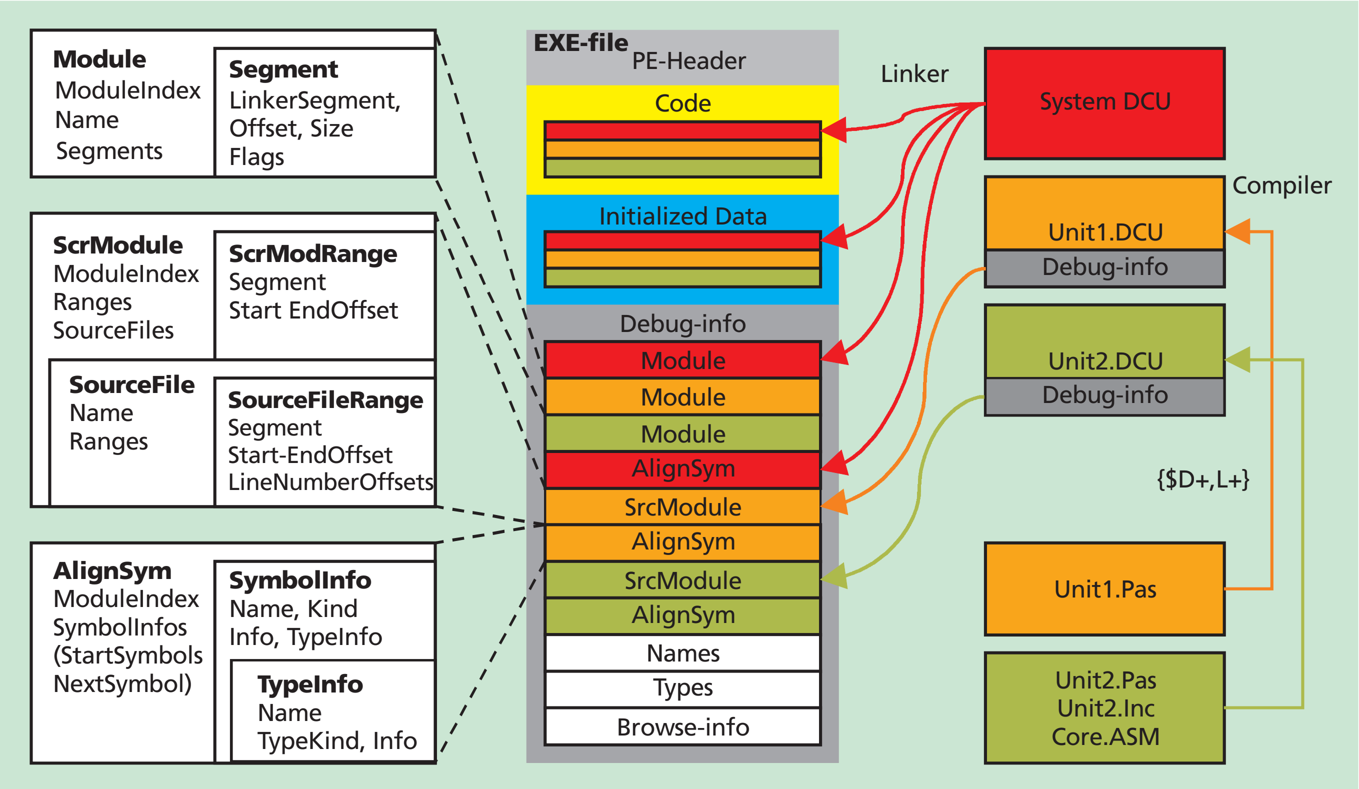
As you can see, in this example all three units have both code and data (shown as red, orange and green boxes inside the .EXE file). There is also a segment for uninitialized global variables, but these don't show up in the .EXE file (other than the `size` field in the PE header). It turns out that all Pascal units always contribute to the code and uninitialized data segment, even though they don't declare any explicit code or variables. This means that a unit containing only constants or simple types still takes up some code and data space. The reason for this is the automatically generated initialization and finalization code and the global counter variable this code uses (see an earlier article[13] for more details).

If the `/V` option is turned on (*Include TD32 debug info*), the linker aggregates the debug information it finds in the .DCU files and adds it to the end of the .EXE file (represented by the grey box). Because the System.DCU file does not have any debug information in it, the linker only generates one `Module` section and one `AlignSym` section for it (the two red boxes inside the grey debug info box).

On the other hand, both the `Unit1` and `Unit2` units were compiled with debug information, and thus they both get `SrcModule` sections in addition to the mandatory `Module` and `AlignSym` sections (the orange and green boxes).

Now we have a complete .EXE file with TD32 info attached. It will run like a normal .EXE file, taking no extra memory at runtime, because the OS loader will simply ignore the debug information. However, when we load the .EXE into a TD32-aware tool, the BorDebug.DLL (or corresponding

➤ *Figure 3*



custom code) will be able to find the debug information.

Finally, the white boxes at the left of the figure give more details about what kind of information the debug sections contain. It will be useful to refer to these while I explain the structure of the TD32 information.

Gigantic Structure

As I have already discussed, the TD32 debug information can be tacked on to the end of the executable file, or it can be placed in a separate .TDS file. To be able to read the debug information, we must of course tell BorDebug where to find it. We do this by calling the BorDebugRegisterFile function. If it succeeds, this returns a handle that we must use in all subsequent calls to the BorDebug routines. When we are done, we call the BorDebugUnregisterFile API to close the file and release any associated resources. From the highest level point of view, the TD32 information just consists of a number of subsections.

SubSections

Subsections are a way to divide the debug information into logical chunks. They don't carry much information in themselves, but act as headers for the specific subsection types (see below).

Each subsection has an associated sequential module index, starting at 1. A module section, source module section and symbol section, that have the same module index number, belong together logically: they all derive from the same unit. The Offset and Size fields are low-level and tell us where in the .EXE file the debug information section can be found.

We use the BorDebugSubSectionCount routine to get the number of sections in the .EXE file and BorDebugSubSection to read the fields for each section.

Subsection Types

Each TD32 subsection contains a different type of information. The BorDebug defines eight different constants with a BORDEBUG_SST prefix to designate the subsection types: see Table 1.

According to the documentation in BorDebug.h and Giant.Txt, the AlignSym section is reserved for symbols defined in the implementation section or in a local scope (ie, local variables). However, I have found that this is not the case. In practice, AlignSym is the same as GlobalSym and GlobalPub. In fact, during my testing I have only encountered AlignSym sections. You have to examine each symbol inside the section to determine if it has a global or local scope. We will refer to all these three section types as Symbol sections.

This simplifies things a little, as we are only left with six different subsection types to worry about. Throw in the fact that the Browse section is not currently supported by BorDebug and we are left with five: Modules, SrcModules, Symbols, Types and Names. The first three are all associated with a specific module (ie a DCU unit). The last two contain information on all global types and all name strings. Typically, we obtain indices obtained from the first three sections (modules, source modules and symbols) and lookup the name string or type information directly from the index.

➤ *Table 1*

BorDebug_sstxxx	Explanation
MODULE	Code and data from one OBJ or DCU file
ALIGNSYM	Symbols belonging to a local block
SRCMODULE	Code from a single source file (.PAS unit)
GLOBALSYM	Globally available symbols (variables, routines, etc)
GLOBALPUB	Same as GlobalSym
GLOBALTYPES	Globally available type definitions
NAMES	Mapping from name- and type-indices to name strings
BROWSE	Browsing information (declaration and usage points)

Modules

In BorDebug-speak, a Module is an entity that contributes code and/or data to the final executable. Examples of this are DCU and OBJ files. Even if you don't have the source code for a DCU or OBJ file, the public information needed for linking is added to a Module section in the final debug information. Each module has a name index and we can use this to look up the name of the DCU file. The module gets its module index from the subsection header. It also has a segment count, and we can retrieve information about each module segment.

A module can contribute code or data to one or more segments. Typically, each module always states that it contributes to three segments: the code segment, the initialized data segment (for typed constants and initialized global variables) and the uninitialized data segment (for other global variables). For some segments, the size can be 0 (if there are no initialized global variables in a unit, for instance).

When the subsection type is BORDEBUG_SSTMODULE, you can read the module information with the BorDebugModule function.

Segments

Each module typically contributes to three segments. Each segment maintains information about the corresponding logical segment index used by the linker (typically 0 for code, 1 for initialized data and 2 for uninitialized data), the offset in the segment, the size and code or data flags. We can use the BorDebugModuleSegment API to read this information.

Source Modules

A source module corresponds to the concept of a Pascal source unit with all its include files and any assembly files. The source module inherits its module index from the corresponding section header. This number can be used to associate the source module with the DCU file it generated (ie, the `Module` section) and the symbols it contains (in the `Symbol` section). Further, each source module contains a number of ranges and one or more source files (more about this below).

When the subsection type is `BORDEBUG_SSTSRCMODULE`, we can read the basic source module information using the `BorDebugSrcModule` function.

Source Module Ranges

A source module will contain one or more ranges. Each of these ranges is a combination of a segment index, a starting offset and an ending offset. This encodes information about where the source module contributes code and data. The source module ranges are read in using the `BorDebugSrcModuleRanges` routine.

Source Files

Each source module will also contain one or more source file entries (one for the main .Pas file and one for each include or assembly file used). Each source file entry has a sequential index, a name and a number of source file ranges associated with it (more of below). The `BorDebugSrcModuleSources` function reads in the information for a single source file entry.

Source File Ranges

As mentioned, each source file entry maintains information about the ranges where they contribute code or data (a segment index, plus starting and ending offsets). These are read using `BorDebugSrcModuleSourceRanges`. Finally, each source file range also contains mapping of line number and address information, see below. Normally, each source file has only a single range defined for it (during my testing, System.Pas had three ranges and SysUtils.pas had two, all other units had one range).

Line Numbers

The mapping of addresses to unit line numbers and vice versa is a very important part of the debug information. Each source file range can have an array of line number and code-offset mappings. These are read using the `BorDebugSrcModuleLineNumbers` routine.

Symbols

There are three subsection types that logically map to the same information: `AlignSym`, `GlobalSym` and `GlobalPub`. We just refer to all of these as `Symbol` subsections. Just as we have seen for `Module` and `Source Module` sections, a `Symbol` section inherits its module index from the corresponding section header. This allows us to easily map the symbol information to the module (.DCU file) and source module (.Pas file) it belongs to.

Because there are typically a large number of symbols associated with a module, there is no direct, index-based, method of reading them. Instead, we must iterate over all of them, one by one. Each symbol contains a fixed common header. This includes a symbol kind, offset and length (in the debug information). Then there is information specific to each kind of symbol. This often includes the name and a type index (more on this below), what segment (code or data) it belongs to, the offset where the information begins (inside the code or data segment), and so on.

When the subsection type is `BORDEBUG_SSTALIGNSYM`, `BORDEBUG_SSTGLOBALSYM` or `BORDEBUG_SSTGLOBALPUB`, we can start to iterate over the symbols using the `BorDebugStartSymbols` routine. Then we read the common symbol header using the `BorDebugNextSymbol` function. Finally, we check the symbol kind field and call the corresponding `BorDebugSymbolXXX` routine to get the specific information.

Types

As mentioned above, many symbols will have an associated type index. This represents the type of the symbol (duh!). More interestingly, we can use this type index to look up more detailed, logical information about the type. It is also possible to simply get a string representation of the type.

All types have a fixed, common header (sounds familiar?). This header includes the type index, a type kind, an offset and a length (in the debug info). Information specific to each type usually contains the name, but is otherwise very diverse. For instance, an array type contains information about the type of the elements and index (ie, a recursive type index), the name of the type, the total size of the array and the number of elements in the array.

With a type index in hand, we can get a string representation from the `BorDebugTypeIndexToString` procedure. If we are interested in the finer details, we first read the common type header using the `BorDebugTypeFromIndex` routine. Then we check the type kind field and call the corresponding `BorDebugTypeXXX` routine to get the specific information. If this contains further type indices, we continue this process recursively.

Names

When we retrieve information about an item that has a name (a module, source file, symbol, type, etc), we always get just a numeric name index. This name index isn't very useful by itself, so we can use it to look up the actual name string in the names section.

In the debug information, the names section is just a large block of length byte prefixed and zero-terminated strings. There is no explicit name index encoded into the .EXE file, so this must be computed at runtime.

The `BorDebugRegisterFile` routine can do this for you and it takes two `Boolean` parameters to decide how it should deal with this subsection. If the `SkipNames` parameter is `True`, `BorDebug` ignores the names section altogether and you cannot call any of the name-related APIs below. If `SkipNames` is `False`, the parameter `CacheNames` acts as a 'quick but memory consuming' versus a 'slow but memory lean' toggle. Normally you will set these parameters to `False` and `True`, respectively.

The following `BorDebug` routines can be used to read the name section: `BorDebugNamesTotalNames` returns the total number of name strings. `BorDebugNameIndexToName` returns the string associated with a specific name index. Alternatively, we can call `BorDebugNameIndexToUnmangledName` to get an unmangled version of the name. Name mangling is the process of encoding parameter and calling convention information in the linker name of a routine. This process can make the information nearly unreadable. Finally `BorDebugRegIndexToName` converts a register index into the string representation of the register (eg, `EAX`, `EBX` etc).

Browse information

The TD32 information can optionally contain information to help browsing tools. The `Browse` section encodes information such as where a symbol or type was defined and where it was used. The current version of the `BorDebug` API does not have any direct support for reading this information. There is an undocumented routine called `BorDebugDumpBrowserInfo`, but it seems like it is not implemented. If you want to read and interpret this information, you would have to start with the offset and size information for the `BORDEBUG_SSTBROWSE` sub-section. After this you are basically on your own.

Information from the Giant.Txt file (dated March 1999) even indicates that browser information is not included in the TD32 info at all: 'Note: There are several fields in the symbol structures which refer to browser information. These fields are set to 0 by the current Win32 compiler and linker. Currently only the OS/2 version of BC++ supports this browser information. Therefore it is not documented here.'

The Raw BorDebug API

OK, with all that background information under our belts, we are ready to dive into the nitty-gritty details of how to use BorDebug. The DLL exports a number of API routines, listed out for you in Listing 1. Note that all the parameters and return types have been stripped from this listing to keep the size down: see the disk for the complete version.

I will not discuss each and every BorDebug routine available with all their parameters and their meaning. If I did, the article would probably fill this entire issue and I would only be repeating much of the information found in the BorDebug.h file. Instead, I will discuss the general structure of

➤ *Listing 1*

```
// General routines
function  BorDebugRegisterFile;
procedure BorDebugUnregisterFile;
function  BorDebugSubSectionDirOffset;
function  BorDebugSubSectionCount;
procedure BorDebugSubSection;
procedure BorDebugModule;
procedure BorDebugModuleSegment;
procedure BorDebugStartSymbols;
procedure BorDebugNextSymbol;
procedure BorDebugDumpBrowserInfo;
// SymbolXXXX routines:
procedure BorDebugSymbolCOMPILE;
procedure BorDebugSymbolREGISTER;
procedure BorDebugSymbolCONST;
procedure BorDebugSymbolUDT;
procedure BorDebugSymbolSSEARCH;
procedure BorDebugSymbolOBJNAME;
procedure BorDebugSymbolGPROCREF;
procedure BorDebugSymbolGDATAREF;
procedure BorDebugSymbolEDATA;
procedure BorDebugSymbolEPROC;
function  BorDebugSymbolUSES;
function  BorDebugSymbolNAMESPACE;
function  BorDebugSymbolUSING;
function  BorDebugSymbolPCONSTANT;
procedure BorDebugSymbolBPREL32;
procedure BorDebugSymbolLDATA32;
procedure BorDebugSymbolGDATA32;
procedure BorDebugSymbolPUB32;
procedure BorDebugSymbolLPROC32;
function  BorDebugSymbolGPROC32;
procedure BorDebugSymbolTHUNK32;
procedure BorDebugSymbolBLOCK32;
procedure BorDebugSymbolWITH32;
procedure BorDebugSymbolLABEL32;
procedure BorDebugSymbolENTRY32;
function  BorDebugSymbolOPTVAR32;
procedure BorDebugSymbolPROCRET32;
procedure BorDebugSymbolSAVREGS32;
function  BorDebugSymbolSLINK32;
// General routines 2:
procedure BorDebugSrcModule;
procedure BorDebugSrcModuleRanges;
procedure BorDebugSrcModuleSources;
procedure BorDebugSrcModuleSourceRanges;
procedure BorDebugSrcModuleLineNumbers;
procedure BorDebugGlobalSym;
procedure BorDebugGlobalTypes;
procedure BorDebugTypeFromIndex;
procedure BorDebugTypeFromOffset;
// TypeXXX routines:
procedure BorDebugTypeMODIFIER;
procedure BorDebugTypePOINTER;
procedure BorDebugTypeARRAY;
procedure BorDebugTypeCLASS;
procedure BorDebugTypeUNION;
procedure BorDebugTypeENUM;
procedure BorDebugTypePROCEDURE;
procedure BorDebugTypeMFUNCTION;
function  BorDebugTypeVTSHAPE;
function  BorDebugTypeLABEL;
procedure BorDebugTypeSET;
procedure BorDebugTypeSUBRANGE;
procedure BorDebugTypePARRAY;
procedure BorDebugTypePSTRING;
procedure BorDebugTypeCLOSURE;
procedure BorDebugTypePROPERTY;
function  BorDebugTypeLSTRING;
function  BorDebugTypeVARIANT;
procedure BorDebugTypeCLASSREF;
function  BorDebugTypeWSTRING;
function  BorDebugTypeARGLIST;
procedure BorDebugTypeStartFIELDLIST;
procedure BorDebugTypeNextFIELDLIST;
function  BorDebugTypeDERIVED;
procedure BorDebugTypeBITFIELD;
function  BorDebugTypeMETHODLIST;
procedure BorDebugTypeBCLASS;
procedure BorDebugTypeVBCLASS;
procedure BorDebugTypeIVBCLASS;
procedure BorDebugTypeENUMERATE;
procedure BorDebugTypeFRIENDFCN;
function  BorDebugTypeINDEX;
procedure BorDebugTypeMEMBER;
procedure BorDebugTypeSTMEMBER;
procedure BorDebugTypeMETHOD;
procedure BorDebugTypeNESTTYPE;
procedure BorDebugTypeVFUNCTAB;
function  BorDebugTypeFRIENDCLS;
function  BorDebugTypeCHAR;
function  BorDebugTypeSHORT;
function  BorDebugTypeUSHORT;
function  BorDebugTypeLONG;
function  BorDebugTypeULONG;
function  BorDebugTypeREAL32;
function  BorDebugTypeREAL64;
function  BorDebugTypeREAL80;
function  BorDebugTypeQUADWORD;
function  BorDebugTypeUQUADWORD;
function  BorDebugTypeREAL48;
// Name and type string routines
function  BorDebugNamesTotalNames;
procedure BorDebugNameIndexToUnmangledName;
procedure BorDebugNameIndexToName;
procedure BorDebugRegIndexToName;
procedure BorDebugTypeIndexToString;
function  BorDebugUnmangle;
```

the information, what it is, and how to typically use it.

The routines can be grouped broadly into four categories. There are a number of general routines to open and close the debug information, iterate through the contents and so on. Then there are the BorDebugSymbolXXX routines that decode each specific kind of symbol you might encounter. The BorDebugTypeXXX decodes the different kind of type information available in the debug information. And finally you have a group of routines used to iterate through and look up string representations of symbol and type indices.

In addition to the constants and routine imports translated from the C header file, I've also added a number of types and constants to make the interface more self-describing and to improve type safety. The BorDebug unit also defines some extra helper routines when BORDEB_EXTRAS is defined (which it is by default). BorDebugRegisterFileEx wraps the native BorDebugRegisterFile call and raises an exception if an error occurs. It also optionally tries to open a .TDS file if the debug info cannot be found in the executable itself. Three CrackXXX routines help you interpret packed C bit fields.

There is also a conditional define called DYNLINK_BORDEBUG. This is off by default, which means that BorDebug.DLL will be implicitly linked to in the usual manner. If the OS cannot find the DLL (in the application directory or in the path), the program will not load. By enabling this define, the routines are linked to dynamically at runtime (by calling LoadLibarary and GetProcAddress). This means that your application will still load even if the DLL cannot be found on the system. This may be useful if it does not totally depend on reading debug information (it could have other functionality as well). The magic that makes this possible is defined in the HVDll unit, presented in Issue 43[14].

Start Digging

Let's get our hands dirty and write a little example program that uses the BorDebug API. The small program in Listing 2 takes the filename given on the command line and dumps all the debug symbols to standard output.

To keep the size of this sample down, it just aborts if an error occurs. The BorDebug API isn't very forgiving if you give it invalid handles or memory pointers. Access violations can occur easily if you are not careful.

```
program BDDmpNam;
{$APPTYPE CONSOLE}
uses
  BorDebug;
var
  Handle: TBorDebHandle;
  NameBuf: array[0..1024] of char;
  NameCount : integer;
  i: integer;
  BorDebError: TBorDebError;
begin
  Handle := BorDebugRegisterFile(PChar(ParamStr(1)), false, true, BorDebError);
  if BorDebError <> deOk then
    Halt;
  NameCount := BorDebugNamesTotalNames(Handle);
  writeln('Total names: ', NameCount);
  for i := 1 to NameCount do begin  // Note: Name index is 1-based!
    BorDebugNameIndexToName(Handle, i, NameBuf, SizeOf(NameBuf));
    Writeln(i, ' = ', NameBuf);
  end;
  BorDebugUnregisterFile(Handle);
end.
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

```
Total names: 10820
1 = System
2 = SysInit
3 = Windows
4 = BorDebug
5 = BDDmpNam
6 = @System@CloseHandle
7 = @System@CreateFileA
...
10815 = @Bddmpnam@TCrackedClassMemberAttrib
10816 = ClassMemberProtection
10817 = ClassMemberProperty
10818 = TClassMemberAttribs
10819 = TVtabOffset
10820 = TVtabOffsets
```

We first call the BorDebugRegisterFile function to open the file given on the command line. If it succeeds in locating the TD32 info, it returns a valid handle that we use in all subsequent BorDebug calls.

Then we get the total number of names in the global symbol name section using the BorDebugNamesTotalNames routine. The indices used by the name routines are 1-based, so we loop from 1 to NameCount, getting each name string using the BorDebugNameIndexToName routine, writing each name to standard output. Finally, we close the BorDebug handle by calling BorDebugUnregisterFile.

That wasn't too hard, was it? You can see parts of the (very long) output produced by this program in Listing 3.

After opening the debug info with BorDebugRegisterFile, you can get the number of available sub-sections by calling BorDebugSubSectionCount. Then you can iterate through these subsections, calling BorDebugSubSection for each index. The retrieved subsections each have a subsection type, offset and size. The subsection types will be one of the eight listed in Table 1.

We can enhance our simple console application to perform these steps, writing information about all the subsections available, and some extra information about the modules and the source modules and counting the number of symbols in each symbol section, see the BDSecInf program in Listing 4.

This is starting to get a little more involved. Let's go through the code.

A module subsection (BORDEBUG_SSTMODULE) doesn't have much extra information associated with it, so we just retrieve its name index (with BorDebugModule) and look up the string for that index (with BorDebugNameIndexToName). We dump out this name and the segment count.

```
program BDSecInf;
{$APPTYPE CONSOLE}
uses
  BorDebug, SysUtils, HVBorDebug;
var
  Handle         : TBorDebHandle;
  BorDebError    : TBorDebError;
  SubSectionCount: integer;
  SubSectionIndex: integer;
  SubsectionType : TSubsectionType;
  Module         : TModuleIndex;
  Offset         : TFileOffset;
  Size           : TByteCount;
  Overlay        : TOverlayIndex;
  LibIndex       : TLibraryIndex;
  Style          : TDebuggingStyle;
  NameIndex      : TNameIndex;
  TimeStamp      : TBDTimeStamp;
  SegmentCount   : TItemCount;
  Name           : array[0..260] of char;
  RangeCount     : TItemCount;
  SourceCount    : TItemCount;
  SourceOffsets  : PFileOffsets;
  NameIndices    : PNameIndices;
  RangeCounts    : PItemCounts;
  i              : integer;
  SymbolKind     : TSymbolKind;
  SymbolOffset   : TFileOffset;
  SymbolLen      : TByteCount;
  SymbolCount    : integer;
begin
  Handle := BorDebugRegisterFile(PChar(ParamStr(1)),
    false, true, BorDebError);
  if BorDebError <> deOk then
    Halt;
  SubSectionCount := BorDebugSubSectionCount(Handle);
  Writeln('Total subsections: ', SubSectionCount);
  // Note: SubSection index is 0-based!
  for SubSectionIndex := 0 to SubSectionCount-1 do begin
    BorDebugSubSection(Handle, SubSectionIndex,
      SubsectionType, Module, Offset, Size);
    Writeln(Format('SUBSECTION #%d: %s, Module=%d,
      Offset=%.8x, Size=%d', [SubSectionIndex,
      SubsectionTypeToString(SubsectionType), Module,
      Offset, Size]));
    case SubsectionType of
      BORDEBUG_SSTMODULE:
        begin
          BorDebugModule(Handle, Offset, Overlay, LibIndex,
            Style, NameIndex, TimeStamp, SegmentCount);
          BorDebugNameIndexToName(Handle, NameIndex, @Name,
            SizeOf(Name));
          Writeln(Format('  %s.DCU, SegmentCount=%d',
           [Name, SegmentCount]));
        end;
      BORDEBUG_SSTSRCMODULE:
        begin
          BorDebugSrcModule(Handle, Offset, RangeCount,
            SourceCount);
          // We'll ignore the ranges for now, lets get info
          // about the source files. Allocate enough memory
          // for each source file array
          GetMem(SourceOffsets, SourceCount *
            SizeOf(SourceOffsets^[0]));
          GetMem(NameIndices  , SourceCount *
            SizeOf(NameIndices  ^[0]));
          GetMem(RangeCounts  , SourceCount *
            SizeOf(RangeCounts  ^[0]));
          // Now get the source file segment offsets, name
          // indices and range counts
          BorDebugSrcModuleSources(Handle, Offset,
            SourceOffsets, NameIndices, RangeCounts);
          // We don't care about the offsets and ranges
          // right now, so just free them
          FreeMem(SourceOffsets);
          FreeMem(RangeCounts  );
          // Write the names of the files that contribute to
          // this unit
          for i := 0 to SourceCount-1 do begin
            BorDebugNameIndexToName(Handle, NameIndices^[i],
              @Name, SizeOf(Name));
            Write('  ', Name, ',');
          end;
          Writeln;
          FreeMem(NameIndices);
        end;
      BORDEBUG_SSTGLOBALSYM, BORDEBUG_SSTGLOBALPUB,
        BORDEBUG_SSTALIGNSYM:
        begin
          SymbolCount := 0;
          BorDebugStartSymbols(Handle, SubSectionType,
            Offset, Size);
          while true do begin
            BorDebugNextSymbol(Handle, SymbolKind,
              SymbolOffset, SymbolLen);
            if (SymbolKind = 0) and (SymbolOffset = 0)
              and (SymbolLen = 0) then
              Break;
            Inc(SymbolCount);
          end;
          Writeln('  ', SymbolCount, ' symbols');
        end;
    end;
  end;
  BorDebugUnregisterFile(Handle);
end.
```

➤ *Listing 4*

A source module (BORDEBUG_SSTSRCMODULE) has much more information available. For our purposes, we are just interested in the names of the source files that make up the source module. We first get the number of source files and ranges using the BorDebugSrcModule call. Ignoring the RangeCount for now, we then dynamically allocate three arrays to hold the offset, name index and range count for each source file. Then we get BorDebug to fill in information into our arrays by calling the BorDebug-SrcModuleSources API. At this point we don't really care about the offsets and ranges, so we just free them again. Now that we have an array of the name indices for the source files, we can just loop and write each name out (again calling BorDebugNameIndexToName to convert each name index into a string). We must remember to free the array of name indices.

Finally, for the symbol sections, we just iterate over all the symbols using the BorDebugStartSymbols and BorDebugNextSymbol routines. This gives us the kind of symbol, and the offset and size of the symbol inside the debug information. Notice that the end of the symbol section is detected by receiving the special value 0 for all three parameters. To keep the size of the output down, we don't write anything for each symbol, but just update a global counter and write the total count for each symbol section.

Phew! As you can see we have to write a fair amount of code just to do some simple dumping of basic information. This API just screams to be wrapped in a set of classes. And we *will* do that. However, I think it is better to first use the raw API in order to learn properly how it works and how we can best design our classes later.

Let's see if we can gain some more understanding by studying the output from this program. Listing 5 is an abbreviated version of what it produces when it's run on itself.

We can see that all the module sub-sections come first. Each sub-section one has a sequential module index, starting with 1. A module index of 0 means that the sub-section is not associated with a specific module, this applies to the Global Types and the Names sub-sections at the bottom of the listing.

Furthermore, the offsets of each sub-section refer to the offset from the beginning of the .EXE file, where the debug information for that sub-section can be found. Likewise, the sub-section size is the size of the debug information. Note that all Module subsections have a segment count of 3. We'll come back to this.

The program was compiled with the debug version of the RTL units. So the first source module is the one for System.Pas. Note that the module index for this sub-section is the same as the module index for the System.DCU module sub-section. This tells us that the source files listed here produced the System.DCU unit. We can also see that the `System` unit actually consists of one unit source file (system.pas), one include file (getmem.inc) and five assembly files (assign.asm, close.asm, opentext.asm, writestr.asm and _ll.asm). There is no mention of the intermediate .OBJ files produced by the assembly files: they are all sucked into the System.DCU file and no trace of them can be seen in the debug information.

Next we find an `AlignSym` sub-section. This also has a module index of 1, so it belongs to the System.DCU module. The `AlignSym` sub-section contains all the symbols linked in from the `System` unit. In this case we see that we have 1,268 symbols. This number will vary according to how much of each unit we use. The smart linker will remove symbols that are not referenced and these will not show up in the debug information either. We will look at the more detailed information inside the symbol sections a little later.

The next source module is for the SysInit.DCU module (index 2) and consists of one file, SysInit.pas. Then follow the symbols for `SysInit`. Then there is another `AlignSym` sub-section, for module index 3, which is the Windows.DCU module. Notice the `Windows` unit doesn't have any source module defined for it, this is because there is no debug version of Windows.DCU in the delphi5\lib\debug directory. Only units compiled with the debug information (`$D+`) setting checked produce source module sub-sections in the EXE's debug info.

Also note that the browse section type is absent, even if the units were compiled with reference information included. The last interesting bit of information gleaned from this output is that the main project file is the only one that has a full path registered. Presumably, this can help a debugger locate the project files.

Class: Can I Have Your Attention, Please?

By now you are probably already dizzy with all the funny concepts and strange inter-dependencies: I know I was when I was writing this article! Writing more involved demo programs, or even full-scale applications, using the raw `BorDebug` API could quickly become very tedious indeed. Not to worry, I have some Delphi classes which will come to the rescue. Listing 6 shows the public sections of the classes in the `HVBorDebug` unit.

There are a number of classes in this unit, but normally you will only create the top-level class: `TBorDebug`. Then you assign the `Filename` property to the file containing the debug information and set `Active` to `True`. Now you can easily iterate or look up a name index using the `Names` array property. This class also lets you easily loop through all subsections and conditionally create modules, source modules, symbol info and type info classes. It will let you iterate though the symbols within a subsection.

The other classes that support this main class are `TBorDebugModule`, `TBorDebugSrcModule`, `TModuleSegment`, `TSourceFileEntry`, `TLineNumberOffsets`, `TSymbolInfo` and `TTypeInfo`. These classes correspond to the concepts we discussed above and they shield us from having to call the `BorDebug` API routines directly.

Records Versus Classes

During the design of the `TBorDebug` classes, several times I had to consider when to use classes and when to use plain old records. Classes are a higher-level construct that allows you to add intelligence and collaboration, the drawback is that you have to explicitly create and free them. Records can live in the stack and need no lifetime management, but are dumber. Using old-style objects might have been a good middle way, but they are not officially supported, and I don't recommend using them.

I ended up using classes for the higher-level concepts that own other items and thus simplify tasks by making them somewhat intelligent. When the burden of continually creating and destroying the items overweighed the potential benefit of using classes, I stuck to plain records.

You will also see in the `HVBorDebug` unit on disk, that I have added a large number of record types. These are used to hold the output parameters of the `BorDebugSymbolXXX` and `BorDebugTypeXXX` routines. It is simpler to have them as records, because I

➤ *Listing 5*

```
Total subsections: 38
SUBSECTION #0: MODULE, Module=1, Offset=00010828, Size=64
  System.DCU, SegmentCount=3
SUBSECTION #1: MODULE, Module=2, Offset=00010868, Size=64
  SysInit.DCU, SegmentCount=3
SUBSECTION #2: MODULE, Module=3, Offset=000108A8, Size=64
  Windows.DCU, SegmentCount=3
(...more...)
SUBSECTION #11: MODULE, Module=12, Offset=00010AE8, Size=64
  HVBorDebug.DCU, SegmentCount=3
SUBSECTION #12: MODULE, Module=13, Offset=00010B28, Size=64
  BDSecInf.DCU, SegmentCount=3
SUBSECTION #13: SRCMODULE, Module=1, Offset=00010B68, Size=23426
  system.pas,  GETMEM.INC,  assign.asm,  close.asm,  opentext.asm,
   writestr.asm, _ll.asm,
SUBSECTION #14: ALIGNSYM, Module=1, Offset=000166EA, Size=28644
  1268 symbols
SUBSECTION #15: SRCMODULE, Module=2, Offset=0001D6CE, Size=388
  SysInit.pas,
SUBSECTION #16: ALIGNSYM, Module=2, Offset=0001D852, Size=1038
  44 symbols
SUBSECTION #17: ALIGNSYM, Module=3, Offset=0001DC60, Size=162350
  7707 symbols
(...more...)
SUBSECTION #32: SRCMODULE, Module=12, Offset=0005CAC6, Size=130
  HVBorDebug.pas,
SUBSECTION #33: ALIGNSYM, Module=12, Offset=0005CB48, Size=3888
  207 symbols
SUBSECTION #34: SRCMODULE, Module=13, Offset=0005DA78, Size=262
  D:\DelMag\BorDebug\Demos\BDSecInf.dpr,
SUBSECTION #35: ALIGNSYM, Module=13, Offset=0005DB7E, Size=966
  39 symbols
SUBSECTION #36: GLOBALTYPES, Module=0, Offset=0005DF44, Size=327976
SUBSECTION #37: NAMES, Module=0, Offset=000AE06C, Size=373882
```

also use them in large variant records to have single `TSymbolInfo` and `TTypeInfo` classes which can store the representation of any symbol or type, respectively.

Simplified Scanning

Even with this level of support, I found that I still had to write a large amount of boiler-plate code to iterate through the subsections, calling the correct methods to create secondary classes and so on. To help with this, I wrote another class called `TCustomBorDebugScanner`. This class is defined in the `BorDebugScanners` unit, see Listing 7.

As you can see, this class has a very simple public interface. You firstly create it, giving it a reference to the BorDebug instance that should be used when scanning. Then you call the `Scan` method, indicating what subsections and information you are interested in scanning. The `TCustomBorDebugScanner` class doesn't do anything interesting by itself, but it defines a number of virtual template methods in the protected section that descendant classes are supposed to override.

By inheriting from this class, it is fairly easy to write code to convert an address to the corresponding unit name and line number. The `TLineNumberScanner` in the same unit does this by overriding the `ScanLineNumberOffset` method and

➤ *Listing 6*

```
unit HVBorDebug;
{   Simplified class interface for the BorDebug API
    Written by Hallvard Vassbotn (hallvard.vassbotn@c2i.net)
    April 1999 - September 2000 }
interface
uses
  Windows, Classes, TypInfo, BorDebug, SysUtils;
type
  // ... removed a lot of stuff -- see the code on disk
  TBorDebug = class(TObject)
  public
    constructor Create(const aFilename: string = '');
    destructor Destroy; override;
    procedure Open;
    procedure Close;
    property Handle: TBorDebHandle read GetHandle;
    property FileName: string;
    property SkipNames: boolean;
    property CacheNames: boolean;
    property Active: boolean;
    property NameCount: TItemCount;
    property Names[Index: TNameIndex]: string;
    property UnmangledNames[Index: TNameIndex]: string;
    property RegisterName[RegIndex: TRegNameIndex]: string;
    property SubSectionCount: TItemCount;
    property SubSections[Index: TSubSectionIndex]:
      TBorDebugSubSection;
    function CreateModule(const SubSection:
      TBorDebugSubSection): TBorDebugModule;
    function CreateSrcModule(const SubSection:
      TBorDebugSubSection ): TBorDebugSrcModule;
    procedure StartSymbols(const SubSection:
      TBorDebugSubSection);
    function GetNextSymbol(var Symbol: TBorDebugSymbol):
      boolean;
    function CreateSymbolInfo(const Symbol:
      TBorDebugSymbol): TSymbolInfo;
    function CreateTypeInfo(const aType: TBorDebugType):
      TTypeInfo;
    property TypeFromIndex[TypeIndex: TTypeIndex]:
      TBorDebugType;
    property TypeFromOffset[Offset: TFileOffset]:
      TBorDebugType;
    property TypeName[TypeIndex: TTypeIndex]: string;
    property GlobalSymbols[const SubSection:
      TBorDebugSubSection]: TBorDebugGlobalSymbol;
    property TypeCount: TItemCount;
    property TypesSignature: TSignature;
    property SubSectionDirectoryOffset: TFileOffset;
  end;
  TBorDebugModule = class(TBorDebugObject)
  public
    constructor Create(BorDebug: TBorDebug; Offset:
      TFileOffset);
    destructor Destroy; override;
    property Overlay            : TOverlayIndex  ;
    property LibIndex           : TLibraryIndex  ;
    property Style              : TDebuggingStyle;
    property TimeStamp          : TBDTimeStamp   ;
    property SegmentCount       : TItemCount     ;
    property NameIndex          : TNameIndex     ;
    property Name               : string         ;
    property ModuleSegmentList  : TList          ;
    property Segments[Index: integer]: TModuleSegment;
  end;
  TBorDebugSrcModule = class(TBorDebugObject)
  public
    constructor Create(BorDebug: TBorDebug; Offset:
      TFileOffset);
    destructor Destroy; override;
    property RangeCount         : TItemCount     ;
    property RangeSegments      : PSegmentIndices;
    property RangeSegmentStarts : PSegmentOffsets;
    property RangeSegmentEnds   : PSegmentOffsets;
    property SourceCount        : TItemCount     ;
    property SourceOffsets      : PFileOffsets   ;
    property NameIndices        : PNameIndices   ;
    property RangeCounts        : PItemCounts    ;
    property SourceFileList     : TList          ;
    property SourceFiles[Index: integer]: TSourceFileEntry;
    property SourceNames[Index: integer]: string;
  end;
  TModuleSegment = class(TObject)
  public
    constructor Create(Module: TBorDebugModule;
      SegmentIndex: TSegmentIndex);
    property LinkerSegment     : TLinkerSegmentIndex;
    property Offset            : TFileOffset        ;
    property Size              : TByteCount         ;
    property Flags             : TSegmentFlags      ;
  end;
  TSourceFileEntry = class(TObject)
  public
    constructor Create(SrcModule: TBorDebugSrcModule;
      SourceFileIndex: TSourceFileIndex);
    destructor Destroy; override;
    property BorDebug           : TBorDebug       ;
    property Handle             : TBorDebHandle   ;
    property Offset             : TFileOffset     ;
    property SrcModule          : TBorDebugSrcModule;
    property Name               : string          ;
    property NameIndex          : TNameIndex      ;
    property SourceFileIndex    : TSourceFileIndex;
    property RangeSegments      : PSegmentIndices ;
    property RangeSegmentStarts : PSegmentOffsets ;
    property RangeSegmentEnds   : PSegmentOffsets ;
    property LineNumberCounts   : PItemCounts     ;
    property LineNumerOffsetList: TList           ;
    property RangeCount         : TItemCount      ;
    property RangeLineNumbers[Index: integer]:
      TLineNumberOffsets;
  end;
  TLineNumberOffsets = class(TObject)
  public
    constructor Create(SourceFile: TSourceFileEntry;
      RangeIndex: TRangeIndex);
    destructor Destroy; override;
    property SourceFile  : TSourceFileEntry;
    property LineNumbers : PLineNumbers    ;
    property LineOffsets : PSegmentOffsets ;
    property LineCount   : TItemCount      ;
    property RangeIndex  : TRangeIndex     ;
  end;
  TSymbolInfo = class(TObject)
  public
    constructor Create(BorDebug: TBorDebug; Symbol:
      TBorDebugSymbol);
    destructor Destroy; override;
    function GetTypeIndex(var TypeIndex: TTypeIndex):
      boolean;
    function GetNameIndex(var NameIndex: TNameIndex):
      boolean;
    property Symbol        : TBorDebugSymbol;
    property SymbolOffset  : TFileOffset    ;
    property Len           : TByteCount     ;
    property Kind          : TSymbolKind    ;
    property Info          : TSymbolInfoRec ;
    property KindAsString  : string         ;
  end;
  TTypeInfo = class(TObject)
  public
    constructor Create(BorDebug: TBorDebug; aType:
      TBorDebugType);
    destructor Destroy; override;
    property BDType        : TBorDebugType  ;
    property TypeIndex     : TTypeIndex     ;
    property TypeOffset    : TFileOffset    ;
    property Length        : TByteCount     ;
    property TypeKind      : TTypeKind      ;
    property Info          : TTypeInfoRec   ;
    property NameIndex     : TNameIndex     ;
    property KindAsString  : string         ;
  end;
```

```
unit BorDebugScanners;
interface
uses
  BorDebug, HVBorDebug;
type
  TScanningOption = (soModule, soAlignSym, soSrcModule,
    soGlobalSym, soGlobalPub, soGlobalTypes, soNames,
    soBrowse, soSrcModuleRanges, soSrcModuleFiles);
  TScanningOptions = set of TScanningOption;
  TCustomBorDebugScanner = class(TObject)
  protected
    function WantSymbol(...); virtual;
    function WantType(...); virtual;
    function WantFieldList(...); virtual;
    function WantTypeInfoForSymbol(...); virtual;
    procedure StartFieldListScan(...); virtual;
    procedure EndFieldListScan(...); virtual;
    procedure ScanLineNumberOffset(...); virtual;
    procedure ScanSrcModuleSourceRange(...); virtual;
    procedure ScanSymbolTypeInfo(...); virtual;
    procedure ScanSrcModule(...);  virtual;
    procedure ScanSrcModuleRange(...);  virtual;
    procedure ScanSrcModuleSource(...); virtual;
    procedure ScanSymbolInfo(...);  virtual;
    procedure ScanSymbols(...);  virtual;
    procedure ScanModule(...); virtual;
    procedure ScanModuleSegment(...); virtual;
    procedure ScanSubSection(...); virtual;
    procedure ScanSubsections; virtual;
    property CurrentSourceFileEntry: TSourceFileEntry;
    property CurrentLineNumberOffsets: TLineNumberOffsets;
    property CurrentSubSection: PBorDebugSubSection;
    property CurrentModule: TBorDebugModule;
    property CurrentSrcModule: TBorDebugSrcModule;
    property ScanningOptions: TScanningOptions;
  public
    constructor Create(ABorDebug: TBorDebug);
    procedure Scan(ScanningOptions: TScanningOptions);
    property BorDebug: TBorDebug;
  end;
```

➤ *Listing 7*

defining a new public method called `FindUnitnameLinenumber`, see Listing 8.

The `FindUnitnameLinenumber` function first saves the `Address` parameter in a field so that the `ScanLineNumberOffset` method can see the value. Then it calls the `Scan` method, indicating that it is interested in the source modules and the files they contain. This ensures that the `ScanLineNumberOffset` will be called for all the source files found in the debug information. Also, all other sub-sections will be ignored, speeding up the scanning process. This could probably be tuned further by ignoring source modules that have ranges which fall outside the address. If you need to look up a large number of addresses quickly, you probably want to load the information into your own custom data structures.

Inside the `ScanLineNumberOffset` method we check if the current address is lower or equal to the address we're looking for. If we don't find a perfect match, we select the closest match. When we find a match, we keep the supplied line number and unit name information (ie, the name of the current source file entry).

After the `Scan` call returns, we check if we found a match, and return those values. On the disk is a small demonstration application, AddrLookup.dpr, which shows how to use this class: you can see it running in Figure 4.

Note that the addresses used by this sample application are offsets relative to the start of the code segment. Actual runtime addresses will be different, so you will have to

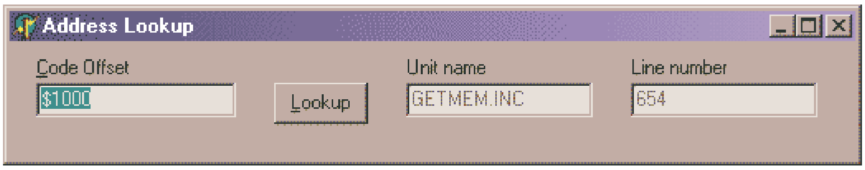
```
  TLineNumberScanner = class(TCustomBorDebugScanner)
  private
    FAddress: TSegmentOffset;
    FBestMatch: TSegmentOffset;
    FUnitname: string;
    FLinenumber: TLinenumber;
  protected
    procedure ScanLineNumberOffset(LineNumber: TLineNumber;
      LineOffset: TSegmentOffset); override;
  public
    function FindUnitnameLinenumber(
      Address: TSegmentOffset; out Unitname: string;
      out Linenumber: TLinenumber): boolean;
  end;
implementation
{ TLineNumberScanner }
procedure TLineNumberScanner.ScanLineNumberOffset(
  LineNumber: TLineNumber; LineOffset: TSegmentOffset);
begin
  if (LineOffset <= FAddress)
    and (LineOffset > FBestMatch) then begin
    FBestMatch := LineOffset;
    FLinenumber := Linenumber;
    FUnitName := CurrentSourceFileEntry.Name;
  end;
end;
function TLineNumberScanner.FindUnitnameLinenumber(
  Address: TSegmentOffset; out Unitname: string;
  out Linenumber: TLinenumber): boolean;
begin
  FAddress := Address;
  FBestMatch := 0;
  Scan([soSrcModule, soSrcModuleFiles]);
  Result := (FBestMatch > 0);
  if Result then begin
    Unitname := FUnitname;
    Linenumber := FLinenumber;
  end;
end;
```

➤ *Listing 8*

convert those into relative offsets by subtracting the base address of the executable (typically $0040000 for an .EXE file). In addition you normally have to subtract $1000 (added by the linker).

The Gigantic Dumper

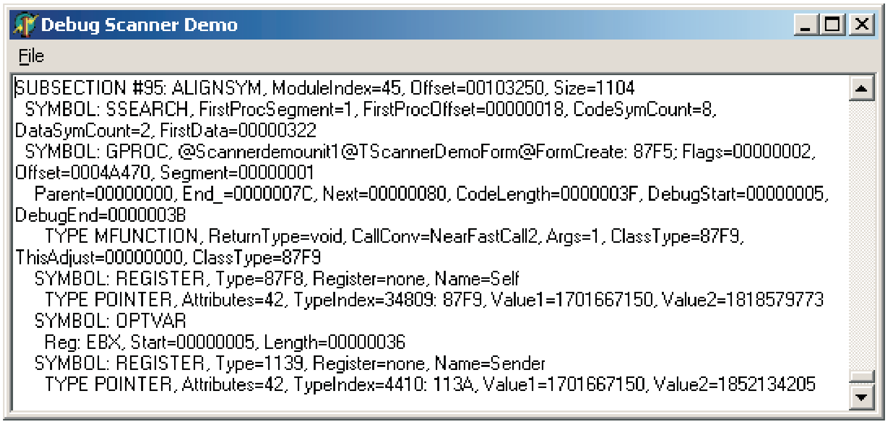
No article about the TD32 debug format would be complete without a generic dumper program. To write this, I first developed a re-usable TDumpBorDebugScanner class. This inherits from TCustomBorDebugScanner and overrides just about every virtual method to dump all the available debug information. The actual dumping is delegated to a public OnDump event. This way we can reuse all that boring dumping code (650 lines), and just plug in any media we like (write it to standard output, dump it to a TMemo control, or stream it).



Code Offset: $1000
Unit name: GETMEM.INC
Line number: 654

➤ *Above: Figure 4*

➤ *Below: Figure 5*



```
SUBSECTION #95: ALIGNSYM, ModuleIndex=45, Offset=00103250, Size=1104
 SYMBOL: SSEARCH, FirstProcSegment=1, FirstProcOffset=00000018, CodeSymCount=8,
DataSymCount=2, FirstData=00000322
 SYMBOL: GPROC, @Scannerdemounit1@TScannerDemoForm@FormCreate: 87F5; Flags=00000002,
Offset=0004A470, Segment=00000001
   Parent=00000000, End_=0000007C, Next=00000080, CodeLength=0000003F, DebugStart=00000005,
DebugEnd=0000003B
    TYPE MFUNCTION, ReturnType=void, CallConv=NearFastCall2, Args=1, ClassType=87F9,
ThisAdjust=00000000, ClassType=87F9
   SYMBOL: REGISTER, Type=87F8, Register=none, Name=Self
    TYPE POINTER, Attributes=42, TypeIndex=34809: 87F9, Value1=1701667150, Value2=1818579773
   SYMBOL: OPTVAR
    Reg: EBX, Start=00000005, Length=00000036
   SYMBOL: REGISTER, Type=1139, Register=none, Name=Sender
    TYPE POINTER, Attributes=42, TypeIndex=4410: 113A, Value1=1701667150, Value2=1852134205
```

I've created a little console program that uses this class to dump all the debug information of the file given on the command line to the standard output: see the code for BDDmpAll.Dpr in Listing 9.

Be warned that this program produces *large* outputs. When I ran it on itself, it produced a text file of about 3.8Mb! No doubt much of this information is redundant, but it does give you a hint of just how much information is in there. Just for the heck of it, I also wrote a GUI version: see Listing 10 and Figure 5.

Again, be aware that loading a file can take quite a few seconds. Notice the special trick I use to grow the Dump string. Without this, the program eats memory like crazy and takes forever to run. I'm using a vanilla TMemo in this sample, because it ran much slower with a TRichEdit control.

Conclusion

I could continue this article almost ad infinitum, but I have to put the

➤ *Listing 9*

```
program BDDmpAll;
{$APPTYPE CONSOLE}
uses
  BorDebug, HVBorDebug, BorDebugScanners,
  BorDebugDumpScanner;
type
  TDumpToStdOut = class(TObject)
  public
    procedure OnScannerDump(
      Sender: TObject; const Msg: string);
  end;
procedure TDumpToStdOut.OnScannerDump(
  Sender: TObject; const Msg: string);
begin
  Write(Msg);
end;
var
  Debug: TBorDebug;
  DumpScanner: TDumpBorDebugScanner;
  DumpToStdOut: TDumpToStdOut;
begin
  Debug := nil;
  DumpToStdOut := nil;
  DumpScanner := nil;
  try
    Debug := TBorDebug.Create(ParamStr(1));
    DumpToStdOut := TDumpToStdOut.Create;
    DumpScanner := TDumpBorDebugScanner.Create(Debug);
    DumpScanner.OnDump := DumpToStdOut.OnScannerDump;
    DumpScanner.Scan([soModule, soAlignSym,
      soSrcModule, soGlobalSym, soGlobalPub,
      soGlobalTypes, soNames, soBrowse,
      soSrcModuleRanges, soSrcModuleFiles]);
  finally
    DumpScanner.Free;
    DumpToStdOut.Free;
    Debug.Free;
  end;
end.
```

```
procedure TScannerDemoForm.FormCreate(Sender: TObject);
begin
  Debug := TBorDebug.Create;
  DumpScanner := TDumpBorDebugScanner.Create(Debug);
  DumpScanner.OnDump := OnScannerDump;
end;
procedure TScannerDemoForm.FormDestroy(Sender: TObject);
begin
  DumpScanner.Free;
  Debug.Free;
end;
procedure TScannerDemoForm.OnScannerDump(Sender: TObject;
  const Msg: string);
var
  NewLength : integer;
begin
  // Grow the dump buffer if it is too small
  if (Length(Dump) - DumpPos) < Length(Msg) then begin
    if Length(Msg) > Length(Dump) then
      NewLength := Length(Dump) + Length(Msg)
    else
      NewLength := Length(Dump) * 2;
    SetLength(Dump, NewLength);
  end;
  // Just move the content quickly
  // over to the dump buffer
  Move(Pointer(Msg)^, Pointer(@Dump[DumpPos+1])^,
    Length(Msg));
  // Update our position in the dump buffer
  Inc(DumpPos, Length(Msg));
end;
procedure TScannerDemoForm.ScanIt;
begin
  SetLength(Dump, 4*1024*1024); // 4Meg!
  DumpPos := 0;
  DumpScanner.Scan([soModule, soAlignSym,
    soSrcModule, soGlobalSym, soGlobalPub,
    soGlobalTypes, soNames, soBrowse,
    soSrcModuleRanges, soSrcModuleFiles]);
  SetLength(Dump, DumpPos);
  Memo.Lines.Text := Dump;
  Dump := '';
end;
procedure TScannerDemoForm.OpenItemClick(Sender: TObject);
begin
  if OpenDialog.Execute then begin
    Debug.Filename := OpenDialog.Filename;
    Debug.Open;
    ScanIt;
    Debug.Close;
  end;
end;
```

➤ *Listing 10*

limit somewhere. Now I have given you a Pascal wrapper around the BorDebug.DLL, a set of wrapper classes, a simplified scanner template class, a couple of descendants that do some real work, plus examples of how to use it all too.

Possible usability extensions would be to write a non-visual component that has events for all the virtual scanning methods. More interesting would be to find applications for the debug information itself.

One possible candidate is the 'old' Exceptional Stack Tracer code, presented back in Issue 50. Currently, Vitaly's RTLI code uses information from the .MAP file. It could be extended to take the information from the TD32 info, either at design-time to re-pack it, or directly at runtime. Another interesting project would be to write a custom Win32 symbolic debugger or runtime inspector. Or maybe take up the competition with QTime and Sleuth QA and write your own profiler? The world is now very much your oyster!

Hallvard Vassbotn is a Senior Systems Developer at Infront AS (visit www.theonlinetrader.com), where he develops systems for distributing real-time financial information over the internet. You can reach him at hallvard.vassbotn@c2i.net

References

1. Borland Turbo Debugger, a free download: www.borland.com/bcppbuilder/turbodebugger/
2. Numega BoundsChecker for Delphi: www.numega.com/products/aed/del.shtml
3. Turbo Power Sleuth QA Suite: www.turbopower.com/products/sleuthqa/
4. Atanas Stoyanov's MemProof home page: www.totalqa.com/downloads/ memproof.asp
5. AutomatedQA QTime: www.totalqa.com/index.asp
6. Intel VTune: http://developer.intel.com/vtune/
7. Wotsit's Format – The programmer's reference: www.wotsit.org/
8. Stefan Hoffmeister: www.econos.de/index.html
9. Hallvard Vassbotn, TDM Issue 50, October 1999, *Exceptional Stack Tracing*
10. John Thomas, Borland Debug Hook Library and Header File: http://ww6.borland.com/codecentral/ccweb.exe/listing?id=14513
11. Unfortunately, TDStrp32.EXE is not included with Delphi, nor the free BC++ compiler: www.borland.com/bcppbuilder/freecompiler/
12. Borland's Turbo Assembler 5.0: http://shop.borland.com/Product/ 0,1057,3-15-CQ100146,00.html
13. Hallvard Vassbotn, TDM Issue 38, November 1998, *Slimming The Fat Off Your Apps*
14. Hallvard Vassbotn, TDM Issue 43, March 1999, *DelayLoading of DLLs*