Exceptional Stack Tracing

Find and fix the causes of exceptions easily

by Hallvard Vassbotn

To paraphrase one of my old articles (*Ref 1*): have you ever had hard-to-find bugs in your code? If not, you can skip this article, otherwise, keep on reading!

Often, during beta testing of an application (and even sometimes in a release version!), users will encounter bugs in the form of exceptions (both logical and hardware). The tricky part is that only the address of where the exception occurred is reported by the default Delphi exception handler. Usually this is less than helpful, as typically the address maps to a line deep inside the VCL or RTL.

What we're really interested in is how we ended up there in the first place with invalid parameters (eg a blank string or a nil pointer). To get that information we would need a complete stack trace of the calls that ended up in the exception being raised.

This article is about developing such an exception stack tracer. Not only will it show a complete stack trace leading up to an exception, but in the presence of *Run Time Location Information* (RTLI), it will also give a complete symbolic stack trace. The RTLI code is taken from Vitaly Miryanov's article (*Ref 2*) in Issue 22. I have updated and modified it slightly to work better with any 32-bit version of the DCC32 command-line compiler.

By following the instructions in this article and including the relevant units in your projects, your beta testers and customers will be able simply to send you a file containing all the stack traces of any exceptions they have encountered. Given this file, it wil be much easier to track down and fix the causes of the exceptions. You'll no longer need to spend hours trying to recreate the problem.

Commercial Products

Before I go any further, I should point out that there are several commercial products that provide the same functionality as the code presented here. These products may be more professional, have more features and be better supported *[Hallvard is being very modest here. Ed]*. Specifically, I know of Per Larsen's ExHook32 (*Ref 3*) and Stefan Hoffmeister's Debug Mapper (*Ref 4*). You might like to check these out.

There's also a product called bugTrapper from MuTek Solutions (*Ref 5*). Currently, it only works with Microsoft's Visual C++ compiler, but the descriptions on their website sound very impressive.

Technical Hurdles

Before we start developing our exception stack tracer, we have to step back and see how we can design this thing. I can immediately see three main areas that we need to address.

First, hook the exception handling system so that we will know an exception has been raised.

Second, create a general stack trace utility that will return the code addresses currently pushed onto the stack.

Third, create a routine that will convert any code address to the corresponding symbolic unit name, line number and routine name.

The third task we already have covered by reusing Vitaly's RTLI code from Issue 22. The second should also be relatively easy. Way back in 1996, I had an article in Issue 7 where I presented a 16-bit stack tracer for Delphi 1.0 and BP7. We will see how this code can be converted to work in a 32-bit environment.

The hard part is the first task. How can we be notified when an exception is raised? One naïve approach would be to simply implement an `Application.OnException` handler. However, by the time this event is called, the stack has already been unwound and we have lost all the code addresses we're interested in on the stack. I have actually seen hints that this approach would work, by inspecting the separate exception-machinery stack on `FS:[0]`, but I have not found an easy way of doing this myself. Reportedly, Stefan Hoffmeister's Debug Mapper can use this approach.

Hooking Mania

There are basically two types of exceptions that can occur in an application. Firstly, there are the hardware generated exceptions, such as `EAccessViolation`. These are initially raised by the CPU when it sees that the application has attempted an illegal operation on a memory page, for instance. Then the OS does its thing and calls into the RTL exception hook. This will convert the OS-level exception into a Delphi exception object. If you have the RTL source code, you can see this in action by searching System.Pas and SysUtils.Pas for `ExceptObjProc`, `_HandleAnyException` and `GetExceptionObject`.

The key here is that `ExceptObjProc` is in effect a procedure variable that we can override to point to our own routine. Here we can run and save the stack trace, call the normal `ExceptObjProc` routine in `SysUtils` and then return. One down, one to go.

The other kind of exception is the type explicitly raised by the application using the `raise` keyword. These calls are spread throughout the application code, so it would be impractical to hook each and every call. Internally, the compiler generates a call to the magic routine `_RaiseExcept` in the `System` unit for these keywords. We could grab the address of this routine (by scanning the code generated by a dummy `raise` statement)

and then hook that address and point it to our own code. This would work, but I feel it is messy and there is a cleaner way.

I read an excellent article about PE files by Matt Pietrek (*Ref 6*). In it he describes how implicit linking to external DLLs work. About the import address table, he says: *'Since the import address table is in a writeable section, it's relatively easy to intercept calls that an EXE or DLL makes to another DLL. Simply patch the appropriate import address table entry to point at the desired interception function. There's no need to modify any code in either the caller or callee images. What could be easier?'*

Yeah, what could be easier! Such a statement just screams: 'Implement me!'. We will implement a completely general way of hooking any routine in any implicitly loaded DLL. We can then use this technique to hook the `Kernel32.RaiseException` routine that is called from `System._RaiseExcept`.

Now we will have the two hooks in place to get notified when both native Delphi exception and hardware exceptions are raised. At this point we will run a stack trace and save the result in a temporary buffer. We cannot report or log the exception just yet, because it might be silenced or converted to another exception by a `try..except` statement somewhere in the application. To avoid reporting these silenced exceptions, we will delay the actual reporting until the `Application.OnExecute` is called (yes, we will hook this too).

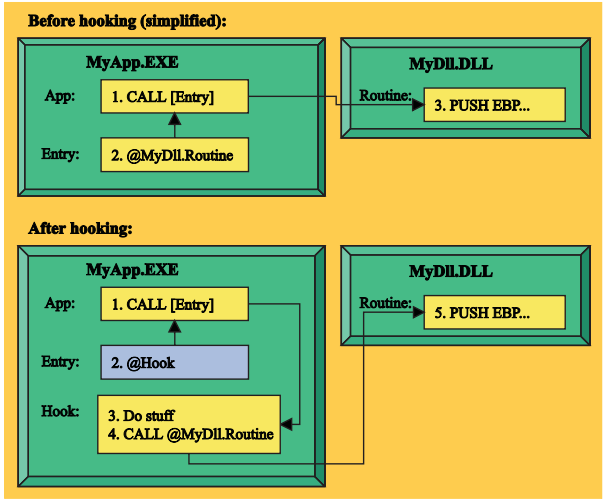
Generic DLL Hooking

As Pietrek pointed out, it should be relatively easy to hook any routine implicitly imported from any DLL. Part of the standard PE file format (which all Win32 executables follow, EXEs, DLLs, OCXs, etc) contains what is called an *Import Address Table* or IAT. After the OS has done its magic patching at load time, this table contains the addresses of all implicitly imported DLL routines. Implicitly imported means that they are imported without writing any special code. For instance, in Delphi, you can implicitly import a routine by using the `external` directive:

```
procedure MyDLLRoutine(
  A: integer);
  external 'MyDLL.DLL';
```

This is the way that most Win32 API routines are imported in the RunTime Library (RTL). You can see examples of this in Source\RTL\Win\Windows.Pas and in Source\RTL\Sys\System.Pas.

This implicit importing is in contrast to DLL routines you import explicitly using code that calls `LoadLibrary` and `GetProcAddress`. If you want to know more about implicit and explicit linking, see my previous article about dynamic DLL loading (*Ref 7*).



➤ *Figure 1: The basic mechanism of the DLL hooking.*

For the purposes of this article, we specifically want to hook the `RaiseException` routine in Kernel32.DLL. However, I want to develop a set of generic routines to do the hooking of any routine in any DLL, so that we might be able to reuse the code for other purposes later. Take a look at Figure 1 for an illustration of how the hooking mechanism should work.

I have simplified the illustration of how implicitly imported routines are called to reduce the clutter (for more details see *Ref 7*). The application might call the imported routine from many places in the code, but they all reference the single import entry in the PE import section. By simply overwriting the value in this entry with the address of our own routine, we have achieved the hooking functionality we're after.

The final hooking algorithm can be found in the `ReplaceImport` routine in the `HVDLLHook` unit. My first attempt at implementing it looked like Listing 1.

This function takes four parameters. `Base` is the loading address of the module which we are going to patch, `ModuleName` is the name of the DLL containing the routine we're going to hook, and the last

➤ *Listing 1: Simplified routine to generically hook DLL routines.*

```
function ReplaceImport(Base: Pointer; ModuleName: PChar;
  FromProc, ToProc: pointer): boolean;
var
  NtHeader          : PImageNtHeaders;
  ImportDescriptor  : PImageImportDescriptor;
  ImportEntry       : PImageThunkData;
  CurrModuleName    : PChar;
begin
  Result := False;
  NtHeader := GetImageNtHeader(Base);
  ImportDescriptor := PImageImportDescriptor(DWORD(Base)+
    NtHeader.OptionalHeader.DataDirectory[
    IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress);
  while ImportDescriptor^.NameOffset <> 0 do begin
    CurrModuleName := PChar(Base) + ImportDescriptor^.NameOffset;
    if StrIComp(CurrModuleName, ModuleName) = 0 then begin
      ImportEntry := PImageThunkData(DWORD(Base) + ImportDescriptor^.IATOffset);
      while ImportEntry^.FunctionPtr <> nil do begin
        if (ImportEntry^.FunctionPtr = FromProc) then begin
          ImportEntry^.FunctionPtr := ToProc;
          Result := True;
        end;
        Inc(ImportEntry);
      end;
    end;
    Inc(ImportDescriptor);
  end;
end;
```

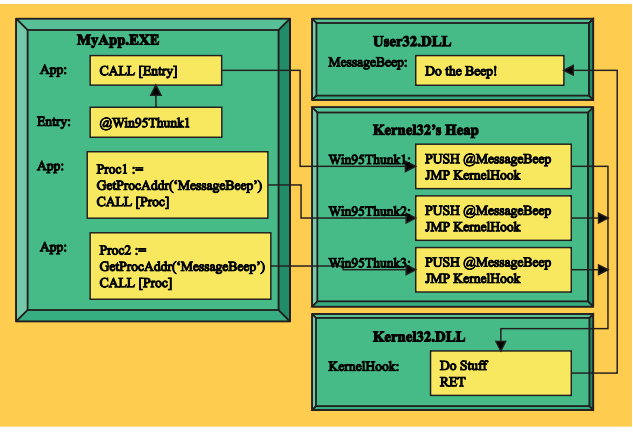
two, `FromProc` and `ToProc`, contain the address of the DLL routine and our hooking routine respectively. When we unhook the routine, the contents of the two last parameters will be reversed, of course.

First, `ReplaceImport` uses the `GetImageNtHeader` function from the `HVPEUtils` unit to get a pointer to the NT-specific header in the PE file, see Listing 2. Then it calculates a pointer to the `ImportDescriptor` table using some type definitions and simple calculations. As Dave Jewell pointed out (*Ref 8*), the `OptionalHeader` field is not really optional in Win32, so we'll just assume it is there.

All offsets referenced in the PE-header are relative to the loading address of the current module. That loading address conveniently equals the value of the `HInstance` variable. We send in this pointer as the `Base` parameter to make the `ReplaceImport` routine even more generic, it can in fact hook DLL routines called from another DLL, mind boggling isn't it?

Then, we loop through the `ImportDescriptor` structures stored in the PE-header. These contain information about each DLL module imported and we use the `NameOffset` field to calculate a pointer to the name of the module. The outer loop looks for module names that match the `ModuleName` parameter. The `ImportDescriptor` structures are stored in a contiguous block in the PE-file, so we can just increment the pointer at the end of the loop. The last descriptor is marked with a zero name offset.

➤ *Figure 2: Win95's thunking of some system DLL routines.*




```
function GetImageNtHeader(Base: Pointer): PImageNtHeaders;
var DOSHeader: PImageDOSHeader;
begin
  DOSHeader := PImageDOSHeader(Base);
  if DOSHeader.e_magic <> IMAGE_DOS_SIGNATURE then
    HookError('Not a valid MZ-file!');
  Result := PImageNtHeaders(DWORD(Base) + DOSHeader.e_lfanew);
  if Result.Signature <> IMAGE_NT_SIGNATURE then
    HookError('Not a valid PE-file!');
end;
```

➤ *Listing 2: Obtaining a pointer to the PE-header.*

If we find a matching module, we get a pointer to the first import entry structure for this imported module. There will be one import entry for every routine imported from the given module. After the OS has loaded and patched the entries, they will contain a pointer to the actual DLL routine. Again we loop through all the structures by incrementing the pointer at the end of the loop; the last entry is marked with a nil function pointer.

Inside the inner loop, we compare the function pointer stored in the entry to see if it matches the address of the routine we want to hook. If it does, we simply overwrite it with the address of our hook routine (the `ToProc` parameter). As Pietrek pointed out (*Ref 6*), all the import sections have both read and write access by default, so there is no need to play around with dirty `WriteProcessMemory` or `VirtualProtect` tricks. We also set the `Result` variable to `True` to show we succeeded with hooking the DLL routine. There could be more imports of the same routine, so we keep looking for more matches.

The Win95 Blues

The description given so far accurately describes how the initial version of this routine operated. My testing on a Windows NT machine showed that the code did work as intended. However, when I tested the code on a Windows 95 machine, it sometimes failed. I could still hook some imported routines, such as `RaiseException` from Kernel32, while trying to hook other routines (like `MessageBeep` from User32) would fail. How could this be?

It was time to bring up that old and trusted debugger again. The much-improved CPU View and Module View in Delphi 5 made it easier to track down the cause of the problem. It turns out that for some specific system DLLs and routines, Windows 95 does not give the actual address of the DLL routine directly on calls to `GetProcAddress` or in import entries in the PE-file. Instead, it gives the address of some dynamically generated code or thunk. This thunk is responsible for redirecting control to a routine that fiddles with the `DebugContext` structure and other thread-specific variables stored in the FS segment (*Ref 9*). Eventually it jumps to the actual DLL routine.

You can most easily see this by calling `GetProcAddress` twice on the same `User32.MessageBeep` routine. The two calls will return different addresses, but they will both take you to the correct routine. The reason for this is that Win95 creates two different thunks for the two calls, but the two thunks both redirect to the same routine. Figure 2 shows how this works.

I have documented the structure of the code in these thunks in the `TWin95CallThunk` record type, see Listing 3.

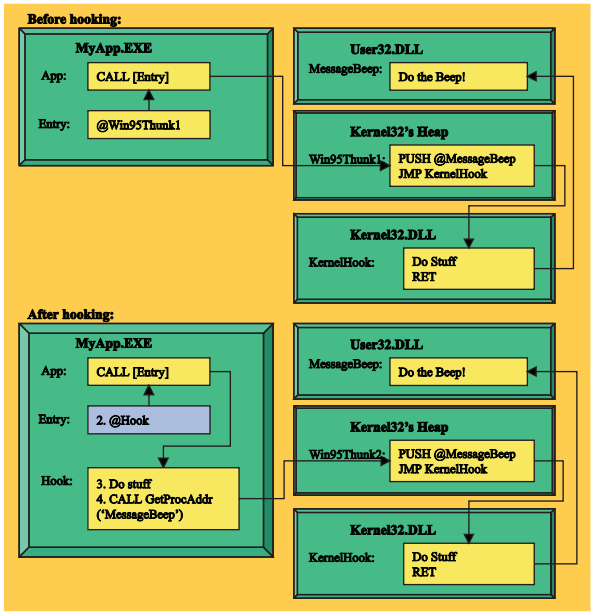
The thunk pushes the actual address of the DLL routine to the stack, then jumps to some internal code in Kernel32. I don't know the exact purpose of this code, but it checks some flags and tweaks the structures maintained on a per-thread basis and pointed to by the FS segment register (*Ref 9*). Eventually, the code returns to the address initially pushed onto the stack and ends up in the DLL routine proper. In effect, the Win95 kernel does its own hooking of these special system DLL routines.

What all this means is that we have to check if we're running in the Win95/98 version of Win32 and see if the import entry and `FromProc` parameter (obtained with a call to `GetProcAddress`) point to such thunks. If so, we must check if the `Addr` field of the records points to the same address and consider it a match. See Listing 4 for a revised version of `ReplaceImport` that implements this additional logic.

I have added a small utility function that checks if a potential thunk pointer actually points to a valid Win95 thunk. Inside `ReplaceImport`, we now first convert the `FromProc` address to a thunk pointer. Then we set the `IsThunked` flag if we're running on Windows 95 or 98 and the `FromProc` address references a valid thunk.

When this flag is set, we know that we should only consider import entries that are themselves thunk pointers. When the flag isn't set we know we should only check for clean matches of the import entry function pointer and the `FromProc` address. Checking and setting this flag outside the loop allows us to simplify the logic and optimise the code inside the loop.

The basic looping through the PE-file structures stays the same as before. If the `FromProc` we're trying to find was thunked, we first convert the `ImportEntry` into a potentially valid thunk pointer. If it is valid and if the `Addr` field of both thunks is equal, we consider it a match and set `FoundProc` to `True`.



➤ *Figure 3: Hooking in the presence of Win95 thunks.*

Otherwise, we don't care about the thunks and simply compare the import entry function pointer directly with the `FromProc` address, just as before. If either of these two approaches find a match, we patch the import entry with the address of the `ToProc` routine.

Note that although the code listings might look a little cryptic, the same code on the CD-ROM has plenty of comments! I have removed the comments in the printed listings to keep the size down.

If you have been watching closely, you will notice that we have now short-circuited the special Win95 installed system hook. When the application calls through the import entry, control will end up directly in our hook routine, without any detour into Kernel32 as depicted in Figure 2.

At first, I considered this a potential problem and wrote a version of `ReplaceImport` that patched the `Addr` fields of the thunks instead of the import entry slot. However, when I started to draw Figure 3, I realised this was not needed and maybe was worse than before. The reason is that, although the initial call from the application to the hooked routine will not go via a Kernel32 hook, the final call to the DLL routine *will* go through a thunk. This is because the address we get back from `GetProcAddress` points to a new, but functionally equal thunk. Take a look at Figure 3 to get a clearer picture of what happens.

➤ *Listing 3: Structure of the Win95 thunks.*

```
type
  PWin95CallThunk = ^TWin95CallThunk;
  TWin95CallThunk = packed record
    PUSH: byte;     // PUSH instruction opcode (=$68)
    Addr: pointer;  // The actual address of the DLL routine
    JMP : byte;     // JMP instruction opcode (=$E9)
    Rel : Integer;  // Relative displacement (a Kernel32 address)
  end;
```

➤ *Listing 4: DLL hooking that works in Win95.*

```
function IsWin95CallThunk(Thunk: PWin95CallThunk): boolean;
begin
  Result := (Thunk^.PUSH = $68) and (Thunk^.JMP = $E9);
end;
function ReplaceImport(Base: Pointer; ModuleName: PChar;
  FromProc, ToProc: pointer): boolean;
var
  NtHeader          : PImageNtHeaders;
  ImportDescriptor  : PImageImportDescriptor;
  ImportEntry       : PImageThunkData;
  CurrModuleName    : PChar;
  IsThunked         : Boolean;
  FromProcThunk     : PWin95CallThunk;
  ImportThunk       : PWin95CallThunk;
  FoundProc         : boolean;
begin
  Result := false;
  FromProcThunk := PWin95CallThunk(FromProc);
  IsThunked := (Win32Platform = VER_PLATFORM_WIN32_WINDOWS)
    and IsWin95CallThunk(FromProcThunk);
  NtHeader := GetImageNtHeader(Base);
  ImportDescriptor := PImageImportDescriptor(DWORD(Base)+
    NtHeader.OptionalHeader.DataDirectory[
    IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress);
  while ImportDescriptor^.NameOffset <> 0 do begin
    CurrModuleName :=
      PChar(Base) + ImportDescriptor^.NameOffset;
    if StrIComp(CurrModuleName, ModuleName) = 0 then begin
      ImportEntry := PImageThunkData(DWORD(Base) +
        ImportDescriptor^.IATOffset);
      while ImportEntry^.FunctionPtr <> nil do begin
        if IsThunked then begin
          ImportThunk :=
            PWin95CallThunk(ImportEntry^.FunctionPtr);
          FoundProc := IsWin95CallThunk(ImportThunk) and
            (ImportThunk^.Addr = FromProcThunk^.Addr);
        end else
          FoundProc := (ImportEntry^.FunctionPtr=FromProc);
        if FoundProc then begin
          ImportEntry^.FunctionPtr := ToProc;
          Result := true;
        end;
        Inc(ImportEntry);
      end;
    end;
    Inc(ImportDescriptor);
  end;
end;
```

```
function HookImport(ModuleName, ImportName: PChar; HookProc: pointer;
  var DLLProc: pointer): boolean;
begin
  Result := not Assigned(DLLProc);
  if Result then begin
    DllProc := Windows.GetProcAddress(Windows.GetModuleHandle(ModuleName),
      ImportName);
    Result := Assigned(DllProc) and ReplaceImport(Pointer(HInstance), ModuleName,
      DllProc, HookProc);
    if not Result then
      DLLProc := nil;
  end;
end;
function UnhookImport(ModuleName, ImportName: PChar; HookProc: pointer;
  var DLLProc: pointer): boolean;
begin
  Result := Assigned(DllProc) and ReplaceImport(Pointer(HInstance), ModuleName,
    HookProc, DllProc);
  if Result then
    DLLProc := nil;
end;
```

➤ *Listing 5: Hooking for novices.*

A Simpler Interface

That covers the workhorse of this unit, `ReplaceImport`. To give the outside world an easier to use interface, I have written two wrapper routines called `HookImport` and `UnhookImport`, see Listing 5.

Both routines take the same set of parameters, but have the opposite effects. `ModuleName` and `ImportName` must contain name of the DLL and imported routine, respectively. `HookProc` must contain the address of the routine we want to install or uninstall as a hook and `DLLProc` must be a procedure pointer. Strictly speaking, `UnhookImport` does not need the `ImportName` parameter, but I like the symmetry of it and it makes it easier to extend at a later date.

`HookImport` requires that `DLLProc` must be nil, while `UnhookImport` requires that `DLLProc` must be non-nil. If `HookImport` fails or `UnhookImport` succeeds, the `DLLProc` var parameter is set to nil. I have designed the routines this way because it makes it easier to use them safely. You don't have to maintain a separate `Boolean` variable to know if the hook has been installed or not. You can simply check the procedure variable containing the back-link to the actual DLL routine you're hooking.

`HookImport` calls `GetModuleHandle` and `GetProcAddress` to get the actual address of the DLL routine we're going to hook. We use `GetModuleHandle`, not `LoadLibrary`, as the module is already loaded by the process (otherwise it couldn't have been implicitly imported) and we don't want to increase the DLL's reference count.

Both routines leave the hard work to the `ReplaceImport` routine, sending `HInstance` as the `Base` parameter, meaning we want to hook DLL imports from the currently running module.

Listing 6 shows an example of how to use these routines, the code is taken directly from the TestHVHookDLL demonstration project on the CD-ROM.

There is nothing special here. The key issue is the procedural variable that keeps the original reference to the DLL routine. We use this to call the original routine from within our hooking routine and to restore things to normal in the `UnhookImport` call. We log whenever our hook is called by adding lines to the memo component.

In Figure 4 you can see the project running after having hooked the `MessageBeep` function. In the background you can see the Delphi 5 IDE with the Module View showing the address of the `MessageBeep` routine (`$BFF623C1`) and the CPU View showing the assembly instructions of the corresponding thunk. You can recognise the Win95 thunk with the characteristic `PUSH` and `JMP` instructions.

Getting Notifications Of Raised Exceptions

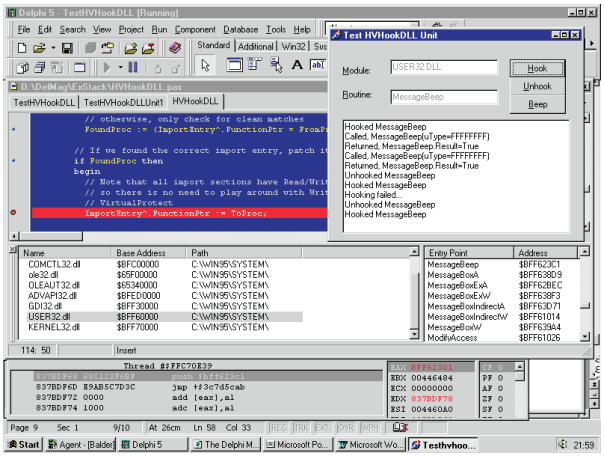
Now that we have the general DLL hooking in place and working, we are ready to move on to the next task. As I said at the start, we need to hook the exception handling system so we will know an exception has been raised. We found that we needed to hook the exception system at two different levels, `Kernel32.RaiseException` to get at the exceptions explicitly raised by the application and `ExceptObjProc` to get at exceptions generated by the CPU and OS.

We want to design a general solution. So instead of hard-coding specific logic inside the hooking routines, we will just call a generic callback procedure variable. The interface of the `HVExceptNotify` unit can be seen in Listing 7.

Anyone that needs to get notifications when exceptions are raised can just set the `ExceptNotify` variable to point to a procedure with the correct signature. It will get a pointer to the Delphi object

➤ *Listing 6: Using the hooking routines.*

```
var
  Windows_MessageBeep :
    function (uType: UINT): BOOL; stdcall;
function HookedMessageBeep(uType: UINT): BOOL; stdcall;
  const
    BoolStr : array[false..true] of string=('False','True');
begin
  if Assigned(Form1) then
    Form1.Memo1.Lines.Add(Format('Called,
      MessageBeep(uType=%.8x)', [uType]));
  Result := Windows_MessageBeep(uType);
  if Assigned(Form1) then
    Form1.Memo1.Lines.Add(Format('Returned,
      MessageBeep.Result=%s', [BoolStr[Result]]));
end;
procedure TForm1.HookBtnClick(Sender: TObject);
begin
  if HookImport('User32.dll', 'MessageBeep',
    @HookedMessageBeep, @Windows_MessageBeep)
  then Memo1.Lines.Add('Hooked MessageBeep')
  else Memo1.Lines.Add('Hooking failed...');
end;
procedure TForm1.UnhookBtnClick(Sender: TObject);
begin
  if UnHookImport('User32.dll', 'MessageBeep',
    @HookedMessageBeep, @Windows_MessageBeep)
  then Memo1.Lines.Add('Unhooked MessageBeep')
  else Memo1.Lines.Add('Unhooking failed...');
end;
procedure TForm1.BeepBtnClick(Sender: TObject);
begin
  Windows.MessageBeep($FFFFFFFF);
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  Form1 := nil;
end;
```



➤ *Figure 4: The DLL hooking demonstration program.*

representing the exception, a pointer to where in the code it was raised, and a boolean flag indicating if the source was an OS exception. We will now discuss how this unit is implemented.

Hooking RaiseException

Now that we have the `HVHookDLL`, it is very easy to hook the `RaiseException` routine in Kernel32: take a look at Listing 8.

We set up and take down the hooking in the `initialization` and `finalization` sections respectively. In the hooking routine itself, we check the parameters against some magic numbers known to be used internally in `System._RaiseException`. Note that Delphi 2 has a slightly different exception signature code to later versions. Once we have made sure we're being called from `_RaiseException`, we call the notification hook if it has been installed, passing the exception-object pointer, code address and false `OSException` flag. The pointers are obtained from the `Arguments` array sent to our `RaiseException` hook. Finally, we call back to the actual Kernel32 routine to get the job of raising the exception done.

Hooking ExceptObjProc

That should take care of all explicitly raised exceptions. Then we have to handle CPU and OS exceptions. Luckily, the `System` unit already defines a hook variable `ExceptObjProc` for this purpose. This is called whenever the exception core routine in `System` needs to convert an OS-style exception into a Delphi exception object. Normally, `SysUtils` redirects this pointer to its own internal routine, `GetExceptionObject`. We need to override this and redirect it to our own routine, see Listing 9.

Again we set up and take down the hooking in the `initialization` and `finalization` sections. We don't want to do all the hard work of mapping OS exceptions to Delphi exception objects ourselves, so we first call back to the original `SysUtils` hook to do it for us. Then we call the notification hook again if it has been installed, passing the exception object pointer, code address and true OSException flag. The object pointer has been obtained from the call to the `SysUtils` version of `ExceptObjProc` and the exception address is taken from a field of the `ExceptionRecord` parameter.

The combined code can be found in the `HVExceptNotify` unit on the CD-ROM. `TestExceptNotify` is a simple demonstration project that you can see running in Figure 5. If you look closely, you can see the code of the notification hook in the Delphi Editor View.

A Win32 Stack Tracer

Phew! We have finished the first main component of our exceptional stack tracer. We can now get a notification whenever an exception is raised anywhere in the application. Just displaying a simple message like the one in Figure 5 isn't very useful, however. The whole point was to gather more useful context information to make it easier to track down and fix the true cause of the exception.

The definition of what might be useful context information can vary according to what kind of application you are developing. The name of the currently focused

➤ *Listing 7: The interface for getting exception notifications.*

```
unit HVExceptNotify;
{ Unit that provides a notification service when exceptions are being raised }
interface
var
  ExceptNotify : procedure (ExceptObj: TObject; ExceptAddr: pointer;
    OSException: boolean);
implementation
```

➤ *Listing 8: Exception notification by hooking Kernel32.RaiseException.*

```
implementation
uses
  Windows, SysUtils, HVHookDLL;
var
  Kernel32_RaiseException : procedure (dwExceptionCode,
    dwExceptionFlags, nNumberOfArguments: DWORD;
    lpArguments: PDWORD); stdcall;
type
  PExceptionArguments = ^TExceptionArguments;
  TExceptionArguments = record
    ExceptAddr: pointer;
    ExceptObj : TObject;
  end;
procedure HookedRaiseException(ExceptionCode,
  ExceptionFlags, NumberOfArguments: DWORD; Arguments:
  PExceptionArguments); stdcall;
const
  cDelphiException    =
    {$IFDEF VER90}$0EEDFACE{$ELSE}$0EEDFADE{$ENDIF};
  cNonContinuable     = 1;
begin
  if (ExceptionFlags    = cNonContinuable)       and
     (ExceptionCode     = cDelphiException)      and
     (NumberOfArguments = 7)                     and
     (DWORD(Arguments)  = DWORD(@Arguments) + 4) then
  begin
    if Assigned(ExceptNotify) then
      ExceptNotify(Arguments.ExceptObj,
        Arguments.ExceptAddr, false);
  end;
  Kernel32_RaiseException(ExceptionCode, ExceptionFlags,
    NumberOfArguments, PDWORD(Arguments));
end;
initialization
  HookImport('Kernel32.dll', 'RaiseException',
    @HookedRaiseException, @Kernel32_RaiseException)
finalization
  UnHookImport('Kernel32.dll', 'RaiseException',
    @HookedRaiseException, @Kernel32_RaiseException);
```

form, the name of active database tables, the name of the user and other global information might be useful. You can easily add such information yourself.

However, in all cases, a complete overview of the function calls that preceded the raised exception will be most useful. To get that, we have to implement a stack tracer. This will analyse the current contents of the stack and try to figure out the return addresses stored there by the CPU as part of the `CALL` instruction operation.

YAST Nostalgia

As I mentioned earlier, I wrote a 16-bit stack tracer called YAST (*Ref 1*) back in the days of Delphi 1. I've now converted it to a 32-bit version and added some bells and whistles along the way: see the complete `HVYAST32` unit in Listing 10. The version on the CD-ROM is commented.

To Stack Frame Or Not

There are generally two different types of algorithms to choose from when implementing a stack tracer: the more elegant stack frame based algorithm and the raw brute force algorithm.

Stack Frame Based Tracing

The cleanest approach is to look at the stack in a logical way and try to find the so-called stack frames. Stack frames are a common convention used by subroutines to keep track of the stack location of parameters and local variables. If you look at the assembly code generated from a procedure using stack frames, you will see the pattern in Listing 11.

You can find more information about how stack frames work in Pietrek's article about assembly language (*Ref 10*). What is important for our purposes is that the `EBP` register points to a linked list of such frames stored on the stack. As can be seen from the `TStackFrame` record, the first field in a stack frame is the next `EBP` pointer and the second field is the return address of the caller routine. It is this return address information that we are interested in.

The `TraceStackFrames` procedure implements a basic, stack frame based stack tracer. It takes a pointer to a callback routine as its parameter. This callback will be called for each stack frame found on the stack. This way we have generalised the algorithm and can do whatever we want with each stack frame in our callback routine.

Let us look at the implementation of `TraceStackFrames`. First, we initialise the `Level` field of a stack info record to zero. The level will be increased for each new stack frame we find. Our callback routine can use this information to ignore the noise generated by the first few stack frames, these will be the stack frames of the stack tracer routines themselves.

Then we call the `InitGlobalVars` routine to initialise some global variables. The two variables `BaseOfCode` and `TopOfCode` contain the lowest and highest valid code segment addresses for the currently running module. These values are picked up from the PE-header information. The variables will be used to verify that potential code addresses lie within our code segment. For instance, we want to avoid reporting addresses from the Windows system modules. Finally, the `TopOfStack` variable is set to the highest valid stack address. This value is obtained from the per-thread information pointed to by the `FS` segment register, at offset 4.

Back in `TraceStackFrames`, we set up a pointer to the current stack frame by getting it from the `EBP` register. This stack frame will be the one belonging to the `TraceStackFrames` routine itself, because we have forced the compiler to use stack frames with the `{$W+}` directive. Then we set the global variable `BaseOfStack` to one less than `EBP`. This will be used together with `TopOfStack` to verify valid `EBP` pointers stored on the stack.

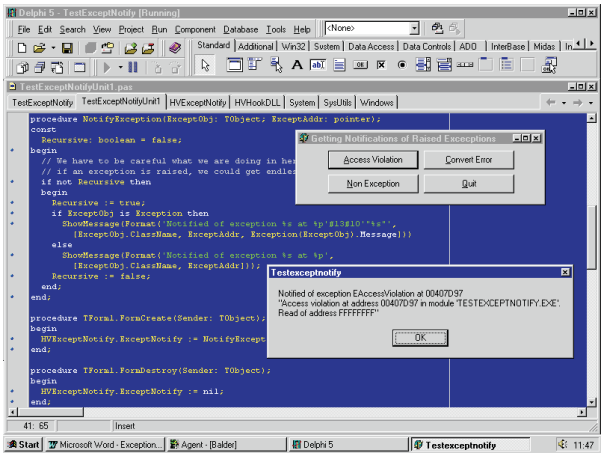
With the initial setup out of the way, we are ready to start looping, getting each stack frame using the `NextStackFrame` function and sending the results to our reporting callback routine. We keep looping until we run out of stack frames, or until our reporting callback says stop (by returning `False`).

The last piece of the puzzle is the `NextStackFrame` function. It is fairly straightforward. We check that the current stack frame pointer is a valid stack address. If the caller's address stored on the stack points to a valid code address in the

```
var
  SysUtils_ExceptObjProc: function (P: PExceptionRecord): Exception;
function HookedExceptObjProc(P: PExceptionRecord): Exception;
begin
  Result := SysUtils_ExceptObjProc(P);
  if Assigned(ExceptNotify) then
    ExceptNotify(Result, P^.ExceptionAddress);
end;
initialization
  SysUtils_ExceptObjProc := System.ExceptObjProc;
  System.ExceptObjProc := @HookedExceptObjProc;
  ...
finalization
  ...
  System.ExceptObjProc := @SysUtils_ExceptObjProc;
  SysUtils_ExceptObjProc := nil;
```

➤ *Listing 9: Exception notification by hooking ExceptObjProc.*

➤ *Figure 5: The exception notification demonstration project running.*



```
unit HVYAST32;
interface
uses
  Windows, SysUtils;
const
  MaxBlock = MaxInt-$f;
type
  PBytes  = ^TBytes;
  TBytes  = array[0..MaxBlock div SizeOf(byte)] of byte;
  PDWORDS = ^TDWORDS;
  TDWORDS = array[0..MaxBlock div SizeOf(DWORD)] of DWORD;
  PStackFrame = ^TStackFrame;
  TStackFrame = record
    CallersEBP : DWORD;
    CallerAdr  : DWORD;
  end;
  TStackInfo = record
    CallerAdr  : DWORD;
    Level      : DWORD;
    CallersEBP : DWORD;
    DumpSize   : DWORD;
    ParamSize  : DWORD;
    ParamPtr   : PDWORDS;
    case integer of
     0 : (StackFrame : PStackFrame);
     1 : (DumpPtr    : PBytes);
  end;
  TReportStackFrame = function(var StackInfo: TStackInfo;
    PrivateData: Pointer): boolean;
procedure TraceStackFrames(ReportStackFrame:
  TReportStackFrame; PrivateData: Pointer);
procedure TraceStackRaw(ReportStackFrame: TReportStackFrame;
  PrivateData: Pointer);
const
  MaxStackLevels = 50;
type
  TStackInfoArray = array[0..MaxStackLevels-1] of
    TStackInfo;
var
  StackDump: TStackInfoArray;
  StackDumpCount: integer;
function PhysicalToLogical(Physical: DWORD): DWORD;
function DefaultReportStackFrame(var StackInfo: TStackInfo;
  PrivateData: Pointer): boolean;
procedure SaveStackTrace(Raw: boolean; IgnoreLevels:
  integer; FirstCaller: pointer);
implementation
uses
  HVPEUtils;
{$W-}
function GetEBP: pointer;
asm
  MOV EAX, EBP
end;
function GetESP: pointer;
asm
  MOV EAX, ESP
end;
function GetStackTop: DWORD;
asm
  MOV EAX, FS:[4]
end;
var
  TopOfStack : DWORD;
  BaseOfStack: DWORD;
  BaseOfCode : DWORD;
  TopOfCode  : DWORD;
procedure InitGlobalVars;
var NTHeader: PImageNTHeaders;
begin
  if BaseOfCode = 0 then begin
    NTHeader := GetImageNtHeader(Pointer(hInstance));
    BaseOfCode :=
      DWord(hInstance) + NTHeader.OptionalHeader.BaseOfCode;
    TopOfCode :=
      BaseOfCode + NTHeader.OptionalHeader.SizeOfCode;
    TopOfStack := GetStackTop;
  end;
end;
function ValidStackAddr(StackAddr: DWORD): boolean;
begin
  Result :=
    (BaseOfStack < StackAddr) and (StackAddr < TopOfStack);
end;
function ValidCodeAddr(CodeAddr: DWORD): boolean;
begin
  Result :=
    (BaseOfCode < CodeAddr) and  (CodeAddr < TopOfCode);
end;
function ValidCallSite(CodeAddr: DWORD): boolean;
var
  CodeDWORD4: DWORD;
  CodeDWORD8: DWORD;
begin
  Result :=
    (BaseOfCode < CodeAddr) and  (CodeAddr < TopOfCode);
  if Result then begin
    CodeDWORD8 := PDWORD(CodeAddr-8)^;
    CodeDWORD4 := PDWORD(CodeAddr-4)^;
    Result :=
        ((CodeDWORD8 and $FF000000) = $E8000000)
      or ((CodeDWORD4 and $38FF0000) = $10FF0000)
      or ((CodeDWORD4 and $0038FF00) = $0010FF00)
      or ((CodeDWORD4 and $000038FF) = $000010FF)
      or ((CodeDWORD8 and $38FF0000) = $10FF0000)
      or ((CodeDWORD8 and $0038FF00) = $0010FF00)
      or ((CodeDWORD4 and $FF000000) = $C3000000);
  end;
end;
function NextStackFrame(var StackFrame: PStackFrame;
  var StackInfo : TStackInfo): boolean;
begin
  while ValidStackAddr(DWORD(StackFrame)) do begin
    if ValidCodeAddr(StackFrame^.CallerAdr) then begin
      Inc(StackInfo.Level);
      StackInfo.StackFrame := StackFrame;
      StackInfo.ParamPtr   := PDWORDS(DWORD(StackFrame) +
        SizeOf(TStackFrame));
      StackInfo.CallersEBP := StackFrame^.CallersEBP;
      StackInfo.CallerAdr  := StackFrame^.CallerAdr;
      StackInfo.DumpSize   :=
        StackFrame^.CallersEBP - DWORD(StackFrame);
      StackInfo.ParamSize  :=
        (StackInfo.DumpSize - SizeOf(TStackFrame)) div 4;
      StackFrame := PStackFrame(StackFrame^.CallersEBP);
      Result := true;
      Exit;
    end;
    StackFrame := PStackFrame(StackFrame^.CallersEBP);
  end;
  Result := false;
end;
{$W+}
procedure TraceStackFrames(ReportStackFrame:
  TReportStackFrame; PrivateData: Pointer);
var
  StackFrame : PStackFrame;
  StackInfo  : TStackInfo;
begin
  StackInfo.Level := 0;
  InitGlobalVars;
  StackFrame := GetEBP;
  BaseOfStack := DWORD(StackFrame) - 1;
  while NextStackFrame(StackFrame, StackInfo) and
    ReportStackFrame(StackInfo, PrivateData) do
    {Loop};
end;
procedure TraceStackRaw(ReportStackFrame: TReportStackFrame;
  PrivateData: Pointer);
var
  StackInfo : TStackInfo;
  StackPtr : PDWORD;
  PrevCaller: DWORD;
begin
  BaseOfStack := DWORD(GetESP);
  FillChar(StackInfo, SizeOf(StackInfo), 0);
  InitGlobalVars;
  PrevCaller := 0;
  StackPtr := PDWORD(BaseOfStack);
  while DWORD(StackPtr) < TopOfStack do begin
    if ValidCallSite(StackPtr^) and
      (StackPtr^ <> PrevCaller) then begin
      StackInfo.CallerAdr := StackPtr^;
      PrevCaller := StackPtr^;
      Inc(StackInfo.Level);
      if not ReportStackFrame(StackInfo, PrivateData) then
        Break;
    end;
    Inc(StackPtr);
  end;
end;
function DefaultReportStackFrame(var StackInfo: TStackInfo;
  PrivateData: Pointer): boolean;
begin
  Result := (StackDumpCount < MaxStackLevels-1);
  if Result and
    (DWORD(PrivateData) < StackInfo.Level) then begin
    Inc(StackDumpCount);
    StackDump[StackDumpCount] := StackInfo;
  end;
end;
procedure SaveStackTrace(Raw: boolean; IgnoreLevels:
  integer; FirstCaller: pointer);
begin
  StackDumpCount := -1;
  if Raw then
    TraceStackRaw(DefaultReportStackFrame,
      Pointer(IgnoreLevels))
  else
    TraceStackFrames(DefaultReportStackFrame,
      Pointer(IgnoreLevels));
end;
const
  LinkerOffset = $1000;
function PhysicalToLogical(Physical: DWORD): DWORD;
begin
  Result := Physical - DWORD(HInstance) - LinkerOffset;
end;
end.
```

➤ *Facing page: Listing 10.*

current module, we fill in the information field of the `StackInfo` record, step to the next stack frame by following the `EBP` pointer to the stack and then return back to `TraceStackFrames` to do the reporting of this stack frame. If it is not a valid code address, we follow the `EBP` pointer and try again.

That's all there is to it. The frame-based stack tracing is elegant and fairly fast, but it has one major weakness. It will not find callers that have no stack frames. With the current crop of optimising compilers, most smaller routines will not have stack frames and this reduces the usefulness of the stack tracer dramatically. There are two solutions to this. Either force stack frames for all your code, and preferably the VCL and RTL too, or use another algorithm.

Brute-Force Stack Tracing

While the frame-based stack tracing we have just been discussing tried to decode the logical structures stored on the stack, the brute-force method is much more primitive. The algorithm is easy: just look at all the `DWORD`s stored on the stack. If a `DWORD` happens to be a value that falls within the valid code segment of this module, include it in the stack trace.

To avoid getting too many false positives, we can add some more constraints (which we'll look at later). The brute force method will almost always find all the routines that have been called, but it is often plagued by giving false positives, so it can sometimes be hard to correctly interpret its output.

With the theory behind us, let's see how I implemented the brute force stack tracer in the `TraceStackRaw` procedure. It takes the same parameters as `TraceStackFrames` and will also use a reporting callback for each code address it finds. Note that the reporting callback will still get complete `TStackInfo` records, but only the `CallerAdr` and `Level` fields will contain valid information, the other fields will all be zero.

```
PUSH    EBP          // Save our caller's stack frame reference
MOV     EBP,   ESP   // Set up a reference to our stack frame
SUB     ESP,   XX    // Allocate space for local variables
// Other code here
MOV     EAX,   [EBP-XX]    // Reference a stack-based parameter
MOV     EDX,   [EBP+XX]    // Reference a stack-based local variable
// Other code here
ADD     ESP,   XX    // Remove the local variables
POP     EBP          // Restore our caller's stack frame
```

➤ *Listing 11: How stack frames are set up.*

First, we set the global variable `BaseOfStack` to the current value of the `ESP` register. This will mark the first slot of the stack we'll examine. Then we clear the `StackInfo` record to make sure all the unused fields are set to zero. We call `InitGlobalVars` again to initialise the same global variables as before.

`StackPtr` is our looping index into the stack. We start at the lowest valid stack address and continue until we have covered the whole stack. Inside the loop we check each `DWORD` on the stack to see if it can be a valid call site, using the `ValidCallSite` function. We also make sure we don't report the same caller's address more than once in a row, to avoid duplicates and to gracefully handle recursive routines. If we go through the tests, we store the caller's address in the `StackInfo` structure, increase the stack trace level and call the reporting function.

Validating Call Sites

The quality of the `ValidCallSite` function determines how many false positives and false negatives we will have. A false positive is an entry in the stack trace that should not have been there. A false negative is a missing entry from the stack trace. I spent quite some time refining and simplifying the code in this function. I admit that it looks rather cryptic, but that is a direct result of the strange Intel instruction encoding scheme (*Ref 11*).

The function takes one `DWORD` parameter and should return `True` if it considers the `DWORD` to be a valid return address. It does this by checking that the address lies within the current module's code segment and that the instruction preceding the return address may contain a `CALL` instruction.

First, it checks if the `CodeAddr` parameter is within the range which is defined by the two variables `BaseOfCode` and `TopOfCode`. Then it picks up the two `DWORD`s stored in the code segment just before the return address. These two `DWORD`s are then closely examined to see if their contents indicate a valid `CALL` instruction.

The Intel manual defines a number of variations of the `CALL` instruction, but for our purposes we are only interested in the ones typically used in application programs, so we only support the `CALL` instruction formats in Table 1 (see page 3-53 of the Intel manual).

Specifically, we don't support the less common inter-segment calling instructions. Delphi does not generate them anyway. The `CALL rel32` is easy. There is an op-code byte reserved for this use, so we only have to check for a `$E8` followed by four bytes of address data (`cd` means a `DWORD`).

The `CALL r/m32` is trickier. First we must check for the `$FF` op-code byte. The `/2` means that the following byte is bit-encoded and the 3-bit `Reg/Opcode` part of that byte must equal `010` binary. The complete byte looks like this: `mod` 2 bits, `reg/opcode` 3 bits and `r/m` 3 bits.

We're only interested in the `reg/opcode` part of the byte and want to make sure it is set to `010`, otherwise this could be a completely different instruction op-code. We filter out the unwanted bits using an `and` instruction, then check that the resulting value is correct. We want the `ModR/M` byte to be `XX010XXX`, were `X` means don't care and `1` and `0` means that the bit must be set or cleared. First we `and` the byte with `00111000`, or `$38`. Then we check that the result is equal to `00010000`, or `$10`.

For the `$FF` op-code there is always the op-code byte, the `ModR/M` byte and optionally 1, 2, 4 or

E8 cd	CALL rel32	Call near, relative, displacement relative to next instruction
FF /2	CALL r/m32	Call near, absolute indirect, address given in r/m32

➤ *Table 1*

5 bytes of offset data. So the total instruction can be 2, 3, 4, 6 or 7 bytes in length.

Instead of checking each op-code byte and `ModR/M` byte separately, I found that it is cleaner and more efficient to use `DWORD` bit masks to check for both bytes at the same time. The `$FF` versions of `CALL` have five different lengths so we must check for these bytes and bits in five different positions. To make that easier, I created Table 2.

The length column shows the total length of the `CALL` instruction. The `DWORD8` and `DWORD4` shows what bit mask values the corresponding code segment `DWORD` should be `and`'ed with. The match column gives the value that should result from the `and` operation to consider it an instruction match.

In fact, there are more bits we could check to make the logic even stricter and reduce the number of false positives. Specifically, we can check bits that determine what length the instruction should have. I didn't have time to explore this, but I might add it at a later date.

Now we have covered the most usual forms of the `CALL` instruction, but there is actually one other way of calling a routine, and that is by using the `RET` instruction. The compiler never calls routines this way, but low-level routines and manually encoded assembly might use this technique. It is not a very common idiom, but I wanted to support it anyway.

There are four forms of the return instruction, but we're only concerned with the two 'near' (intra-segment) versions, see Table 3. In fact, we will only bother with the first variant, the single byte instruction with `$C3` as the op-code, as we are only interested in hand-coded `PUSH XX / RET` sequences.

Now all those bit masks and `and` operations in `ValidateCallSite` should become easier to understand. If any of the bit mask/value pairs matches the bytes preceding the potential call site, it is considered valid. The commonest instruction types are checked first.

The Default Stack Tracer

The code we have discussed so far is very generic and doesn't store the result of the stack trace anywhere. That detail is left to the `ReportStackFrame` callback. To save the hassle of having to write the wrapper code again and again, I have also included a default stack tracer in the `HVYAST32` unit.

As can be seen from the `interface` part, this stack tracer is limited to storing fifty slots of stack trace information, but that should be plenty for most purposes. Call `SaveStackTrace` to run a stack trace and save the results in the `StackDump` array. `StackDumpCount` will then contain the number of valid slots in the array. `SaveStackTrace` takes three parameters. The `Raw` parameter determines which one of the two stack trace algorithms to use. Set it to `False` or `True` to get the frame-based or brute-force stack trace respectively. The `IgnoreLevels` parameter determines how many code addresses will be skipped before it starts filling the `StackDump` array. This is useful when you want to ignore the return addresses of the routines that implement the stack tracer itself. Finally, if you send it a non-nil `FirstCaller` parameter, that address will be added as the first entry in the stack dump array. This is useful for OS exceptions because the address that caused the exception is normally not on the stack.

TestYAST32 is a demonstration project that shows how to use these routines; you can see the code and application in Figure 6.

Dusting Off The RTLI

While having the stack trace in hand is a great step in the right direction, it is still rather cumbersome having to locate the correct copy of the project's MAP file (providing we have it somewhere) and then start searching for each logical address from the stack trace.

Ideally, the stack trace itself should include symbolic information (such as the unit name, filename, line number and routine name) the logical address corresponds to. Thanks to Vitaly Miryanov and his RTLI (*Ref 3*), we get this wonderful capability almost for free. He has already developed the framework and set of routines to make this possible. We just have to tweak the code a little to make it work with the newer compiler versions.

The updated units can be found on the CD-ROM. The changes I have made include: renaming LIGen.Pas to LIGen.Dpr to reflect that it is a project file, adding `{$APPTYPE CONSOLE}` in LIGen.Dpr; adding the `/V` option to specify what Delphi version of DCC32 to use, adding more checks to find the DCC32 path in the registry, adding `DWORD` typecasts to avoid warnings, plus removing the `rtliFixup` field and the `___Fixup___` routine. `OfsDelta` is now calculated using `HInstance`. I added an `RTLIAvailable` function, and `GetLocationInfo` now works correctly.

TestRTLI is a demonstration project where we use the RTLI routines to show symbolic information for the stack trace in addition to the raw addresses. You can see it running in Figure 7.

➤ *Table 2*

Length	DWORD8	DWORD4	Match
2		$000038FF	$000010FF
3		$0038FF00	$0010FF00
4		$38FF0000	$10FF0000
5			
6	$000038FF		$000010FF
7	$0038FF00		$000010FF

Putting The Pieces Together

Now we have separately developed the components of our exceptional stack tracer, so it is time to put it all together. The `HVEST` unit in Listing 12 does just this.

As you can see, this unit has a very simplistic interface. You should include it as the very first unit in your project file, but after `ShareMem`. Then, whenever an exception is raised in the application, a stack dump is saved into the global `StackDump` array. Note this happens even for exceptions that happen in other threads than the main thread. I have not added any thread-specific functionality, but that should be fairly easy to add.

If you for some reason want to turn off the exception stack tracing, set the `ESTEnabled` flag to `False`. By default the `HVEST` unit uses the stack frame-based tracing. If you prefer the raw, brute-force algorithm, set `ESTRaw` to `True`.

In the initialisation and finalisation sections we hook and unhook the `ExceptNotify` procedure variable. Whenever an exception is raised, our `NotifyException` procedure will be automatically invoked. Here we first check that we're not getting into a recursive loop. That could happen if there was a bug inside the code that caused another exception to be raised.

Then we have to determine how many levels of the stack trace should be ignored. The number of levels is different depending on which stack tracer we're using. I have found that three levels for the stack frame-based and five levels for the raw stack tracer works well. If the exception comes from the operating system, the address that caused the exception will most often be gone from the stack. To avoid this important address falling out of the stack trace, we explicitly send the address to the `SaveStackTrace` routine.

Finally, we chain back to any other hook that might have been installed for `ExceptNotify`.

C3	RET	Near return to calling procedure
C2 iw	RET imm16	Near return to calling procedure, pop imm16 bytes from stack
CB	RET	Far return to calling procedure
CA iw	RET imm16	Far return to calling procedure, pop imm16 bytes from stack

➤ *Table 3*

Demonstration Project

We have seen quite a number of demonstration projects in this article. In Figure 8 you can see the last one of them. The TestEST project shows how you can get fully symbolic stack traces whenever an exception is raised in your application.

The test form hooks the `Application.OnException` event in its `FormCreate` handler. When there is an exception, control eventually ends up in the `AppException` handler. The `HVEST` unit has already saved the stack trace, so we only have to present it to the user. In this case, we dump it out to the memo control on the form.

Application needs will vary here, but one good default solution is to dump the stack trace to a file in the same directory as the EXE file. The `HVDumpExceptToFile` unit does this, see Listing 13.

This is straightforward code. We hook the `Application.OnException` event. When there is an exception, we dump the contents of the stack trace to a file. The file has the same name and location as the EXE file, only the extension is changed to .EST. There is a project on the CD-ROM called TestFileDump that demonstrates this unit.

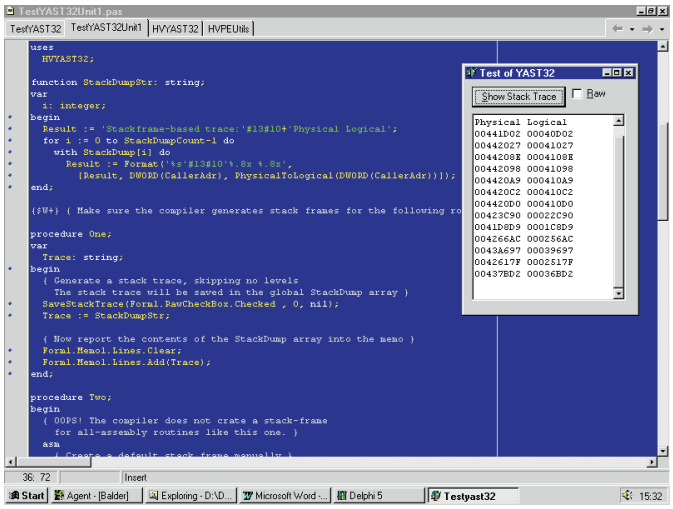
Step-By-Step Usage

Here is what you have to do to include the exception stack trace functionality in your own applications. First, you have to make sure you have the RTLI information for your project included as a special resource in the EXE file.

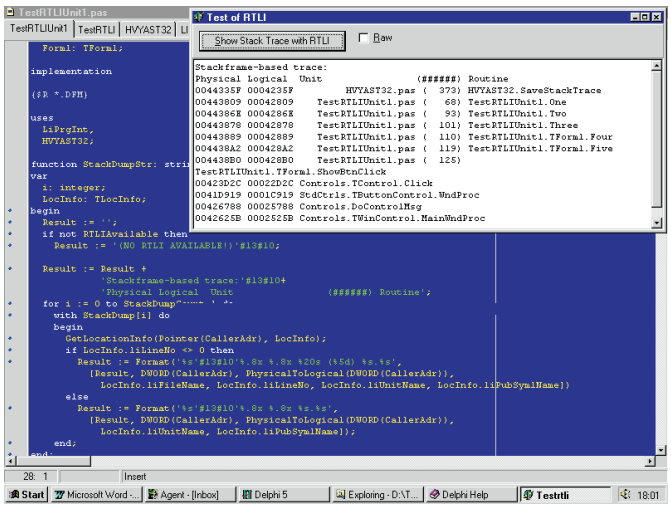
Install the RTLI Generator on the `Tools` menu with the parameters shown in Figure 9. Replace the `5` in the `/v5.0` option with the version of Delphi you are using.

Use the `Project|View Source` command to get the project file in editor. Add the `HVEST` unit as the very first unit (after `ShareMem`) in

➤ *Figure 6: The frame-based stack tracer in action.*



➤ *Figure 7: Demonstrating stack trace with symbolic RTLI.*



```
unit HVEST;
interface
var
  ESTEnabled: boolean = true;
  ESTRaw    : boolean = false;
implementation
uses
  HVExceptNotify, Windows, SysUtils, HVYAST32, HVHookDLL;
{$W+}
var OldExceptNotify: TExceptNotify;
procedure NotifyException(ExceptObj: TObject;
  ExceptAddr: pointer; OSException: boolean);
const
  Recursive: boolean = false;
var
  IgnoreLevels: integer;
  FirstCaller : pointer;
begin
  if not Recursive then begin
    Recursive := true;
    IgnoreLevels := 3;
    if ESTRaw then
      IgnoreLevels := 5;
    FirstCaller := nil;
    if OSException then
      FirstCaller := ExceptAddr;
    SaveStackTrace(ESTRaw, IgnoreLevels, FirstCaller);
    if Assigned(OldExceptNotify) then
      OldExceptNotify(ExceptObj, ExceptAddr, OSException);
    Recursive := false;
  end;
end;
initialization
  OldExceptNotify := HVExceptNotify.ExceptNotify;
  HVExceptNotify.ExceptNotify := NotifyException;
finalization
  HVExceptNotify.ExceptNotify := OldNotifyException;
  OldExceptNotify := nil;
end.
```

➤ *Listing 12: The HVEST unit implements the Exceptional Stack Tracer.*

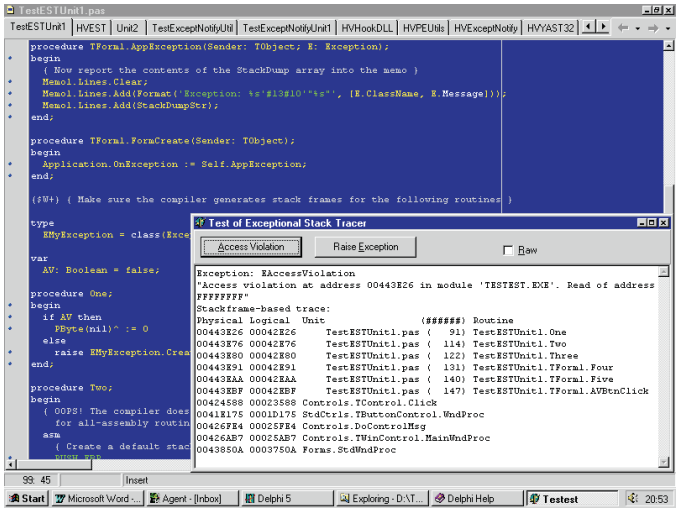
your project file. Add the `HVDumpExceptToFile` unit to the projects `uses` clause. Add the following lines just below the existing `{$R *.RES}` directive in the project file:

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

Note that `BindingRTLI` should *not* be defined in your project's options.

With the project source as the current file in the editor, select `Tools | RTLI Generator`. Now the LIGen utility will recompile your project twice: once to update the .MAP file and once to link the RLI resource into the EXE file.

If you are careful, you can still debug your application from within the IDE. Just make sure the IDE does not recompile the project file, because then you will lose the RTLI resource.



➤ *Figure 8: The stack tracer in action.*

This approach seems to work with Delphi 5. Use `Project | View` source to make the project file current in the IDE. Select `File | Save` all to save all the changes.

➤ *Listing 13: Dumping the exception stack trace to a file.*

```
unit HVDumpExceptToFile;
interface
implementation
uses
  HVEST, HVYAST32, Windows, SysUtils, Forms, LiPrgInt;
type
  TAppExceptionHandler = class(TObject)
  private
    procedure AppException(Sender: TObject; E: Exception);
  end;
function StackDumpStr: string;
var
  i: integer;
  LocInfo: TLocInfo;
begin
  Result := '';
  if not RTLIAvailable then
    Result := '(NO RTLI AVAILABLE!)'#13#10;
  if ESTRaw then
    Result := Result + 'Stack trace (raw):'#13#10
  else
    Result := Result + 'Stack trace:'#13#10;
  Result := Result +
            'Physical Logical  Unit  (######) Routine';
  for i := 0 to StackDumpCount-1 do
    with StackDump[i] do begin
      GetLocationInfo(Pointer(CallerAdr), LocInfo);
      if LocInfo.liLineNo <> 0 then
        Result :=
          Format('%s'#13#10'%.8x %.8x %20s (%5d) %s.%s',
          [Result, DWORD(CallerAdr), PhysicalToLogical(
          DWORD(CallerAdr)), LocInfo.liFileName,
          LocInfo.liLineNo, LocInfo.liUnitName,
          LocInfo.liPubSym1Name])
      else
        Result := Format('%s'#13#10'%.8x %.8x %s.%s',
          [Result, DWORD(CallerAdr),
          PhysicalToLogical(DWORD(CallerAdr)),
          LocInfo.liUnitName, LocInfo.liPubSym1Name]);
    end;
end;
procedure TAppExceptionHandler.AppException(Sender: TObject;
  E: Exception);
var F: System.Text;
    LogFileName: string;
begin
  {$I-} { Don't raise any EInOutError exceptions in here }
  LogFileName :=
    SysUtils.ChangeFileExt(System.ParamStr(0), '.EST');
  System.Assign(F, LogFileName);
  System.Append(F);
  if IOResult <> 0 then
    System.Rewrite(F);
  try
    System.Writeln(F);
    System.Writeln(F, DateTimeToStr(Now));
    System.Writeln(F, 'Exception: ', E.ClassName);
    System.Writeln(F, 'In ', ParamStr(0));
    System.Writeln(F, E.Message);
    System.Writeln(F, StackDumpStr);
    if IOResult <> 0 then ;
  finally
    System.Close(F);
    if IOResult <> 0 then ;
  end;
  E.Message := E.Message +
    #10#13'The exception has been logged in '+LogFileName;
  Application.ShowException(E);
end;
var AppExceptionHandler : TAppExceptionHandler;
initialization
  AppExceptionHandler := TAppExceptionHandler.Create;
  Application.OnException :=
    AppExceptionHandler.AppException;
finalization
  AppExceptionHandler.Free;
  AppExceptionHandler := nil;
end.
```

References *(in The Delphi Magazine except where noted)*

References quoted in the text:

1. Hallvard Vassbotn, Issue 7, March 1996, *YAST: Yet Another Stack Tracer!*
2. Vitaly Miryanov, Issue 22, June 1997, *RunTime Location Information In Delphi 2*
3. Per Larsen's ExHook utility: http://home.turbopower.com/~perl/page3.html
4. Stefan Hoffmeister's Delphi Debugging Package: http://www.econos.de/software/borland/mapper/index.html
5. MuTek Solutions' BugTrapper: http://www.mutek.com/bthome.html
6. Matt Pietrek, MSJ March 1994: *Peering inside the PE: A Tour of the Win32 Portable Executable File Format*: http://msdn.microsoft.com/library/techart/msdn_peeringpe.htm
7. Hallvard Vassbotn, Issue 43, March 1999, *DelayLoading of DLLs*
8. Dave Jewell, Issue 26, Oct 1997, *Beating the System: Runtime Library Determination Utility*
9. Matt Pietrek, MSJ Under the hood, on the Thread Information Block (TIB): http://msdn.microsoft.com/library/periodic/period96/periodic/msj/F1/D6/S2CE.htm
10. Matt Pietrek, MSJ Under the hood, on compiler generated assembly code: http://msdn.microsoft.com/library/periodic/period98/html/procedureentryexit.htm
11. Intel Corp, *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference* (6.5Mb .PDF): http://developer.intel.com/design/pentiumii/manuals/243191.htm

Other useful references:

Chad Z. Hower, Issue 39, November 1998, *Preventative Programming: Code Smarter, Debug Less*

Matt Pietrek, MSJ Under the hood, *A Crash Course on the Depths of Win32 Structured Exception Handling*: http://msdn.microsoft.com/library/periodic/period97/periodic/msj/1201/pietrek.htm

Bob Swart, Issue 8, April 1996, *Structured Exception Handling*

Andrew McLellan, Issue 13, September 1996, *Cascading Exceptions*

Dave Jewell, Issue 17, January 1997, *Beating the System: Code Generation Part 2*, on how exceptions work in Delphi1

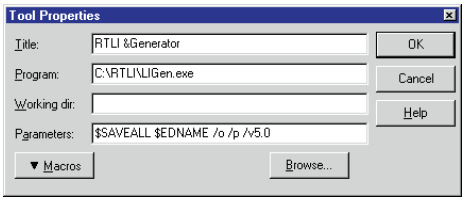
Press `Ctrl+F9` to recompile the project. Chose `Tools | RTLI Generate` to generate and bind the RTLI information. Now, without any further changes, press `F9` to run and debug the project in the IDE.

Conclusion

There are many possible ways to improve and extend the exception stack tracer. We can make it more thread-aware. The `ValidateCallSite` function of the brute-force stack tracer can be improved to reduce the number of false positives reported. We could develop a hybrid stack tracer combining the strengths of the stack-frame and brute-force algorithms.

However, even in its present state, I think that it represents a major improvement over traditional exception reporting.

Special thanks are due to Vitaly Miryanov for the RTLI technology. Thanks also to Lars Fosdal of Reuters Norge for helping me check the code and article.



➤ *Figure 9: Setting up the Tools menu.*

Hallvard Vassbotn is a Senior Software Developer at Million Handshakes AS. You can reach him at hallvard@millionhandshakes.com