Under Construction: Creative Debugging Techniques

by Bob Swart

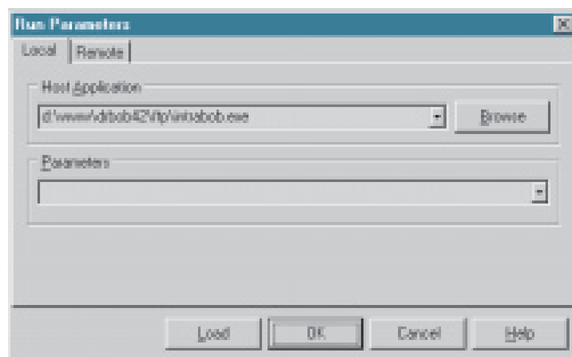
It happens to all of us, even to the best of us. We write programs, and these programs contain bugs. And more often than not, these bugs hide themselves very well. I need to fall back onto some creative debugging techniques more often than not to diagnose what's wrong and remedy it. In this article, I'll cover some basic and less basic Delphi debugging techniques, including something special called remote debugging.

Integrated Debugging

First things first, however. If we want to locate and fix a bug in our program, we need to compile and link it with debug information included. This is achieved by using two compiler options: `{$D+}` to include debug information that maps object code addresses to source code line numbers and `{$L+}` (for *local symbols*), meaning the names and types of all the local variables and constants in a unit. The `$L` directive is ignored if the compiler is in the `{$D-}` state, by the way. Setting these options increases the size of unit files, but does not affect the size or speed of the final executable program. If we compile a program with `{$L+}` then Delphi's integrated debugger gives us the ability to watch and modify local variables. In addition, we can see all the calls to procedures and functions in the call-stack window. The great thing about Delphi 4 is the fact that you can dock all the debug windows so you have them available while debugging. Also, while we currently need to do this again and again for every project we want to debug, Delphi 5 will feature 'named desktops' (demonstrated by Borland's John Kaster at a recent conference in The Netherlands), which will enable us to save a 'debug desktop' where all these views are open and docked by default! Quite a handy feature, which I will show in detail at a later time, of course.

To actually debug with the aforementioned Delphi 4 integrated debugger, we don't need to do anything else. If we want to use an external debugger, such as Turbo Debugger, we should also specify the `Include TD32 debug info` linker option, as well as click on the option to let the linker generate a detailed MAP file. Turbo Debugger is sold separately as part of Turbo Assembler 5.0, for those who are interested. We'll just focus on the integrated debugger here, which is included in every version and edition of Delphi, of course.

Before we start with actual debugging examples, there's one more feature I would like to mention, which is the `Run | Parameters` *'host application'* feature, which enables us to debug DLLs using the integrated debugger, by specifying the host application that will load the DLL and call methods from the DLL. This is one of the most useful enhancements made to the integrated debugger, since previously it was necessary to (buy and) use Turbo Debugger for this purpose.



➤ *Figure 1: IntraBob as the host application.*

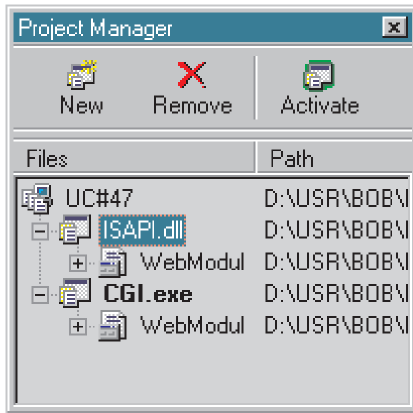
Distributed Applications

Assuming that we've all debugged a Delphi program using the integrated debugger before, I want to focus on situations you may not be familiar with yet: debugging distributed applications. We'll start easy with web server applications (ISAPI, using WebBroker), moving on to ActiveForms and N-tier applications. We'll need more than one active copy of Delphi to debug an N-tier application on a single machine. Finally, we'll cover the basics of remote debugging: debugging a remote application that runs on another machine! And while these may not be exactly the debugging tasks you're carrying out today, the day could well come sooner than you think when knowing how to debug distributed applications will be absolutely essential.

Debugging WebBroker ISAPI Applications

Let's start with debugging a web server application. This usually is a CGI/WinCGI or ISAPI/NSAPI application. It doesn't really matter, for the sake of this example, whether or not we're using the WebBroker components for these kinds of applications, or whether we're creating plain CGI or ISAPI applications from scratch. So let's assume we're using WebBroker, which is much easier anyway.

First, I want to focus on the most sophisticated of the two solutions: ISAPI DLLs. Of course, it is important to test and verify these DLLs, because they run inside the web server's memory space. When an ISAPI DLL crashes, it can take the entire web server (such as Microsoft Internet Information Server, IIS) down with it. And believe me, you won't make yourself very popular doing just that. Fortunately, Delphi has a built-in



➤ *Figure 2: Project Manager, CGI and ISAPI sharing the same Web Module.*

feature that helps us to debug ISAPI DLLs, based on the host application option in the aforementioned `Run | Parameters` dialog.

As stated on page 28-23 of the *Delphi 4 Developer's Guide*, in this dialog you can specify your web server as host application for the ISAPI DLL. However, this implies that you need to start (and stop) the entire web server every time you want to debug your ISAPI DLL. That seems a little bit like overkill. To answer this need, I extended the IntraBob CGI tester (first created way back in Issue 19 and downloadable free from my website at www.drbob42.com) to support ISAPI debugging as well. All you need to do is specify IntraBob as the host application instead of your web server.

Now, all we need is a starting HTML web page that contains a call to the ISAPI DLL. Once you run the application (that is, once you start to debug your ISAPI DLL), IntraBob will start up instead. And Delphi will wait until you click on the `submit` button on the HTML web page form, which is the trigger for IntraBob to load the ISAPI DLL and run it (that is, debug it from within the Delphi IDE, meaning that you can set breakpoints, and all the usual debugging tasks).

This has been my preferred technique for debugging web server applications for almost a year now. It's a shame that this method is not mentioned in the *Delphi 4 Developer's Guide*, but maybe that will change with Delphi 5.

Debugging CGI Applications

Now, to debug a CGI executable is actually quite hard, because you need a web server to start the CGI application (and we can only specify hosting applications in Delphi for DLLs, not for executables). It's much harder than it is to debug an ISAPI DLL. Fortunately, if we're using WebBroker technology, we can work on one web module and 'export' it as both a CGI executable (our initial target) and an ISAPI DLL (in order to debug).

The trick is to use the Project Manager and create a second (empty) web module project. Then remove the web module from it, and add a reference to the web module that's used by the first application. The full source code is on this month's disk for an example, and you can see the effect in Figure 2.

Apart from functioning as a host application for ISAPI DLLs, my IntraBob utility can also be used to test CGI and WinCGI applications.

IntraBob has a few limitations that you should keep in mind. First of all, there is no support for the `ReadClient` function, so any input data over 48Kb is clipped. Also, there is only limited support for `ServerSupportFunction`, which supports

```
HSE_REQ_SEND_URL_REDIRECT_RESP
```

and

```
HSE_REQ_SEND_RESPONSE_HEADER
```

but not

```
HSE_REQ_MAP_URL_TO_PATH
```

Finally, when you are using IntraBob as a host application, the ISAPI DLLs are loaded and unloaded directly, so you cannot reproduce true multi-threaded or cached behaviour.

On the bright side, I'm always working on improvements for IntraBob, and if you have any suggestions, don't hesitate to email me!

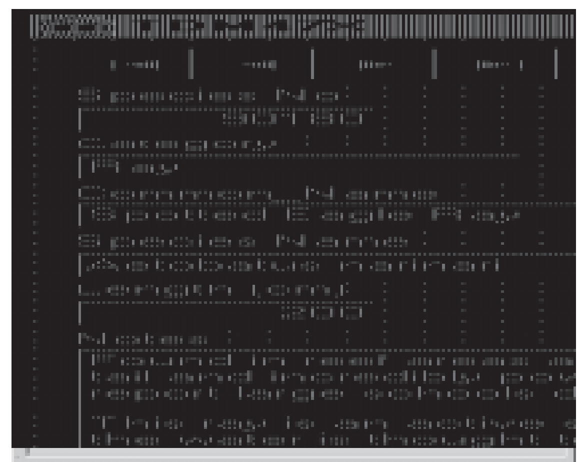
Debugging ActiveForms

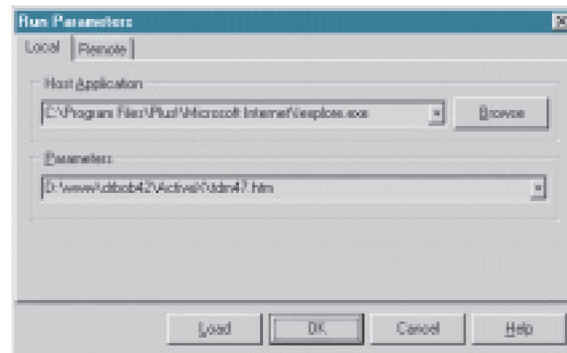
Another internet related technique is based on Microsoft's ActiveX and COM. Of course, I'm talking about ActiveForms, a client-side solution that works in Internet Explorer (or in Netscape too, with the right plug-in). ActiveForms are OCX files, which are actually plain DLLs with the OCX extension. This means that we can debug an OCX just like we can debug a DLL, by specifying a host application.

Let's create a sample ActiveForm (see Issue 22 for more details of how to do this) that uses some database tables (so we can eliminate the Borland Database Engine from it and turn it into a thin client using `TClientDataSet` in the next section). Create a new ActiveForm, drop a `Table` component on it, connect it to BIOLIFE.DB from the `DBDEMOS` alias, and add a number of data-aware controls to display all the fields (sneaky hint: add all the fields to the fields editor and drag-and-drop them on the form for a quick layout). Don't forget to put a `DBNavigator` on the ActiveForm, as we need some way to navigate through the table.

Assuming that we deployed the ActiveForm to the local directory d:\www\drbob42\ActiveX, then to debug the ActiveForm, we need to specify Internet Explorer 5 as the host application, and pass the name of the generated HTML file (that is, d:\www\drbob42\ActiveX\TDM47.htm) as the start-up argument, see Figure 4.

➤ *Figure 3: A 'fat' ActiveForm client (requires the BDE).*





➤ *Figure 4: Internet Explorer 5 as the host application, ActiveForm as the argument.*

This will almost work, but not quite, since Delphi's integrated debugger will only get triggered if Internet Explorer loads the very same (physical) version of the ActiveForm that Delphi created as output (ie the tdm47.ocx file in the current project directory). And Internet Explorer will not load that particular copy of our ActiveForm, but rather it will pick up the deployed version from the d:\www\drbob42\ActiveX\ directory and copy if to either the OCCACHE directory (for Windows 95) or the Downloaded Program Files directory (for Internet Explorer 5 on my Windows NT machine) and load the OCX from that particular directory.

In order for Delphi to know that Internet Explorer is loading the Delphi version of the ActiveForm, we should set the Output Directory to that location, using the `Project | Options` dialog.

This will make sure that after each rebuild or recompile the ActiveForm will be placed in the C:\winnt\downloaded program files\ directory. During the 'deployment', the ActiveForm will be copied to the D:\www\drbob42\ActiveX\ directory. When we run the application, Internet Explorer will pick up the file from the D:\www\drbob42\ActiveX\ directory, copy it to the C:\winnt\downloaded program files\ directory (overwriting the one we just created when we compiled the program, but with the same version), and load the ActiveForm from that directory. Now, Delphi will know that the ActiveForm (currently being debugged) is indeed the one loaded by Internet Explorer, so it will use the integrated debugger on it.

Debugging N-Tier Applications

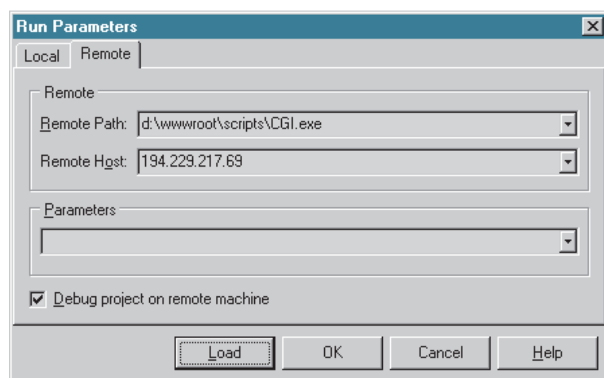
We can turn the ActiveForm into a thin client by removing the `Table` component, replacing it with a `TClientDataSet` component. Of course, we then need some sort of middleware application, for example a remote datamodule using DCOM as the communication protocol. The ActiveForm would then also need a `DCOMConnection` component to actually receive packages of records from the middleware server (see Issue 38 for more details of this).

In order to debug both the ActiveForm client (using Internet Explorer 5 as the host application) and the middleware data server, we actually need two running copies of Delphi: one with the ActiveForm project, specifying Internet Explorer 5 as the host application (see above), and the other running the middleware server.

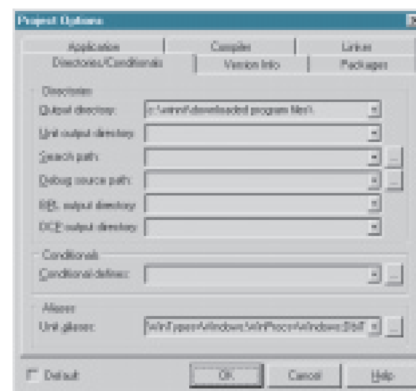
Remote Debugging

Sometimes, debugging distributed applications on one machine isn't possible, or doesn't yield the required result. In some cases, we really need to simulate the 'real world' situation as much as possible, meaning that we need to debug an application that runs on another machine. This is called remote debugging, and requires two machines that are connected via TCP/IP.

In order for remote debugging to work, we need to install the 'debug server' (which



➤ *Figure 6: Debugging CGI.exe project on a remote machine.*



➤ *Figure 5: Point Output Directory to the IE5 downloaded program files directory.*

is the file borrdbg.exe, which can be found in the `RDEBUG` directory on the Delphi CD-ROM) on the remote machine, that is, on the machine where the application runs that we want to debug with our 'local' machine, as if it were running on our local machine. On our local machine, we only need to run the Delphi IDE itself. The debug server will then control the remote program being debugged using a network connection to communicate with our local Delphi IDE.

Now, on the remote server, we first need to start the remote debugger using the `-listen` command line option (alternatively, it can be installed as an NT service, which means it always runs): `BORRDBG.EXE -listen`.

Once the debug server is running, we need to load the project source file, click on the `include remote debug symbols` option in the `Linker` page, and set the `Remote Path` in the `Run | Parameters` dialog to the executable name on the remote machine (note: for the

Remote Host entry either the machine name or IP address will be sufficient).

If we now run the application, we can set breakpoints in the local source of the project, but the project on the remote host is executed. Stepping through the source on the local machine also actually steps through the object code on the remote machine, which is quite an experience to see, and necessary from time to time.

Next Time

We've seen how to debug ISAPI and CGI web server applications, ActiveForms and N-tier applications. Lastly, we covered the basics of remote debugging.

Next time, I'll be showing you how to make Delphi go dynamic: dynamically creating components on a form and also dynamically assigning event handlers to these components. It'll be fun, I promise you, so *stay tuned...*

Bob Swart (aka Dr.Bob, visit www.drbob42.com, email him at drbob@chello.nl) is a technical consultant and webmaster using Delphi, JBuilder and C++Builder, and a freelance technical author. In his spare time, Bob likes to watch video tapes of Star Trek Voyager and Deep Space Nine with his 5-year-old son Erik Mark Pascal and his 2.5-year-old daughter Natasha Louise Delphine.