Safecall Defined

Handling exceptions in COM and Automation

by Brian Long

If you have dabbled with Automation or COM in Delphi, you have probably seen the reserved word `safecall` scattered about the source code generated automatically by Delphi. You might also have bumped into the term in the Environment Options dialog. If you have ever seen it and wondered what it meant, then read on.

I have seen a few references to `safecall`, but nothing that covers the whole subject from start to finish, so I'll try and rectify that now. As usual, we will firstly look a little into the background of the subject.

On what initially appears to be an unrelated subject, you can find many references telling you that when you write a DLL, you must not allow an exception to escape from an exported routine. The DLL client might not be written in Delphi, and so wouldn't know what to do about it. Instead you are supposed to write exception handlers around every exported routine and, if an exception occurs, you should return some error value.

The COM Exception Rule

COM applications have a similar rule. They must not allow any exceptions to bubble out, and instead must catch and handle them all. This is because COM doesn't support propagation of exceptions across proxied interface calls, meaning calls to COM object interface methods in out-of-process servers, MTS apartment model servers or remote DCOM servers. However, COM does provide a mechanism that allows a COM object to tell a client that an exception occurred, and give some details about it.

This so-called rich error information is provided by COM error objects. The Win32 COM system supports one error object per logical thread in a COM application. If an exception occurs, and the server supports error objects, the error object will be filled up via its `ICreateErrorInfo` interface with information about the error. If the client sees that the server supports error objects, it can retrieve the error object's `IErrorInfo` interface and manufacture a suitable exception with the available information on the client side to report the problem.

COM objects indicate that they support error objects by implementing the `ISupportErrorInfo` interface. Delphi's COM and Automation objects do just that.

In many development tools, the business of catching server exceptions and filling up error objects is down to the programmer. As seems to be very common with writing Delphi COM applications, the Inprise developers have relieved you of this task. This is exactly what the `safecall` directive does. An interface method marked `safecall` uses the same calling conventions as a `stdcall` method, but does a fair amount of behind-the-scenes work on your behalf.

When you use the IDE wizard and type library editor to create an Automation object, the IDE creates an interface definition and also an implementation of the interface in a class. If you look carefully at the method declarations in the generated interface and class, you will see them all marked with the `safecall` calling convention. The same is true of COM object methods if the environment options are set appropriately.

If an exception happens in a COM interface method which is marked with `safecall`, and does not get explicitly caught with a `try..except..end` statement, Delphi code automatically sets up an error object in case the client wants to make use of it.

Any Delphi client program that calls a `safecall` method will take note of any error object that gets generated and automatically raise an exception on the client side.

Two Sides To Safecall

So there are two sides to the Delphi `safecall` directive. The implementation of a COM `safecall` method picks up unhandled exceptions and builds error objects. But also, when a `safecall` method is invoked, any reported COM error is picked up and transformed into a Delphi exception.

The `safecall` directive is a language construct unique to Delphi, but it was created to support and implement defined COM error reporting standards. Consequently, a Delphi COM interface with `safecall` methods can be called from any non-Delphi client. Also, any non-Delphi written COM server can have its interface modified appropriately to use the safecall directive and still be accessed perfectly well by a Delphi client.

Now that we have an idea of what `safecall` does, let's dig a little deeper. The operation of `safecall` is based around COM function return values. To indicate what happened during its execution, a COM routine returns an `HResult`, as defined in the sidebar on page 13. The `HResult` value indicates if something went wrong or not, and often goes on to describe exactly what it was that went wrong. If an exception happens in a COM routine it is typical to have an `HResult` returned indicating an error, but if you examine any `safecall` routine, you will not see any evidence of an `HResult` in the definition.

This apparent lack of an `HResult` is deceptive because all `safecall` routines *do* actually return one, but it is hidden in the code generated for you by Delphi. Let's take an example method declaration, plucked from an interface implemented by a Delphi Automation object:

```
procedure Foo(const Msg:
  WideString); safecall;
```

This looks to the reader as a procedure method that takes a wide string parameter. However, because it is marked as `safecall`, the routine is actually implemented as if it were defined as:

```
function Foo(
  const Msg: WideString):
  HResult; stdcall;
```

Similarly, a function method declared:

```
function Bar(const D: Double):
  WideString; safecall;
```

is really implemented like this:

```
function Bar(D: Double;
  out Value: WideString):
  HResult; stdcall;
```

You can see the original function return value is actually passed back through the last parameter, making way for the `HResult` to be the function result. Whilst we Delphi developers see the former declarations of these routines, the outside world will always see the latter versions (the type library records the full `HResult` returning versions).

So why does `safecall` hide the `HResult` from us? Put simply, it is to make things easier for us whilst building COM applications. In most cases, when implementing an interface in a COM server, the only `HResult` codes that need to be returned are `S_OK` for success and `E_UNEXPECTED` for failure. With the error object fully describing the problem, more details in the `HResult` are not strictly necessary.

So what a `safecall` routine does is to wrap an exception handler around the code in your interface method implementation. If no exception happens, an `HResult` of `S_OK` is returned. However, if an unhandled exception occurs, the error object is set up for the client application to pick up, and an `HResult` of `E_UNEXPECTED` is returned. On the client side, when you call a `safecall` method, Delphi checks the returned `HResult`. If it is an error code, the error object is retrieved to recreate the exception.

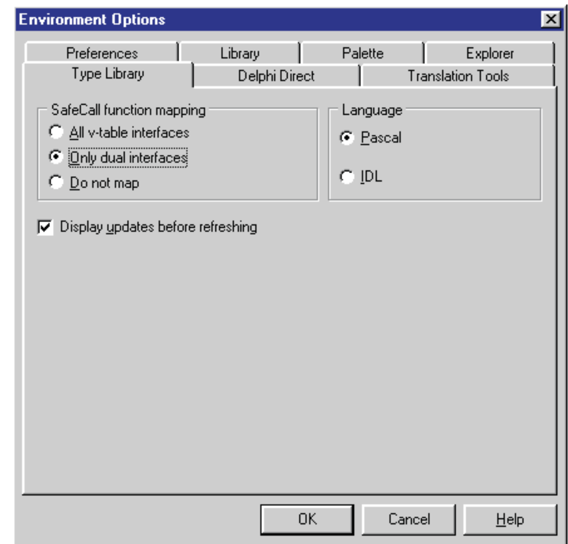
This makes it look like the COM server raised an exception and the client reported it, which is great! For the Delphi programmer, there are no `HResult` values, and no need to be concerned about getting the exception across process boundaries. The exception just magically propagates from server to client without any help from the developer.

With this information to hand, you can now see another example of how COM applications written in Delphi are that much easier to put together than in other languages. The COM error conventions are catered for automatically through `safecall`. This adds to Delphi's automatic generation of the calls to `IUnknown`'s reference counting methods, encapsulation of both safe arrays and late bound Automation within the Delphi `Variant` data type and no-fuss COM aggregation with the `implements` keyword.

Is There A Downside?

So I have painted a very rosy picture of the effect of `safecall`, but it is not without its little problems. The `safecall` syntax does not provide access to the `HResult` that will be returned. For your own interfaces this should not pose a problem. However, if you are implementing a pre-defined interface in a COM server, and if that interface specifies that certain `HResult` values must be returned under various circumstances, then `safecall` is not an option. The same is true if you decide to return custom `HResult` values from COM methods.

In these cases, you will have to forego the `safecall` directive, and change the declaration of the routine in the type library editor (on the `Text` page, where you can see the whole declaration listed). This



➤ *Figure 1: Safecall and type library editor options in Delphi 5.*

is best done with the type library language set to Pascal, as opposed to IDL, in the `Type Library` page of the `Environment Options` dialog (see Figure 1). Change `safecall` to `stdcall`, add the `HResult` return type and, if the routine started life as a `safecall` function (as opposed to a procedure), make a new `out` parameter to take care of what was the function result type. When the type library editor refreshes the source files, this change will be reflected in the COM implementation class. The method implementation must also be changed to include an exception handler, to prevent any exceptions going unhandled.

As an example of changing `safecall` declarations, Figure 2 shows the two interface methods from earlier defined as `safecall` in the type library editor (it shows the whole interface in textual form). Figure 3 then shows how to edit these methods to turn them into non-`safecall` versions with access to the `HResult`.

Another issue is that of hardware exceptions, such as Access Violations and divide by zero errors. If one of these is raised in a `safecall` method, the RTL code is unable to correctly set up the error object with the Delphi exception details. This is because the raised exception is not a Delphi software exception at that point, it is

instead an OS hardware exception. Because there is no Delphi exception object to work with, no error object is set up. Instead, `E_UNEXPECTED` is returned directly and the client ends up generating an exception saying simply: *Unexpected failure*. This is the textual description of the `E_UNEXPECTED` `HResult`.

In Delphi 4 and 5, you can control whether all Automation object and COM object methods are automatically marked as `safecall` with another option on the `Type Library` page of the `Environment Options` dialog (see Figure 1 again). The `SafeCall function mapping` group box gives you three choices. The default setting (`Only dual interfaces`) ensures methods of all dual interfaces (those created with the Automation object) are marked `safecall`, where COM objects will be marked `stdcall`. The `All v-table interfaces` option will mark COM object and Automation object methods as `safecall`, and the `Do not map` option prevents Delphi marking any methods as `safecall`.

If you needed to access the `HResult` for all your COM methods, you may choose to disable `safecall` function mapping to save on the manual editing of method declarations in the type library editor.

Safecall Internals

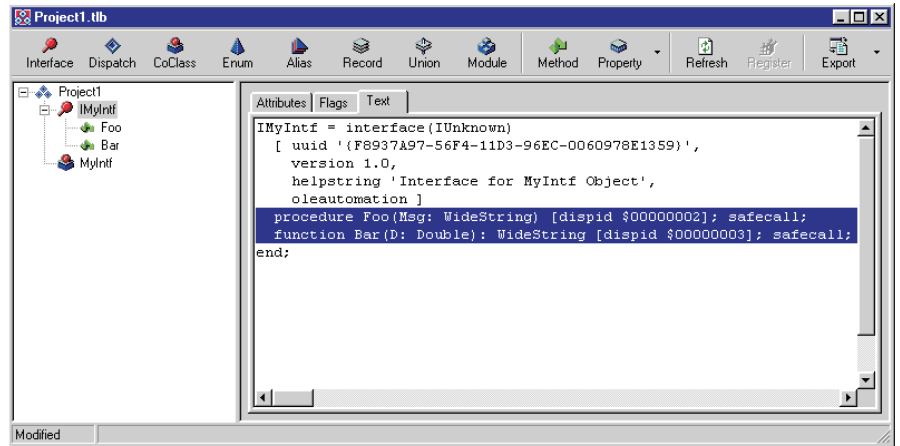
What has been discussed so far represents the most well known side of `safecall` subroutines. But there is more information surrounding the subject, pertaining to the underlying mechanics of `safecall`. The fact that an exception causes a `safecall` method to return `E_UNEXPECTED` is not intrinsically linked with the `safecall` reserved word. The `HResult` is actually returned from the object's virtual `SafeCallException` method.

When an exception occurs in a `safecall` method, code in the `System` unit's `HandleAutoException` routine calls the method whose address is stored `vmtSafeCallException` bytes before the object's main VMT. This equates to calling the virtual `SafeCallException` method, which is originally defined in `TObject`:

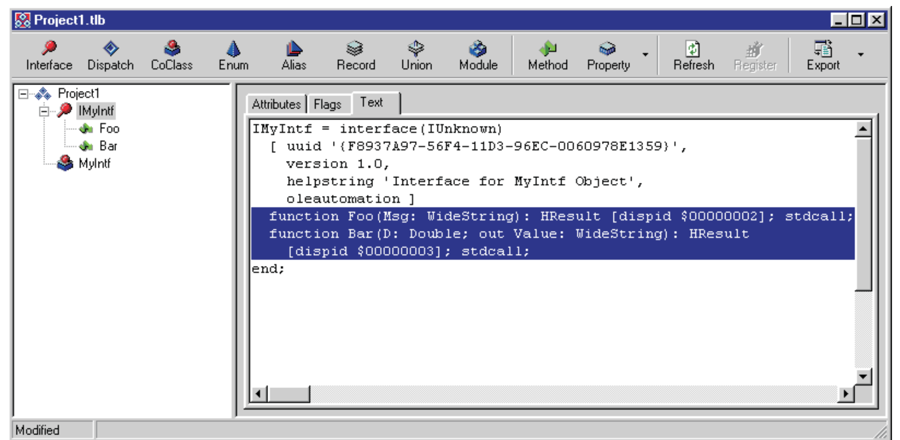
```
function SafeCallException(
  ExceptObject: TObject;
  ExceptAddr: Pointer):
  HResult;
```

The implementation of `TObject.SafeCallException` simply ignores its two parameters and returns `E_UNEXPECTED`. The COM object classes override this method to additionally set up the COM error object. `TComponent` also overrides this method, primarily to help when a VCL component is wrapped up as an ActiveX. If there is a COM object connected to the component's `ComObject` property (and by definition, also to its `VCLComObject` property), `TComponent.SafeCallException` makes use of the COM object's `SafeCallException` method.

A `safecall` error return value can only be correctly picked up by the code that called the routine if the address of a `safecall` error return handling routine has been assigned to the global `SafeCallErrorProc` pointer variable (which defaults to `nil`). This routine takes the error code and address, and should react to the problem by raising some appropriate



➤ *Figure 2: Normal safecall methods in the type library editor.*



➤ *Figure 3: Safecall methods changed back to stdcall.*

```
procedure TSomeClass.Foo(const Msg: WideString);
begin
  //Your code goes here, e.g.
  Application.MainForm.Tag := StrToInt(Msg)
end;
function TSomeClass.Bar(D: Double): WideString;
begin
  //Your code goes here, e.g.
  Result := FloatToStr(D)
end;
```

➤ *Listing 1*

HResults Dissected

To indicate problems, COM object methods return an HResult (result handle). An HResult is a bit of a misnomer as it is not really a handle to anything. It is just a 32-bit integer value used to return success, failure or warning codes.

The high bit of an HResult (bit 31, called the severity bit) indicates success (if clear) or failure (if set). The next four bits are reserved by Windows and must currently be zero. The other eleven bits of the high word represent the facility code, and specify which group of status codes the HResult belongs to, effectively identifying the system service that generated the error. The low word represents the error code and indicates what happened.

To explicitly examine the severity bit, which means to find out if the HResult-generating call succeeded or failed, you have several options. The preferred approach is to use the Succeeded or Failed functions which do the check for you. Other options include calling HResultSeverity and comparing the result against SEVERITY_SUCCESS or SEVERITY_ERROR. Finally, there is an IsError function which does the same as Failed.

Routines also exist for getting the other sections out of an HResult, including HResultCode and HResultFacility. More exist to manufacture HResult values, such as MakeResult (out of a severity, facility and status code) and HResultFromWin32. All these routines (and constants) are declared in the Windows import unit in 32-bit Delphi. Additionally, the ComObj unit (or OleAuto unit in Delphi 2) defines a helper routine called OleCheck. This takes an HResult and, if it indicates failure, will raise an EOleSysError exception.

Delphi 4 made a small *faux pas* with respect to the Delphi translation of the HResult type. It is supposed to be a signed type and was correctly defined to be a Longint in Delphi 3. Delphi 4 (wrongly) changed this to a DWord (itself defined as a LongWord), but Delphi 5 repairs the damage. When correctly defined as signed, just checking if the value is negative (or not) will indicate failure (or success). However, the changing definition will not affect your applications if they only use the utility routines, such as Succeeded and Failed.

A handful of sample HResult values are listed in Table 1, with small commentary on the meaning held by their bits. Specific HResult error codes are named with this pattern:

```
<Facility>_<Severity>_<Reason>
```

where <Facility> is either the facility code, or other distinguishing identifier, <Severity> is either S (success) or E (error) and <Reason> is a short descriptive identifier. Error codes within FACILITY_NULL omit the <Facility>_ prefix, for example E_UNEXPECTED. The currently available facility codes are listed in Table 2.

Microsoft is responsible for creating new facility codes, and also for creating error codes in all facilities other than these. Custom HResult codes should really use FACILITY_ITF and the error code value should be between $200 and $FFFF (COM reserves the lower codes).

Windows error constant	**Value**	**Severity Bit**	**Facility Code**	**SCode**	**Meaning**
S_OK or NOERROR	$00000000	SEVERITY_SUCCESS (0)	FACILITY_NULL (0)	0	Function worked and returned True
S_FALSE	$00000001	SEVERITY_SUCCESS (0)	FACILITY_NULL (0)	1	Function worked and returned False
E_UNEXPECTED	$8000FFFF	SEVERITY_ERROR (1)	FACILITY_NULL (0)	$FFFF	Catastrophic failure
E_NOTIMPL	$80004001	SEVERITY_ERROR (1)	FACILITY_NULL (0)	$4001	Method has no useful implementation
DISP_E_MEMBERNOTFOUND	$80020003	SEVERITY_ERROR (1)	FACILITY_DISPATCH (2)	3	Automation method/ property does not exist in server
DISP_E_PARAMNOTFOUND	$80020004	SEVERITY_ERROR (1)	FACILITY_DISPATCH (2)	4	Parameter in Automation method does not exist

➤ *Above: Table 1*

➤ *Below: Table 2*

Facility Code	**Meaning**	**Sample HResult**
FACILITY_NULL (0)	Error code with no specific grouping	E_UNEXPECTED ($8000FFFF)
FACILITY_RPC (1)	Error code from an underlying Remote Procedure Call	RPC_E_SERVER_DIED ($80010007)
FACILITY_DISPATCH (2)	Error code related to IDispatch interface	DISP_E_TYPEMISMATCH ($80020005)
FACILITY_STORAGE (3)	Persistent storage related error code. Status codes lower than 256 have the same meaning as the corresponding DOS error code	STG_E_FILENOTFOUND ($80030002)
FACILITY_ITF (4)	Interface-specific error code. If another interface returns the same HResult, you can expect the meaning to be different	OLE_E_OLEVERB ($80040000)
FACILITY_WIN32 (7)	The status code is a Win32 API result code or network error mapped to an HResult	E_OUTOFMEMORY ($8007000E)
FACILITY_WINDOWS (8)	Error code from a Microsoft-defined interface	CO_E_CLASS_CREATE_FAILED ($80080001)
FACILITY_SSPI (9)	CryptoAPI-related error code	NTE_BAD_HASH ($80090002)
FACILITY_CONTROL ($A)	OLE-control related error code	
FACILITY_CERT ($B)	Trust Verification related error code	TRUST_E_PROVIDER_UNKNOWN ($800B0001)

exception. When writing a COM application, the ComObj unit's initialisation section installs a safecall error handler that can pick up an error object and manufacture an EOleSysError exception.

If a safecall method has been set up in a non-COM object, and it lets an exception go unhandled, then things don't initially work as well as within a COM application. A call to a safecall method that returns a value other than S_OK will trigger the System unit's CheckAutoResult procedure. When SafeCallErrorProc is found to be nil, this causes the SysUtils error and exception handling to generate a runtime error with the internal code reSafeCallError.

In an application without SysUtils used anywhere, this will cause a runtime error 229. With SysUtils used, all runtime errors should be reported as specific exceptions. Unfortunately, Delphi 3 and 4 had no dedicated exception type for this runtime error, so you end up with an EInOutError raised. In Delphi 5 and higher, you get a more descriptive ESafecallException exception.

➤ *Listing 3*

```
intf.Foo('1234');
...
intf.Bar(Pi);
```

```
function TSomeClass.Foo(const Msg: WideString): HResult;
begin
  try
    Result := S_OK;
    //Your code goes here, eg
    Application.MainForm.Tag := StrToInt(Msg)
  except
    //The COM object's SafeCallException will set Result
    //to E_UNEXPECTED and also set up a COM error object
    Result := SafeCallException(ExceptObject, ExceptAddr)
  end;
end;
function TSomeClass.Bar(D: Double; out Value: WideString): HResult;
begin
  try
    Result := S_OK;
    //Your code goes here, eg
    Value := FloatToStr(D)
  except
    //The COM object's SafeCallException will set Result
    //to E_UNEXPECTED and also set up a COM error object
    Result := SafeCallException(ExceptObject, ExceptAddr)
  end;
end;
```

➤ *Listing 2*

You will also get this same error/exception generated if the routine referred to by SafeCallErrorProc fails to raise an exception.

Undressing Safecall

Now that we know how safecall works, we can go back to the sample safecall interface method declarations shown earlier (Foo and Bar) and consider their implementations in a COM server object. The source code you would typically see for these safecall methods is shown in Listing 1, but their actual implementations would be more like Listing 2 by the time Delphi compiles them.

On the client side, Listing 3 shows how we should call safecall interface methods. Listing 4 shows what really happens during safecall method invocations, thanks to the extra code Delphi generates. As you can see, the HResult return value gets checked for error status. If it is an error code, the address at which the error occurred is extracted from the stack and passed, along with the HResult, to the safecall error handler. If no handler exists, a runtime error is generated. Normally, SysUtils will turn this into an exception, but if SysUtils has not plugged itself in, a terminal runtime error is generated.

➤ *Listing 4*

```
procedure CheckHResult(Res: HResult);
type
  TErrorProc = procedure(ErrorCode: Integer; ErrorAddr: Pointer);
var
  ErrorAddr: Pointer;
const
  reSafeCallError = 24;
  rteSafeCallError = 229;
begin
  if Failed(Res) then begin
    //Get exception address from stack
    asm
      mov ECX, [ESP+20]
      mov ErrorAddr, ECX
    end;
    //Call safecall error handler, if set
    if Assigned(SafeCallErrorProc) then
      TErrorProc(SafeCallErrorProc)(Res, ErrorAddr)
    else
      //No safecall error handler, so call RTL error handler, if installed
      if Assigned(ErrorProc) then
        TErrorProc(ErrorProc)(reSafeCallError, ErrorAddr)
      else
        //No RTL error handler, so generate run-time error
        RunError(rteSafeCallError)
  end
end;
...
var
  FuncRes: WideString;
...
CheckHResult(intf.Foo('hello'));
CheckHResult(intf.Bar(Pi, FuncRes));
```

More Safecall Support

In addition to COM error propagation, the COM and Automation object classes from the ComObj unit have extra code in their SafeCallException method (from Delphi 4 onwards). They cater for a possible requirement for COM servers to report safecall exceptions to some dedicated error logging process. Normally, a server exception is raised on the client, and would usually produce a modal dialog that needs user intervention to remove.

For hands-off systems, the programmer can create an object that implements the ComObj unit's IServerExceptionHandler interface and assign it to the COM object's ServerExceptionHandler property. Whenever a safecall exception happens, this causes the interface's OnException method to

```
TErrorLogger = class(TInterfacedObject,
IServerExceptionHandler)
protected
  procedure OnException(const ServerClass, ExceptionClass,
    ErrorMessage: WideString; ExceptAddr: Integer;
    const ErrorIID, ProgID: WideString;
    var Handled: Integer; var Result: HResult);
end;
...
procedure TErrorTrappingTest.Initialize;
begin
  inherited;
  ServerExceptionHandler := TErrorLogger.Create
end;
...
procedure TErrorLogger.OnException(const ServerClass,
  ExceptionClass, ErrorMessage: WideString; ExceptAddr:
  Integer; const ErrorIID, ProgID: WideString; var Handled:
  Integer; var Result: HResult);
var
  Log: TextFile;
const
  LogName = 'C:\DelphiLog.Txt';
begin
  AssignFile(Log, LogName);
  if FileExists(LogName) then
    Append(Log)
  else
    Rewrite(Log);
  try
    WriteLn(Log, Format('Class %s (ProgId: %s) raised an ‘+
      ‘%s exception at $%x: %s', [ServerClass, ProgID,
      ExceptionClass, ExceptAddr, ErrorMessage]))
  finally
    CloseFile(Log)
  end;
  //Could kill off exception like this, but not in this case
  //Handled := Integer(True)
end;
```

➤ *Listing 5*

execute. This can then perform whatever logging actions are necessary, and optionally kill off the error (so it doesn't propagate through to the client application).

The `ServerExceptionHandler` property can be particularly useful for standalone or MTS COM objects used in active server pages (directly supported in Delphi 5 and later). The `OnException` method could, for example, write exception information in HTML for the web browser to display.

The sample project group ComServerErrorLogger.bpg (on this month's disk) contains a COM server project and a COM client project that provide a simple demonstration of this. The server defines a `TErrorLogger` class which implements the `IServerExceptionHandler` interface. An instance of this is constructed when the `TErrorTrappingTest` COM object is created (within the overridden `Initialize` method), and assigned to the `ServerExceptionHandler` property. The COM class has five methods, each of which generates a different exception.

The `TErrorLogger`'s `OnException` method writes various pieces of information out to a specified text file so they can be reviewed later. In this case, the code does not kill off the exception, and so the client still reports it, but this could easily be changed. The `Handled` parameter indicates whether the exception should be propagated back to the client and defaults to 0 (not handled, so send to client). Setting the parameter to non-zero would kill off the exception. Listing 5 has the important bits of the code.

If you test out this COM client and server, you can prove the point made earlier about hardware exceptions. Among the exceptions generated by the server are a divide by zero and an Access Violation. In both these cases, the client does raise a representative exception but all it says is *Unexpected Failure*.

What Next?

So now all the mysteries of `safecall` have been laid bare, why would we want to use it outside of a COM application? Well, maybe you wouldn't. Maybe just knowing what `safecall` means will be quite enough.

But then again, maybe you have some object(s) whose more interesting methods have a propensity for generating exceptions and you want to keep track of which ones are raised. In the case of a COM

➤ *Listing 6*

```
uses
  BDE;
var
  EClass: ExceptClass = nil;
  EMsg: String;
procedure SafeCallError(ErrorCode: Integer; ErrorAddr:
  Pointer);
begin
  //Must raise exception, but if SafeCallException handled
  //the problem, we'll raise a "silent" exception
  if EClass <> nil then
    raise EClass.Create(EMsg)
  else
    SysUtils.Abort
end;
procedure LogError(ExceptObject: Exception;
  ExceptAddr: Pointer);
var
  Log: TextFile;
const
  LogName = 'C:\DelphiLog.Txt';
begin
  AssignFile(Log, LogName);
  if FileExists(LogName) then
    Append(Log)
  else
    Rewrite(Log);
  try
    WriteLn(Log, Format('%s'#9'%s'#9'$%p'#9'%s',
      [ExceptObject.ClassName, ExceptObject.Message,
       ExceptAddr, Application.ExeName]))
  finally
    CloseFile(Log)
  end;
end;
function TForm1.SafeCallException(ExceptObject: TObject;
  ExceptAddr: Pointer): HResult;
var
  Handled: Boolean;
begin
  Result := E_UNEXPECTED;
  Handled := False;
  if ExceptObject is EDBEngineError then begin
    Handled := True;
    case EDBEngineError(ExceptObject).Errors[0].ErrorCode of
      DBIERR_DETAILRECORDSEXIST :
        ShowMessage(
          'Delete/change detail records first...');
      DBIERR_KEYVIOL:
        ShowMessage('Already used that one')
    else
      Handled := False
    end
  end;
  LogError(Exception(ExceptObject), ExceptAddr);
  if Handled then
    EClass := nil
  else begin
    EClass := ExceptClass(ExceptObject.ClassType);
    EMsg := Exception(ExceptObject).Message
  end
end;
...
initialization
  SafeCallErrorProc := @SafeCallError;
finalization
  SafeCallErrorProc := nil;
end.
```

object, the previous section described the perfect way of logging exceptions.

For non-COM objects, you could write exception handlers in all the methods in question and log any caught exceptions, but this has a tendency to involve repeatedly typing the same constructs and calls. So, instead, you could mark the methods in question as `safecall`, override `SafeCallException` (adding into it any exception handling code you need) and assign some suitable routine to `SafeCallErrorProc`.

The TestSafeCall.Dpr sample project on the disk tests the theory out. The form has a `safecall` method which is imaginatively called `DoSomething`, which in this case simply calls the `Next` method of a `TTable` component. Various buttons entice `DoSomething` into failing, by setting the table up in various ways before calling it. One shuts the table first, causing an `EDatabaseError`. The other two cause `EDBEngineError` exceptions relating to a key violation and a referential integrity problem (detail records exist, so we cannot modify or delete the record).

The form overrides its `SafeCallException` method primarily to log the exception, writing out to a text file the exception class name and message, the address at which the exception was raised, and finally the application name. Additionally, in this case, the code also picks out a couple of specific exceptions that might come about, and deals with them in a specific way (by displaying its own message boxes).

To keep the exception logic of the application roughly normal, `SafeCallException` notes down the message and exception class type that was picked up. The `safecall` error handler then recreates the exception when it gets invoked. Any exceptions that `SafeCallException` decides to handle must cause the `safecall` error handler not to produce an exception dialog for consistency (it calls `Abort`). Listing 6 shows some pertinent code excerpts.

Summary

From reading this article, you should have learnt that the `safecall` reserved word causes a COM method to use the standard Win32 calling convention (`stdcall`) and correctly handle exception propagation from COM server to client using standard COM error reporting mechanisms. Whilst you can shoe-horn `safecall` into non-COM applications, its primary purpose is to ease the development of well-behaved COM applications with good error handling abilities.

Acknowledgements

Thanks go to Inprise R&D's Danny Thorpe for input and advice on this `safecall` description.

Brian Long is an independent consultant and trainer. You can reach him at brian@blong.com

Copyright ©1999 Brian Long. All rights reserved.