YAST:
Yet Another Stack Tracer!

by Hallvard Vassbotn

Have you ever had any really hard-to-find bugs in your code? If not, you can skip this article, otherwise you'd better keep on reading!

A stack-tracer is a utility that lets you unwind the stack to find the entry points that led up to the current state of the program. This is useful when you detect an error-condition in a very general routine and you want to display information about how you got there. For instance, it could be very handy to combine the stack tracer which is described here (and included on this month's disk, of course) with an exit procedure that reports run-time errors.

Background

The stack is used to store local variables, parameters and return addresses. For instance, after running the code in Listing 1 the stack would look as shown in Figure 1.

The addresses shown on the left are offsets relative to the BP register. We simply use SS:BP as a pointer to find the first stack frame. Then we can follow a linked list of stack frames by reading the saved BP value at offset 0.

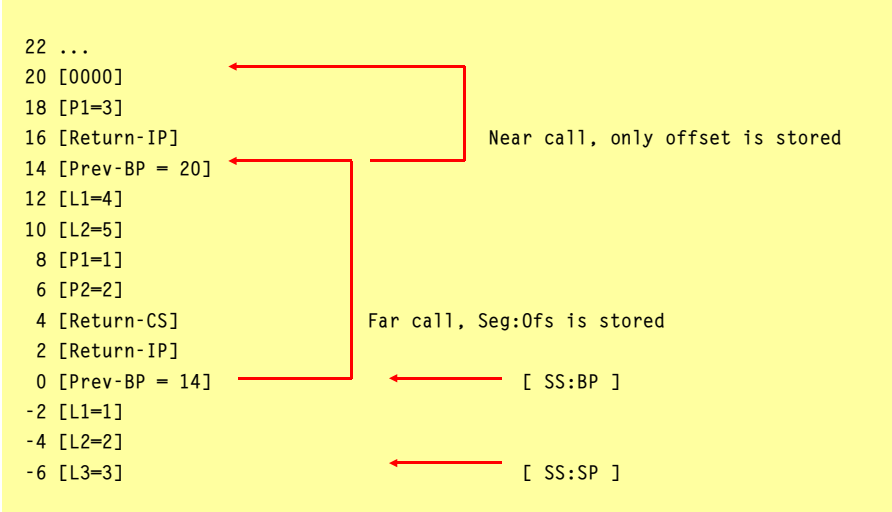
The values stored between the current BP and the previous BP comprise one stack frame. The first value will be CS:IP, or only IP if it was a near call, of the caller, followed by any parameters in reverse order, followed by any local parameters of the caller.

The tricky part is to decide whether the call made was near or far, and thus whether a complete segment:offset pair or only an offset is stored on the stack. The code itself 'knows' this by a hard-coded near or far return at the end of the routine.

So, we *could* disassemble the code until we reached a return instruction, to find out which one was used. However, this would be difficult to implement (I don't have any disassembler source code readily available...!) and could be a bit slow to execute.

I chose a simpler but less reliable method. I always assume that the call is far. Then I check the assumption by validating that the segment:offset pair on the stack is a valid pointer to a code segment. This can only be done in protected mode and Windows, so in real mode I always assume far calls.

Even in protected and Windows mode, this approach might fail, reporting a far call when it is really a near call. This will happen if the word following the offset happens to be a valid code segment selector. To avoid this problem, compile all your code with forced far calls on (use the compiler directive {$F+}) or you can implement the disassembler method that I mentioned above...



➤ *Figure 1*

```
procedure Level2(P1, P2: word);
var
  L1: word;
  L2: word;
  L3: word;
begin
  L1 := 1;
  L2 := 2;
  L3 := 3;
  { Stop here! }
end;
procedure Level1(P1: word); near;
var
  L1: word;
  L2: word;
begin
  L1 := 4;
  L2 := 5;
  Level2(1, 2);
end;
begin
  Level1(3);
end.
```

➤ *Listing 1*

Implementation

Listing 2 shows the unit, YAST.PAS, which implements a simple but effective stack trace utility.

I have compiled and tested it with Delphi 1.0 and Borland Pascal 7.0 in real mode, protected mode and Windows.

The stack tracer simply gathers information about each stack frame (one for each call-level in the stack) and uses a user defined callback function to report it back to the user. There is a single interfaced function, which is called TraceStack, and is defined as follows:

```
procedure TraceStack(
  ReportStackFrame :
  TReportStackFrame;
  PrivateData: Pointer);
```

ReportStackFrame is the user defined callback function that will be called with information from the stack. It has to be of the type:

```
TReportStackFrame =
  function(var StackInfo :
    TStackInfo; PrivateData :
    Pointer): boolean;
```

This means that it has to be declared as:

```
function ReportStackFrame(
  var StackInfo: TStackInfo;
  PrivateData: Pointer):
  boolean; far;
```

You can choose your own function name. The other parameter for TraceStack is PrivateData: a general user-defined pointer. This is simply sent to your callback function and gives the caller of TraceStack a means of communication with the callback routine. Without this, we would have to use global variables. This pointer can be anything from nil to a pointer to a complex object (eg a form) that you can use in the callback routine to output the stack information.

For each stack frame, the callback function ReportStackFrame will be called with a TStackInfo record (see Listing 2). Note the use of a variant to alias the pointer fields with corresponding offset and segment fields. This will save us from some typecasting and makes the code clearer. Table 1 explains the TStackInfo record.

Example Usage

An example project using the stack tracing unit is included on the disk as TTDELPHI.DPR (plus there are Borland Pascal examples too). Perusing this code should help you to implement stack tracing in your own projects.

Usually, you will want to display the logical segment number (ReturnLog) and offset (ReturnOfs) of the return address. Then you might also display some sort of stack dump to show the value of parameters and local variables (use ParamPtr and ParamSize).

The output shown in Figure 2 was created by running the Delphi test program on my machine. This might look like a lot of gibberish, but by examining the stack trace carefully and looking at the source code we can wring out a lot of information from this hex output.

If we take the two first lines for a closer examination and combine this with a MAP file we will find:

```
F 0001:002F =
 42BD,2677,00BD,26FF,000D,000D,
```

This is a far call from 0001:002F which is line 22 in the TRACETST unit called by the example project (the Level3a procedure):

```
TraceStack(
  ExampleReportStackFrame,
  @FirstFlag);
```

Note that the parameters on the stack will match the source code parameters if we read the source from right to left.

The first two words on the stack (42BD,2677) give us the PrivateData parameter, which is a pointer to the FirstFlag boolean variable in

CallersBP	Value of the callers BP (if you hadn't guessed!). It marks the start of a new stack frame. This is an internal value and is usually not referenced from the callback routine.
ParamSize	The number of elements referenced by the ParamPtr array.
DumpSize	The number of elements referenced by the DumpPtr array.
IsFar	A flag indicating if the call was a far or a near call.
ReturnLog	The logical segment number of the return address. This is the address you will see in the MAP file. For protected and Windows mode this will differ from the actual selector value.
CallerAdr	The far address taken directly from the stack. It will not be valid if it is a near call. Normally not used directly.
ReturnAdr	The segment:offset pair for the return address.
DumpPtr	A pointer into the stack to an array of raw bytes. This also includes the raw caller's address.
ParamPtr	A pointer into the stack to an array of words. This does not include the callers address and is usually better suited to show to the user.

➤ *Table 1: Details of the* TStackInfo *record*

```
Starting stack trace:
CS=26FF, DS=2677, SS=2677
F 0001:002F = 42BD,2677,00BD,26FF,000D,000D,
F 0001:0053 = 0007,0006,0009,
F 0001:0077 = 0005,0004,0005,
F 0001:008E = 0003,0002,
N 0001:00A1 = 0001,
F 0001:00C1 =
```

➤ *Figure 2*

the stack segment. Next are the offset and segment of the callback routine (00BD,26FF). We can verify this by seeing that SS=2677 and CS=26FF.

Now we have found all the parameters to the TraceStack routine, so the next data displayed will be the value of the local variables in the Level3a procedure. These must be matched up with the source code by reading the source from bottom to top.

So, the first 000D value is actually the FirstFlag boolean variable. But it only occupies the last byte of this word, so the value is 00. The other byte (with the value 0D) is simply there to preserve stack alignment and has an undefined value.

The last 000D value is the C local word variable, which makes sense, because it should be the sum of 6 and 7 which is 13 in decimal or 0D in hex.

To see what parameters Level3a was called with, we have to look at the next stack frame:

```
F 0001:0053 = 0007,0006,0009,
```

➤ *Listing 2: The YAST unit*

```
unit YAST;
interface
type
  PBytes = ^TBytes;
  TBytes =
    array[0..(High(Word)-$f) div sizeof(byte)] of byte;
  PWords = ^TWords;
  TWords =
    array[0..(High(Word)-$f) div sizeof(word)] of word;
  TStackInfo = record
    CallersBP : word;
    DumpSize  : word;
    ParamSize : word;
    IsFar     : boolean;
    ReturnLog : word;
    case integer of
    1: (CallerAdr : pointer;
        ReturnAdr : pointer;
        DumpPtr   : PBytes;
        ParamPtr  : PWords);
    2: (CallerOfs : word;
        CallerSeg : word;
        ReturnOfs : word;
        ReturnSeg : word;
        DumpOfs   : word;
        DumpSeg   : word;
        ParamOfs  : word;
        ParamSeg  : word)
  end;
  TReportStackFrame = function(var StackInfo:
    TStackInfo; PrivateData: Pointer): boolean;
procedure TraceStack(ReportStackFrame: TReportStackFrame;
  PrivateData: Pointer);

implementation
uses
{$IFDEF WINDOWS}
  {$IFDEF VER80}
    WinProcs;
  {$ELSE}
    Win31;
  {$ENDIF}
{$ELSE}
  ValidPtr;
{$ENDIF}
type
  PtrRec = record
    Ofs, Seg : Word;
  end;
  TFarStackFrame = record
    CallersBP : word;
    case integer of
    1: (CallerAdr : pointer);
    2: (CallerOfs : word;
        CallerSeg : word)
  end;
  TNearStackFrame = record
    CallersBP : word;
    CallerOfs : word;
  end;
  PStackFrame = ^TStackFrame;
  TStackFrame = TFarStackFrame;

function GetSSBPPtr: pointer; inline
($8C/$D2    { MOV DX, SS }
/$89/$E8);  { MOV AX, BP }

function LogSeg(Seg: word): word;
begin
{$IFDEF MSDOS}
  LogSeg := Seg;
{$ELSE}
  if Seg <> 0 then
    LogSeg := Word(Ptr(Seg, 0)^)
  else
    LogSeg := Seg;
{$ENDIF}
end;

procedure CorrectBP(var BP: word);
{ Handle Windows stack frames (ie Inc BP in far prolog code) }
begin
  if Odd(BP) then Dec(BP);
end;

function IsFarCode(Addr: pointer): boolean;
begin
{$IFDEF WINDOWS}
  IsFarCode := not IsBadCodePtr(Addr);
{$ELSE}
  IsFarCode := ValidCodePointer(Addr, 1);
{$ENDIF}
end;

function NextStackFrame(var StackFrame: PStackFrame;
  var StackInfo : TStackInfo): boolean;
var More: boolean;
begin
  More := (StackFrame^.CallersBP <> 0) and
    (StackFrame^.CallerAdr <> nil);
  if More then with StackInfo do begin
      CallersBP  := StackFrame^.CallersBP;
      CorrectBP(CallersBP);
      CallerAdr  := StackFrame^.CallerAdr;
      DumpPtr    := Pointer(StackFrame);
      DumpSize   := (CallersBP - PtrRec(StackFrame).Ofs);
      ParamPtr   := Pointer(DumpPtr);
      ParamSize  := DumpSize div 2;
      IsFar      := IsFarCode(CallerAdr);
      if IsFar then begin
        ReturnAdr := CallerAdr;
        Dec(ParamSize, SizeOf(TFarStackFrame) div 2);
        Inc(ParamOfs , SizeOf(TFarStackFrame));
      end else begin
        ReturnOfs := CallerOfs;
        Dec(ParamSize, SizeOf(TNearStackFrame) div 2);
        Inc(ParamOfs , SizeOf(TNearStackFrame));
      end;
      ReturnLog := LogSeg(ReturnSeg);
      PtrRec(StackFrame).Ofs := StackFrame^.CallersBP;
      CorrectBP(PtrRec(StackFrame).Ofs);
    end;
  NextStackFrame := More;
end;

procedure TraceStack(ReportStackFrame:
  TReportStackFrame; PrivateData: Pointer);
var
  StackFrame : PStackFrame;
  StackInfo  : TStackInfo;
begin
  FillChar(StackInfo, SizeOf(StackInfo), 0);
  StackInfo.ReturnSeg := CSeg;
  StackFrame := GetSSBPPtr;
  while NextStackFrame(StackFrame, StackInfo) and
        ReportStackFrame(StackInfo, PrivateData) do
    {Loop};
end;

end.
```

This shows that `Level3a` was called as a far routine from `0001:0053` or line 30, which is the `Level3b` routine. The parameters to `Level3a` were `B=7` and `A=6`. In `Level3b` there was a local word variable with the value 9. This makes sense as the `C` variable should have the sum of 4 and 5.

Continuing like this we can probably find out a lot more why a certain error condition is met.

Possible Improvements

The logic to decide if a call is far or near can be made smarter by using disassembly to find the next return-instruction. There could be other ways of doing this as well. Note also that the code has not been tested extensively with `TCollection`-type callbacks, or callbacks from DLLs, etc.

To show more information about the state of the program, one could dump out selected global variables, dump a raw output of the data segment to match up with the globals shown in the MAP file, etc.

A nice improvement would be to detect if there is debug information in the EXE file and find the unit, line number, procedure and parameter names instead of only the raw data it displays today.

Conclusion

With a stack tracer tool such as the one I've presented here, you are better equipped to track down errors and bugs that would otherwise be very difficult to find. A real debugger (like Turbo Debugger) is of course better to use when you are in the development phase, but for error-reporting at user-sites, automatic logging with a stack trace facility could save your day (and possibly your contract!).

Hallvard Vassbotn lives and works in Norway can be contacted by email at hallvard@falcon.no