Beating the System: Code Generation Part 2

by Dave Jewell

As promised last month, we're going to continue our exploration of Delphi code generation with a look at `try-finally` blocks, exception handling and nested procedure calls. Having got a feel for the amount of code generated by these different language features, you'll be in a position to write leaner, more efficient apps.

Try-Finally Blocks

What's the magic that causes the `finally` clause of a `try-finally` block to be executed no matter what? To answer that question, feast your eyes on Listing 1 and all will be revealed. In order to make things easier to understand, I've highlighted the most interesting parts in red. The highlighted parts of the code correspond to the control structures that the compiler inserts into your code in order to implement the `try-finally` functionality.

Right at the top of the listing, you can see a six-byte block which I've referred to as a descriptor block. This is a special case of the more general descriptor block which the compiler uses to implement `try-except` blocks, as we shall see later. Whenever you use `try-finally` in your code, the first two bytes of the descriptor block are always zero and the other four bytes represent a far pointer into your code. This points to the beginning of the `finally` clause. In this particular example, the `finally` clause consists of one simple call to `FreeMem`. Looking at the descriptor block, we can see that the pointer value is `$4052:019A` which does indeed point to the call to `FreeMem`.

When the `try-finally` block is entered (location `$176`), the compiler generates code which pushes a far pointer to the descriptor block onto the stack. The current procedure's stack-frame pointer (`BP`) is also pushed onto the stack. Finally, (ha-ha!) a special global variable `ExceptList` is pushed onto the stack and `ExceptList` is set to the value of the current stack pointer. The `ExceptList` variable is shown as being at location `$9A6` in Listing 1.

Having done all that, the housekeeping code has effectively saved the 'context' of the running procedure so that its `finally` block can be called if and when an exception occurs. If an exception does take place, the `finally` code gets called from deep inside the run-time library. I won't explain the operation of this code here, because it's very lengthy, complex, and the code is already described in the file SOURCE\RTL\SYS\EXCP.ASM. Suffice to say that if an exception occurs, the code at location `$19A` (the start of our `finally` clause) is called as a far procedure from the library code.

This immediately presents a problem: how does the run-time

Listing 1

```
procedure TForm1.Button1Click(Sender: TObject);
var
  APointer: Pointer;
  AnInteger, ADividend: Integer;
begin
  ADividend := 0;
  GetMem (APointer, 1024);          { allocate 1K of memory }
  try
    AnInteger := 10 div ADividend; { this generates an error }
  finally
    FreeMem(APointer, 1024);  { execution resumes here, despite error }
  end;
end;

014F  00 00 9A 01 52 40                      ; descriptor block for try-finally

0155  55            push  bp
0156  89 E5         mov   bp,sp
0158  B8 0008       mov   ax,8                ; stack probe
015B  9A 67122686   call  far ptr sub_0067
0160  83 EC 08      sub   sp,8
0163  31 C0         xor   ax,ax               ; ADividend := 0;
0165  89 46 F8      mov   [bp-8],ax
0168  68 0400       push  400h                ; GetMem
016B  9A 671223C4   call  GetMem
0170  89 46 FC      mov   [bp-4],ax           ; result in APointer
0173  89 56 FE      mov   [bp-2],dx
0176  B8 014F       mov   ax,14Fh             ; try . . .
0179  0E            push  cs
017A  50            push  ax
017B  55            push  bp
017C  FF 36 09A6    push  word ptr ds:[9A6h]
0180  89 26 09A6    mov   word ptr ds:[9A6h],sp
0184  B8 000A       mov   ax,0Ah              ; AnInteger := 10 div ADividend
0187  99            cwd
0188  F7 7E F8      idiv  word ptr [bp-8]
018B  89 46 FA      mov   [bp-6],ax
018E  8F 06 09A6    pop   word ptr ds:[9A6h]
0192  83 C4 06      add   sp,6
0195  B8 01A9       mov   ax,1A9h v
0198  0E            push  cs
0199  50            push  ax
019A  FF 76 FE      push  word ptr [bp-2]     ; finally block
019D  FF 76 FC      push  word ptr [bp-4]
01A0  68 0400       push  400h
01A3  9A 671223DE   call  FreeMem
01A8  CB            retf
01A9  C9            leave
01AA  CA 0008       retf  8
```

library code call a chunk of another routine as if it were a separate, stand-alone procedure? This is where things get a bit sneaky. As you'll see from the code listing, the compiler inserts a far return (RETF) instruction at location $1A8, specifically so that the run-time library can make a far call to the finally clause in the event of an exception. But this means that if an exception doesn't occur deeply bad things are going to happen when this RETF instruction is encountered! In order to get around this, the cleanup code at location $18E onwards must not only restore the state of the ExceptList and pop the three words that were originally pushed onto the stack, but it also has to push the far address of the exit point from the try-finally block. Thus, when the bogus RETF statement gets executed, the processor restarts execution at location $1A9. You'll notice, incidentally, that the eight bytes of parameters consumed by this routine aren't popped off the stack until the 'real' return address is encountered. This is important because the exception-handling code in the run-time library wants a simple far routine that it can call. It has no knowledge of the number of bytes of parameters used by the target code.

This is a cunning mechanism and, because it's stack based, it means that try-finally and try-except blocks are inherently 'nestable'. As the code winds its way out of a deeply nested chunk of code, the saved context information is successively popped off the stack until the status quo has been restored.

Try-Except Blocks

As mentioned above, try-finally blocks are a specific case of the more general try-except mechanism. To see how this works, take a look at Listing 2. Here, you can see a simple exception handler designed to catch EConvertError exceptions which might be thrown while executing the StrToInt routine. You've almost certainly written code like this yourself, to cater for those times when an end-user needs to enter some numeric value. In order to keep the code listing as short as possible, this particular error handler just calls MessageBeep, but you'd typically call MessageDlg or something similar.

Notice that this time round, the descriptor block has grown to ten bytes. Why's this? The reason is that, generally speaking, a descriptor block consists of a 16-bit count, followed by a list of exceptions and the addresses to which the code should jump if one of those exceptions is triggered. The try-finally descriptor is the degenerate case: the count word is zero and the only pointer is the address of the finally clause. In the case of Listing 2, we've specified only one exception, EConvertError, and therefore the count word is set to one. This is followed by a 32-bit pointer to the RTTI (run-time type information) for EConvertError, and then by a far pointer to the except clause that you've written: in this case at address $0198.

If you'd written a 'blind' exception handler (in other words, if you hadn't specified an exception type), then the RTTI pointer in the descriptor block would have pointed at the type information for Exception itself, which of course is the base class for all exception types. This would have meant that the exception handler would have been triggered whatever type of exception occurred. As you've probably guessed, the run-time library's exception handling code steps through the list of exception types described in the descriptor block trying to find the best match with the exception that's actually been raised. If the descriptor block only contains an entry for Exception, then a match will always take place.

Just as with the try-finally example, the procedure starts off by pushing a far pointer to the descriptor block together with the routine's stack frame. The current stack pointer is added to the exception list as previously. However, when the 'normal' part of the try-except block terminates, it simply fixes up the stack, unwinds the exception list and then branches around the exception handling

➤ *Listing 2*

```
procedure TForm1.Button1Click(Sender: TObject);
var count: Integer;
begin
  try
    count := StrToInt (Edit1.Text);
  except
    on EConvertError do MessageBeep (0);
  end
end;

0156  01 00 AE 02 6B 68 98 01 6B 40   ; descriptor block for try-except
0160  C8 0102 00       enter  102h,0
0164  B8 0156          mov   ax,156h
0167  0E               push  cs
0168  50               push  ax
0169  55               push  bp
016A  FF 36 09E0       push  ExceptList
016E  89 26 09E0       mov   ExceptList,sp
0172 8D BE FEFE        lea   di,[bp-102h]
0176  16               push  ss
0177  57               push  di
0178  C4 7E 06         les   di,dword ptr [bp+6]
017B  26: C4 BD 017C   les   di,dword ptr es:data_0047e[di]
0180  06               push  es
0181  57               push  di
0182  9A 4F5B1BD4      call  TControl.GetText
0187  9A 686B06EA      call  StrToInt
018C  89 46 FE         mov   [bp-2],ax
018F  8F 06 09E0       pop   ExceptList
0193  83 C4 06         add   sp,6
0196  EB 0C            jmp   01A4     ; no exception, so skip exception block
0198  6A 00            push  0        ; start of the exception block
019A  9A FFFF0006      call  MessageBeep
019F  9A 686B2901      call  DoneExcept
01A4  C9               leave
01A5  CA 0008          retf  8
```

block. There's no need to play funny tricks with the stack because `finally` clauses are called from the run-time library, whereas `except` clauses are jumped into from the run-time library.

Finally, while still on the subject of exception handling, take a look at the code in Listing 3. This demonstrates a typical use of the `Raise` statement in conjunction with a call to `FmtLoadStr`. I discussed the 'Format-series' routines last month. The important thing here is the way in which an exception object is generated, passing a pointer to the RTTI information to the class and calling the constructor for that routine. The object address, returned from the constructor, is then passed to the `RaiseExcept` routine in the run-time library. Again, when this is combined with the open array usage of `FmtLoadStr`, a surprising amount of code gets generated by a single line of Object Pascal. If you routinely use this type of construction inside your application, you'll get a considerable reduction in code size by localising the exception creation and raising mechanism into a single procedure.

Nested Procedures

Nested procedures are a useful (and in my experience under-used) feature of Pascal. They allow you to chop up a large routine into manageable pieces whilst keeping it clear who can call what! At the same time, nested procedures can substantially reduce code size by effectively allowing you to common-up chunks of code which would otherwise have to be duplicated.

However, if used injudiciously, nested procedures can increase code size and make a program less efficient because of the way in which a nested procedure gets access to the parameters and variables of its parent. As you'll no doubt appreciate, methods of a class have a hidden 32-bit parameter which is the `Self` pointer: a pointer to the current dynamically allocated instance of the class. In the same way, local procedures also have a hidden parameter, but in this case its a pointer to the stack frame of the parent.

Listing 4 should make this clearer. The somewhat contrived (and totally useless) `FormCreate` routine shown here does nothing except call a local procedure called `SwapXY` which swaps a couple of

➤ Listing 3

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  raise Exception.Create (FmtLoadStr (61440, ['Screwdriver', 'Squirrel']));
end;

0145  C8 0110 00     enter  110h,0
0149  8D BE FEF0     lea  di,cs:[0FEF0h][bp]
014D  16             push  ss
014E  57             push  di
014F  68 F000        push  0F000h
0152  B8 0130        mov  ax,130h      ; 'Screwdriver'
0155  8C CA          mov  dx,cs
0157  89 46 F0       mov  [bp-10h],ax
015A  89 56 F2       mov  [bp-0Eh],dx
015D  C6 46 F4 04    mov  byte  ptr [bp-0Ch],4
0161  B8 013C        mov  ax,13Ch      ; 'Squirrel'
0164  8C CA          mov  dx,cs
0166  89 46 F8       mov  [bp-8],ax
0169  89 56 FA       mov  [bp-6],dx
016C  C6 46 FC 04    mov  byte  ptr [bp-4],4
0170  8D 7E F0       lea  di,[bp-10h]
0173  16             push  ss
0174  57             push  di
0175  6A 01          push  1
0177  9A 660B070F    call  FmtLoadStr
017C  B0 01          mov  al,1
017E  50             push  ax
017F  B8 0022        mov  ax,22h        ; push RTTI info for Exception
0182  BA 660B        mov  dx,660Bh
0185  52             push  dx
0186  50             push  ax
0187  9A 660B119B    call  Exception.Create    ; create exception object
018C  52             push  dx
018D  50             push  ax
018E  9A 660B2815    call  RaiseExcept    ; and raise it
0193  C9             leave
0194  CA 0008        retf  8
```

➤ Listing 4

```
procedure TForm1.FormCreate(Sender: TObject);
var
  x, y: Integer;
  procedure SwapXY;
  var
    temp: Integer;
  begin
    temp := x;
    x := y;
    y := temp;
  end;
begin
  SwapXY;
end;
end.

SwapXY:
0130  C8 0002 00      enter  2,0
0134  8B 7E 04        mov  di,[bp+4]
0137  36: 8B 45 FE    mov  ax,ss:[di-2]
013B  89 46 FE        mov  [bp-2],ax
013E  36: 8B 45 FC    mov  ax,ss:[di-4]
0142  36: 89 45 FE    mov  ss:[di-2],ax
0146  8B 46 FE        mov  ax,[bp-2]
0149  36: 89 45 FC    mov  ss:[di-4],ax
014D  C9              leave
014E  C2 0002         retn  2
0151  C8 0004 00      enter  4,0
0155  55              push  bp
0156  E8 FFD7         call  SwapXY
0159  C9              leave
015A  CA 0008         retf  8
```

Slimming Techniques For Delphi Developers

➤ Get-Set (or Read-Write) Routines

Design your Get-Set routines to be small and fast: in many cases the *Get* routine should just map onto the read of a specific private variable. Where you make extensive use of Get-Set routines in the VCL framework, be mindful of the code that's being generated. If you have a case statement with multiple clauses which do something like this:

```
Bitmap.Canvas.Pen.Color := <whatever>
```

then just set up a local `TColor` variable in each branch of the case statement and make the actual assignment to the pen once only.

➤ With Statements

Use `with` statements to make life easier for the compiler and reduce the amount of typing you've got to do. However, bear in mind that for very simple routines a `with` statement can actually increase the size of the code generated.

➤ Open Arrays

A great idea, but use with caution. If you make heavy use of open arrays to convert integers into strings and vice versa, you'll get the job done a lot more efficiently with `StrToInt` and `IntToStr`. Also, don't use fancy calls to `Format` when you want to concatenate a few strings together: the standard string concatenation routines are a lot more economical in terms of code size.

➤ Try-Finally And Try-Except Blocks

Again, a great idea, but don't bother if you're protecting code that can't possibly throw an exception anyway! If the code you're protecting can throw exceptions, then be sure that your exception handler can 'catch' the type of exception that's being thrown, otherwise you're just needlessly bloating your code.

➤ Raising Exceptions

If you raise exceptions at many places in your code, consider doing the `raise` in just one routine, passing it the string resource ID (for example) of an error message to differentiate between error conditions from the user's point of view. A simple technique like this can save a lot of code.

➤ Nested Procedures

Sometimes a Pascal programmer will deliberately duplicate a lot of code in different parts of the same routine specifically to avoid the humiliation of using a `goto` statement! With nested procedures, you can avoid the duplication and retain your street-cred! But if you have nested procedures inside a very time-critical chunk of code such as a numerical analysis routine, then consider passing parameters to local procedures instead of relying on the implicit scoping rules. This is generally a lot faster than going through the frame pointer, especially if you nest more than one level.

variables in the parent procedure and then exits gracefully.

Notice that in the call to `SwapXY`, the `BP` parameter gets pushed onto the stack even though the routine isn't declared as taking any parameters: this is the undocumented frame pointer I mentioned. By using positive offsets from this frame pointer, the nested subroutine can access the parameters (if any) of its parent routine. By using negative offsets, it can get to the local variables.

Thus, in this simple example, the `SwapXY` routine loads up the `DI` register with the frame pointer and then uses stack-relative addressing to access the `x` and `y` local variables. In this case, the routine is so trivially simple that the compiler can do the whole thing without having to reload the `DI` register, but in a more real-life example, the code would typically have to set up `DI` several times within the procedure. This makes the code slower and larger than it would otherwise be, but against that you have to balance the potential reduction in code size that can be obtained by de-duplicating code in a nested procedure. This is why programming is an art, not an exact science!

Conclusions

From the last couple of months worth of deliberations I've come up with a set of guidelines for putting your code on a diet – and making it run faster too. See the box at left.

Next time round, by popular demand, I'll look at *more* ways in which you can interact with the Windows 95/NT Explorer, using the various COM interfaces that Microsoft provide for the purpose.

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows and DOS work. He is the author of *Instant Delphi Programming* published by Wrox Press. Contact Dave as DaveJewell @msn.com or DSJewell@aol.com or as 102354,1572 on CompuServe