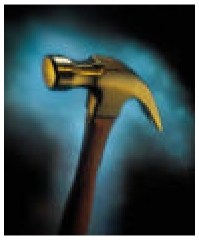


Oracle Forms Server Troubleshooting

An Oracle Technical White Paper

March 2000

INTRODUCTION

Figuring out the source of run-time errors in a form can be extremely confusing for novice Oracle Forms developers. Most of the work in resolving such issues consists of identifying the root and cause of the error; once that task is completed, correcting the code is often relatively simple. With the myriad of Forms triggers at different levels, it is sometimes difficult for even experienced programmers to determine which code statement is the culprit in causing runtime error messages or unexpected results.

There are various methods available to assist in this determination. One is simply a coding practice to immediately handle the failure of a Forms built-in. There are also many troubleshooting techniques, some of which are quick and easy to implement. The Forms debugger is a useful tool, and there are several diagnostic methods using log creation and interpretation. Most of the practices discussed apply to client-server applications, since a form must be able to run correctly in client-server mode before deploying it to the web. In addition, some techniques will be explained which apply only to web-deployed forms. It is advantageous to be acquainted with all of these methods, so that you can quickly pull out the appropriate troubleshooting tool for whatever situation you encounter.

CHECKING FOR FORM_SUCCESS AFTER CALLING A FORMS BUILT-IN

One reason that the source of problems can be difficult to isolate is that Forms by default does not abort execution of a trigger when a Forms built-in fails. For example, suppose you have a go_block statement toward the beginning of a lengthy trigger, but for some reason the navigation to the specified block fails. Forms will display an error message after the trigger completes, but in the meantime will continue to execute the long trigger for which the programmer assumed the current block would be the block specified in the go_block statement. Imagine the confusion and strange results that this could cause!

Because of this, it is good practice to check for form_success after calling each Forms built-in. If the built-in did not succeed, you should handle the situation and probably abort trigger execution by raising the form_trigger_failure exception. For additional information and sample code, see "form_success" in Forms on-line help.

TROUBLESHOOTING WITH "QUICK AND DIRTY" TECHNIQUES

Some troubleshooting techniques are quick and easy to implement and are suitable mostly for situations where there is already a good idea of where the problem lies. Most of these approaches lack complexity, but also lack the power of the more formal methods. Four techniques are discussed here: commenting out code, using the "message" built-in, using the "debug_messages=yes" runtime option, and using the "break" built-in.

COMMENTING OUT SUSPICIOUS LINES OF CODE

If you have a suspicion that certain lines of code are problematic, try commenting them out and testing the form again. Comments are non-executable portions of a program, denoted in PL/SQL by:

- lines prefaced by a double dash (--) or
- a section of code beginning with /* and ending with */

If you comment out an entire trigger or program unit, you will at least need to include the code statement:

```
null;
```

A disadvantage of commenting out lines of code is that you must remember to uncomment the lines once you either fix the problem or determine that those lines are not its cause. Keeping track of many comments in this manner can get confusing, so it is best to minimize your use of this technique, and to uncomment one section before commenting out another.

USING THE "MESSAGE" BUILT-IN

The "MESSAGE" built-in, strategically placed, will allow you to output to the screen any desired message. This message could include the name of the code that is executing or the values of any variables about which you would like information at that point in the code. Although the message string may be up to 200 characters long, the length of the message that can be displayed is affected by such factors as the message font and the limitations of the runtime window manager. In practicality, then, it is best to break up long messages into multiple smaller ones. Also, make sure the "Console Window" property of the form is set to a valid window; otherwise, messages will not appear.

To ensure that the message displays at the desired point in the code, add the "SYNCHRONIZE" command immediately after it.

```
Example:
message('The table contains '||to_char(empcount)||' employees');
synchronize;
message('Runtotals = '||:control.runtotals);  --Runtotals will be either
"YES" or "NO"
synchronize;
message('Beginning Runtotals program unit');
synchronize;
```

Rather than "SYNCHRONIZE", you could instead use "PAUSE", which not only synchronizes the display, but also pauses the program and forces the user to hit a key to continue. This would be advisable if you want to call attention to the message and the form's display at that point.

The above example illustrates the fact that the "MESSAGE" built-in displays a character message line, so you need to convert any numeric or date values to character. Also, variable values must remain outside the single quotes, and must be concatenated into the string using the || concatenation operator.

Using the "MESSAGE" built-in may cause problems with focus in applications containing more than one window. As with the commenting technique, you must remember to remove (or comment out) the message code once you have isolated the problem, as these messages can be disruptive and confusing to the user. Also, you may find that using the "MESSAGE" built-in makes the runtime error disappear! If this should happen, try using the "SYNCHRONIZE" built-in by itself, because this symptom indicates that the problem may be due to a lack of synchronization between the display and the internal state of the form.

USING THE "DEBUG_MESSAGES=YES" RUNTIME OPTION

A third "quick and dirty" technique for pinpointing code problems is the "debug_messages=yes" specification on the runform command line. This causes a message to automatically display to let you know each trigger as it executes. Once you encounter the runtime error, you will know the last trigger that fired and should be able to trace through the code from that point. For anyone that was around in the SQL*Forms 3.0 days, this was the default behavior when running a form in debug mode.

To use this option from the Form Builder, check the "debug_messages" option on the runtime tab after choosing Tools -> preferences from the menu. From the command line, simply add the "debug_messages=yes" option. If you are using an icon in Windows to start your form, add the option to the shortcut command. It would look something like this for Windows platforms:

```
c:\orant\bin\ifrun60.exe module=myform userid=scott/tiger
debug_messages=yes
```

On Unix:

```
f60runm module=myform userid=scott/tiger debug_messages=yes
```

The major disadvantage of using this technique is that you have to acknowledge every message as each trigger executes, which can be very annoying. Although it shows the triggers that fire, it does not show program units that execute, and there is no way to display the values of variables. It can also cause the same focus problems and program flow interruption that you see with the "MESSAGE" built-in. However, this method is useful if you have no idea which trigger is causing the problem; once you have narrowed down the scope, other troubleshooting techniques would be more appropriate.

USING THE "BREAK" BUILT-IN

While running a form in debug mode, the "BREAK" built-in invokes the Forms Debugger. "BREAK" can be considered a quick and dirty method for troubleshooting a form because you have to know very little about the Forms Debugger in order to utilize it.

In situations where the "MESSAGE" built-in is useful, you can use the "BREAK" built-in to examine *any* variables at a point in the code without the need to define a message line. When the form encounters the break line, the debugger will pop up, and with very little knowledge of the debugger tool you can expand the nodes of the navigator pane of the debugger to look at values of system variables, form variables, program variables, and global variables. Then, when you are ready to continue program execution, just hit the "GO" (lightning bolt) icon. Even if that's all you know about the Forms debugger, you will find it very useful. Of course, once the debugger is invoked, you can use any of its other functions if desired.

Besides avoiding the need to explicitly code the message line to display the desired variable values, another advantage over the "MESSAGE" built-in is that there is no need to remove or comment out the line calling the "BREAK" built-in. Unless the form is run in debug mode, it will have no effect at runtime, so your users will not be bothered by it if you leave it in.

TROUBLESHOOTING WITH THE FORMS DEBUGGER

The Forms Debugger is a simple interface which allows the developer a high degree of control in debugging applications. You can not only view values of variables, but you can actually change them at runtime to explore how various values affect results. You can step through code and even modify it for that debugger session, if desired. With Forms 5.0 and above, you can debug server-side code, view values of PL/SQL tables, and see which row a cursor is processing.

The Forms Debugger provides a comprehensive set of tools for debugging Oracle Forms, and its primary features are quite easy to use. Once you master the basics of the debugger, you will probably relegate the "quick and dirty" techniques to the back of your troubleshooting tool box, to be used only for the most elementary tasks.

INVOKING THE FORMS DEBUGGER

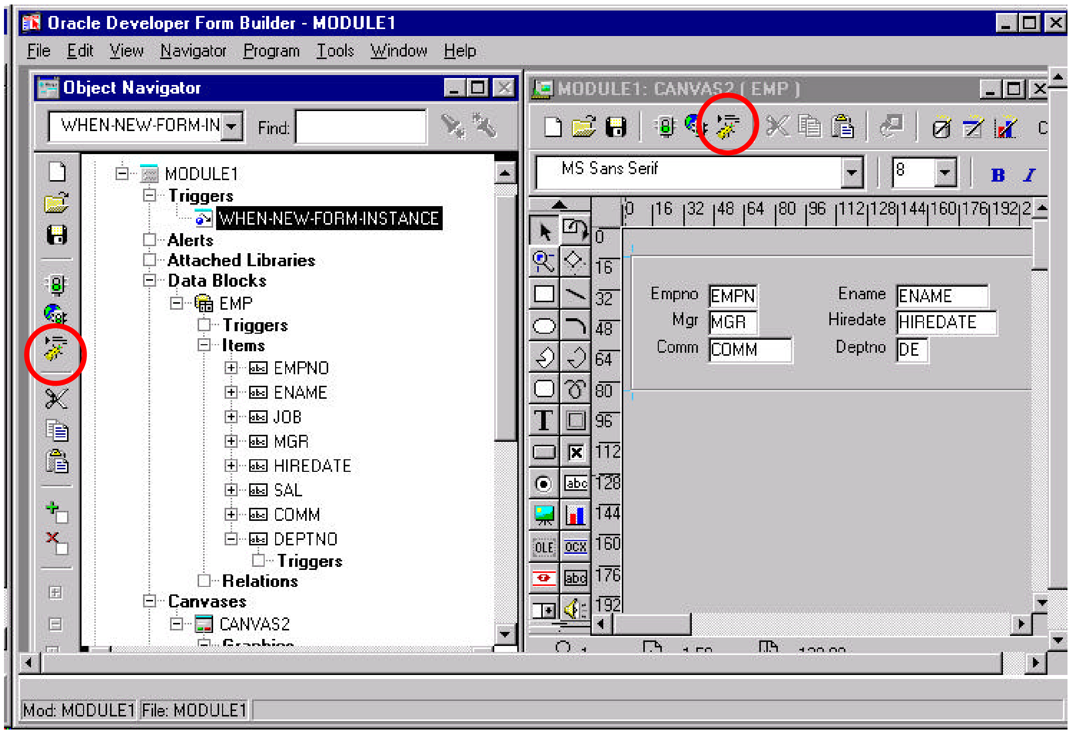


Figure 1 -- Invoking debugger from Form Builder

The simplest way to invoke the Forms Debugger is by clicking the debug icon in the Form Builder, which appears in both the object navigator and the canvas layout editor. In Forms 6.0, this runs the form in debug mode. In earlier versions, it is a toggle-type lamp which specifies whether the form is to be run in debug mode when you click the run icon. You can accomplish the same thing prior to Forms 6.0 in the Tools Options or Preferences menu.

You can also run the Forms Debugger by specifying "debug=yes" on the command line. To do this, you must use the debug executable, rather than the normal runform executable, AND the form must have been generated in debug mode. For example, to run in debug mode on Windows platforms:

```
c:\orant\bin\ifdbg60.exe module=myform userid=scott/tiger debug=yes
```

A third method of invoking the form in debug mode is to check "Run in Debug Mode" in the runform window.

If you run your form in debug mode and find that you are not able to see your program unit code, it is because you did not generate the form in debug mode. Running the form in debug mode from the Form Builder automatically generates it in debug mode.

If you are not able to see server-side code in Forms 5.0 and above, you may need to recompile that code on the server to include debug information:

```
      ALTER PROCEDURE procname COMPILE DEBUG;
```

This recompilation should enable you to step through server-side code with the Forms debugger and 5.0 or above.

When the form is running in debug mode, the debugger window will be displayed when the form first comes up, and also whenever a debug action (discussed below) is encountered. You can also manually bring up the debugger window while the form is running in debug mode by selecting Help -> Debug from the menu.

COMPONENTS OF THE FORMS DEBUGGER

When the debugger window appears, you will see three panes, which can be resized relative to one another, and a toolbar/menu section.

Toolbar/Menu Section

The toolbar and menu options available will vary depending upon what is displayed in the panes. On the toolbar, the five buttons at the left control the execution of code, while the red X button closes the debugger and the question mark displays help. The remaining buttons relate to the navigator pane.

Source Pane

The top pane displays the current program unit, if any, in read-only mode.

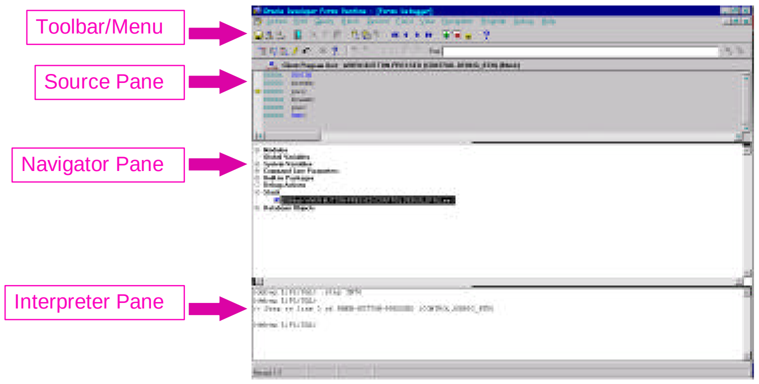


Figure 2 -- Components of Forms Debugger

Navigator Pane

The middle pane displays debugger objects, such as debug actions, built-in packages, database objects, modules, global variables, and system variables, in a tree-like structure similar to the object navigator in the Form Builder.

Interpreter Pane

The bottom pane is for entering and displaying commands to the PL/SQL interpreter.

USING THE DEBUGGER

The most common use of the debugger is to pause a program at a certain point in its execution to examine the runtime environment, then to continue to step through the code to see how each line affects the environment and the form. In addition, you could issue commands directly to the PL/SQL interpreter, or you could change variable values or lines of code to see how these modifications affect the running form.

Pausing Program Execution in a Form

Program execution is suspended when a debug action is encountered. There are two types of debug actions: breakpoints and debug triggers.

Breakpoints

A breakpoint tells the form when to pause code execution and invoke the debugger. You can set breakpoints only on executable statements; you will get an error if you attempt to set a breakpoint on a comment, begin statement, variable declaration, or other non-executable line of code. There are three ways of setting breakpoints:

1. DOUBLECLICK THE LINE OF CODE IN THE SOURCE PANE: In the navigator pane, locate the module you are running, then highlight the trigger or program unit where you wish to insert the breakpoint. This will bring that program unit into the source (upper) pane. To insert a breakpoint, double-click the line number of the line *before which* you wish to pause.

2. ISSUE A COMMAND TO THE PL/SQL INTERPRETER: Issue the following command in the interpreter (bottom) pane:
   ```
   .BREAK  PROG  <program unit name>  LINE  <number>
   ```
   Or, in the source pane, click on the desired line, then issue the following command in the interpreter:
   ```
   .BREAK
   ```
   But why would you do this, when double-clicking on the code line number in the source pane issues the break command to the interpreter automatically?

3. USE THE "BREAK" BUILT-IN IN THE FORM: As discussed under "Quick and Dirty Techniques" – this is the only way to set "permanent" breakpoints. All breakpoints set in a debugger session disappear when you close the form and must be reset the next time you run the form in debug mode. If you know you will be wanting a program to break each time you run the debugger, hard-code it in the form. Remember, the "break" built-in only has an effect when the form is run in debug mode, so will not be disruptive to users.

Debug Triggers

Another way to pause program execution is through the use of debug triggers. A debug trigger is a block of code which you define to execute either (a) every time the debugger takes control, (b) at every source code line being executed, or (c) when a specific line of source code is encountered. Debug triggers are typically used to define a conditional breakpoint. For example, you could set a debug trigger on a line of source code which would suspend the program execution if a certain variable value was greater than a specified amount:

```
IF Debug.Getn('my_sal') > 5000 THEN
   raise Debug.Break;
END IF;
```

This code calls functions of the DEBUG package, which is documented in the online help topic "About the DEBUG package".

Examining the Runtime Environment

Once you have set the desired debug action(s), you can close the Forms Debugger by clicking the red "X" icon. The form will run normally until it encounters a line of code with a breakpoint set on it. It will stop, and invoke the debugger, *prior to* executing that line of code. At that point, it may be useful to look at the runtime environment, including the following:

- Values of items (including non-displayed items), parameters, global variables, system variables
- Command line parameters
- Program stack, along with values of program unit variables for each level of the stack

Stepping through Code

Once the debugger is invoked, you can use the program control buttons on the toolbar to execute the code of the form in any manner desired. If using Forms 5.0 and above with RDBMS 7.3.4^{+} or 8.0.4^{+}, you will also be able to step through program units which are stored on the database (server-side debugging).

- Step into: The first button on the left will execute the next line of code, useful if you want to execute each line individually to examine how the code affects the form and the runtime environment. It will also step into called subprograms, allowing you to step through that code as well.

- Step over: The next button allows you to execute the next line of code while stepping over called subprograms; in other words, the subprogram will execute, but you will not step through each line of the subprogram code. The debugger will resume control after the execution of the subprogram.

- Step out: The third program control button returns control to the debugger after execution of the current subprogram, stopping where the subprogram was called. Use this if you start to step through a subprogram, then decide you do not need to step through its code.

- Go: The GO button executes all pending code, without stepping through it, until the code completes execution or another breakpoint is encountered. Use this button to dismiss the debugger window and resume normal execution of the form.

- Reset: The fifth and last program control button aborts the program units that are currently running, returning control to an outer debug level. There may be various debug levels if multiple breakpoints are encountered while the debugger has control of the program.

At each step in the code, you can enter PL/SQL commands directly if desired, and perform examination or modification of variable values or code.

Issuing Commands to the PL/SQL Interpreter

If you have been performing GUI actions in the debugger, you may notice that the commands are echoed in the PL/SQL interpreter pane. You can issue these and other commands directly, if desired. Here are some examples of PL/SQL commands that can be issued in the interpreter pane:

Set a breakpoint

```
.BREAK  PROG  <program unit name>  LINE  <number>
```

PROG stands for PROGRAMUNIT, but if you prefer, you may use the more specific terms PACKAGE, SUBPROGRAM, PROCEDURE, or FUNCTION.

You can also define a trigger to execute when the breakpoint is reached.

View source code

```
.LIST  PROG  <program unit name>
```

List currently defined program units

```
.SHOW  PROG
```

Export program units to a file

```
.export prog * file C:\<filename>.pls
```

This function is useful for documenting all of the PL/SQL code in a form. However, it has one major drawback: the names of the program units are not printed in the file. Therefore, if you intend to use this functionality, you should put a comment at the beginning of each program unit which includes its name and trigger level. For example, in a when-button-pressed trigger, you might start the trigger with:

/* CONTROL.BUTTON1.WHEN-BUTTON-PRESSED trigger

```
*/
```

Print out public package variable values

```
      TEXT_IO.PUT_LINE(<PACKAGE.VARIABLE>);
```

This is a good way to determine the values of package variables, which, if defined in the package specification, can normally not be viewed in the debugger.

Making Modifications at Runtime

In debugging your form, you may want to see the effects of modifying code. You could change the code in the Form Builder and retest it, but if you want to temporarily test how a code modification affects the running form, you can make the changes in the Forms Debugger. However, these modifications cannot be made permanent from the debugger; they are in effect only for the current debugger session. If you want to save the changes, you can copy and paste them from the debugger to the program unit in the Form Builder and save them there to make them permanent.

In a similar fashion, you may test how various changes in the values of program variables will affect the results in your form by making temporary modifications to these values as the form is running in the debugger.

Program code or variable value modification is done from the navigator pane. You can double-click on a program unit icon to bring up the PL/SQL editor. Note that it will be in read-only mode if that program unit is in the current call stack. Otherwise, you will be able to make modifications which will be in effect for the remainder of the debugger session. You can change the values of program variables in the call stack of the navigator pane, and you can even change the values of forms items by drilling down in the "module" node of the navigator. Just locate the variable in the navigator, then highlight and change its value. Subsequent program lines referencing that variable will use its new value.

Debugger Limitations

Error handling operates differently in debug mode. On-error and on-message triggers do not fire, and any errors or messages are displayed in alerts. This is actually an improvement over SQL*Forms 3.0, where you were advised to manually disable on-error and on-message triggers temporarily when running in debug mode.

There are many built-ins that will not execute prior to form startup when running in debug mode. A list of these is in the online help topic: "About Debugger startup restrictions."

Don't try to invoke help from the debugger. This causes a PDE error. See Forms bug 438135.

There is currently no easy way to determine what line of code caused an exception. However, a future release of the debugger may include this enhancement.

As already mentioned, there is no way to save debug actions (breakpoints and triggers) created in the debugger. They must be re-entered in each debugger session, unless you use the "BREAK" built-in in your code. There is also no way to save modifications made to program units.

Advantages of the Forms Debugger

Despite these limitations, the Forms debugger is a useful tool for getting a detailed picture of most elements of the form's runtime environment. It enables you to create "what if" scenarios by changing program code and variable values on the fly, and you can easily trace the effects of each line of code. In addition, when using packages such as OLE2 or TEXT_IO, you will get more meaningful error messages when running in debug mode, even if you don't set any breakpoints or step through code. Because of its functionality and ease of use, the Forms Debugger is a valuable addition to your troubleshooting tool box.

TROUBLESHOOTING WITH LOGGING AND TRACING TECHNIQUES

There are several techniques which use logging and tracing to create a record of some aspect of a running form. These are useful for detailed analysis, and often you will notice something while analyzing a log that would not have been apparent in the debugger. In addition, if you are unable to solve the problem on your own, you can send the log to an Oracle Support analyst to help resolve your Technical Assistance Request.

Don't overlook a source of help that is readily available (and free) if you have an Oracle Support license: electronic support through MetaLink. This gives you access to many of the same articles which Oracle Support analysts use to help resolve customer issues. Logging can give you information about error codes, Java error messages, and stack trace data, and you can query these keywords in MetaLink to find articles pertaining to your particular issue. You can register for MetaLink at http://metalink.oracle.com/.

To decide if logging is appropriate and which type of log is best in a given situation, you need to know the information that results from each logging technique, how to initiate logging, and how to interpret the log files created by each method.

DEBUG_SLFIND

On non-Windows platforms, problems sometimes may result from inappropriate use of resource files. You may find, for example, that some of your keys are not working as expected, or that a form's visual appearance is not correct. This could be due to Forms not being able to locate a custom resource file or TK2Motif file, and instead substituting a default resource file where the keys or visual aspects of the form are defined differently. You may also be receiving FRM errors at form startup. You can use the DEBUG_SLFIND log, available on Unix or VMS platforms, to detect problems with resource file utilization.

This log is initiated by setting an environment variable, then running the form. An example of the syntax for Unix:

```
> setenv DEBUG_SLFIND output.txt
> f60runm module=test userid=scott/tiger
> more output.txt
```

The log created is fairly easy to interpret. It displays all the calls that Forms makes to the environment, and indicates the environment variables, search paths, and resource files being searched for or accessed.

SQL TRACE

SQL Trace is server-side tracing of SQL statements. It outputs a trace file on the database server, in the directory specified by USER_DUMP_DEST in the init.ora file. Its main purpose is the tuning of SQL statements.

Tracing can be initiated for all database connections by specifying SQL_TRACE=TRUE in the init.ora file. However, it is usually much more useful to trace each session separately. You can do this in the Form Builder by checking "Statistics" in the screen where runtime options are specified, or on the runform command line with the STATISTICS=YES parameter.

Trace files must be converted to readable output before they can be interpreted. You can use the TKPROF command to perform this conversion. TKPROF will create a file that lists each SQL statement, the time taken (if TIMED_STATISTICS is set to TRUE in the init.ora file for the database), and the execution plan (if the "explain=userid/password" option is used on the command line for TKPROF). An example of using TKPROF to convert a SQL trace file to readable form:

```
tkprof ora_8927.trc /tmp/tkprof_8927.txt explain=scott/tiger
```

On Windows platforms, the command varies according to the database version (tkprof73, tkprof80) -- look in the $ORACLE_HOME\bin directory for the name of the appropriate executable.

SQL*NET TRACE

SQL*Net Trace is client-side tracing of SQL statements. It captures every interaction between any client tool and any version of the database, and can tell you the SQL statements issued and data returned.

You can initiate SQL*Net tracing by two settings in the sqlnet.ora file on the client: Set TRACE_LEVEL_CLIENT=16 (there are 16 levels of tracing -- this is the most detailed level). Set TRACE_DIRECTORY_CLIENT to the directory where you want the trace file to be created. Optional: Set TRACE_FILE_CLIENT to the desired name of the trace file (default is sqlnet.trc).

Because SQL*Net Trace creates very large trace files and can impact performance, you should exit your client session and set tracing off as soon as possible by again modifying the sqlnet.ora file:

```
TRACE_LEVEL_CLIENT=OFF
```

Also, subsequent trace operations will keep appending to the trace file, so you should either use a unique name each time, or delete or rename the file before beginning another trace. If you don't specify a file name, a unique name will be assigned each time by default.

If you are not able to change the sqlnet.ora file, perhaps because it is on a shared network, on Windows you can make a local copy of it and set your TNS_ADMIN registry setting to point to the local directory where that file is located. On Unix, you can make a copy in your home directory and name it ".SQLNET.ORA" (note the dot preceding the file name). This file will be used instead of the shared copy.

You can read the trace files directly; however, they can be somewhat difficult to interpret. They will probably contain much information that is not relevant to your purpose, since the main element of the file that will probably interest you is the listing of SQL statements being generated by the form. Fortunately the TRCEVAL utility will help you cull the useful information from the trace file:

```
Usage: trceval [-options] <filename>
Where -options can be one or more of the following:
      c     summary connectivity information
```

```
d     detailed connectivity information
f     forms application trace statistics generated
s     overall trace statistics generated
t     detailed two task packet information
u     summary two task packet information
q     displays SQL commands complete for the -u option
```

For example, you could use the following command, which redirects the output of the TRCEVAL utility to a file called "mytrace.txt":

```
trceval -qu sqlnet.trc > mytrace.txt
```

After running a simple form to generate a trace file and using TRCEVAL on it, a comparison of file sizes showed the trace file at 169K, while the file generated by TRCEVAL was only 5K. In this particular trace file, you have to scroll through 3,041 lines to find the first SQL statement issued by the form, which appears like this:

```
nspsend:00 CD 00 00 06 00 00 00  |........|
nspsend:00 00 03 47 0C 71 80 00  |...G.q..|
nspsend:00 02 00 00 00 C4 E5 91  |........|
nspsend:01 28 00 00 00 00 00 00  |.(......|
nspsend:00 00 00 00 00 B0 81 7A  |.......z|
nspsend:02 07 00 00 00 3C 6E D7  |.....<n.|
nspsend:00 02 00 00 00 00 00 00  |........|
nspsend:00 8C 6E D7 00 04 00 00  |..n.....|
nspsend:00 00 00 00 00 00 00 00  |........|
nspsend:00 53 45 4C 45 43 54 20  |.SELECT |
nspsend:44 45 50 54 4E 4F 2C 44  |DEPTNO,D|
nspsend:4E 41 4D 45 2C 4C 4F 43  |NAME,LOC|
nspsend:2C 52 4F 57 49 44 20 46  |,ROWID F|
nspsend:52 4F 4D 20 44 45 50 54  |ROM DEPT|
nspsend:20 01 00 00 00 02 00 00  | .......|
nspsend:00 00 00 00 00 00 00 00  |........|
nspsend:00 00 00 00 00 00 00 00  |........|
nspsend:00 00 00 00 00 02 03 00  |........|
nspsend:00 17 00 00 00 00 00 00  |........|
nspsend:00 00 00 00 00 01 03 00  |........|
nspsend:00 0E 00 00 00 00 00 00  |........|
nspsend:00 00 00 00 00 01 03 00  |........|
nspsend:00 0D 00 00 00 00 00 00  |........|
nspsend:00 00 00 00 00 01 03 00  |........|
nspsend:00 13 00 00 00 00 00 00  |........|
nspsend:00 00 00 00 00 00 00 00  |........|
```

However, in the file generated by the TRCEVAL command (using -qu option) from the same trace file, the SQL statement is on line 37 and looks like this:

```
SELECT DEPTNO,DNAME,LOC,ROWID FROM DEPT
```

TRCEVAL does not show the data that is sent and received, but you can refer to the trace file if this is needed. So, unless you need all the detailed network packet information, the TRCEVAL utility is the way to go!

The SQL statements in these files use bind variables. For example:

```
SELECT EMPNO,ENAME,JOB,MGR,HIREDATE,SAL,COMM,DEPTNO,ROWID FROM EMP
WHERE (DEPTNO=:1)
```

The bind variable (in this instance, ":1", shows that this is a statement internally generated by Forms. If it were a programmer-written statement, as it would be if you defined an explicit cursor to fetch the data, the bind variable would begin with "b", like ":b1".

SQL*Net tracing not only allows you to see the SQL statements being issued by Forms, therefore getting a glimpse into what's occurring behind the scenes, but it also can give you details about server-side errors that are returned in a stack. Its usefulness is apparent when you consider the purpose of a Forms application: to provide an easy user interface for issuing SQL DML statements (select, insert, delete, update) to the database.

FORMS RUNTIME DIAGNOSTICS

Beginning with Developer 1.6.1 (Forms 4.5.10.1.0), Developer 2.1 (Forms 5.0.6.5.1) and Developer 6.0 (all Forms versions), you are able to use Forms Runtime Diagnostics (FRD). This is analogous to running a form in debug mode and printing out all events that occur. While you can't affect the outcome by changing values or programs on the fly, you will get a complete log containing the following types of information:

- Files opened (fmx and mmx)
- Every trigger that fires, including the trigger level
- Every built-in that executes, including its arguments
- The complete state of the form at its inception, and changes that occur after each action
- The complete text of forms error messages and unhandled exceptions
- Every menu selection that is made

There are disadvantages to FRD as opposed to running the form in debug mode. FRD does not allow you to step through code, and in fact does not even show what program units are executing. You are not able to see the value of program variables, or even of Forms items during the execution of a trigger, although you will see the forms items that have changed after a trigger's execution has completed.

It is easy to run a form with diagnostics enabled. Just add the “record=collect” option to the command line, or for a web-deployed form, to the serverArgs in the HTML file. You can also optionally specify the name for your log file by adding the “log=<filename>” option to the command line. Examples:

```
D:\ORANT\BIN\ifrun60.EXE module=testform userid=scott/tiger
record=collect log=frd.log

Netscape:  serverArgs="module=testform userid=scott/tiger
record=collect log=frd.log"

Internet Explorer: <PARAM NAME="serverArgs" VALUE="module=testform
userid=scott/tiger record=collect log=frd.log">
```

You can combine runtime diagnostics with running the form in debug mode, getting the best of both worlds:

```
D:\ORANT\BIN\ifdbg60.EXE module=testform userid=scott/tiger
record=collect log=frd.log debug=yes
```

Note that this command uses the debug executable, and don't forget to first generate the form in debug mode.

Be aware that FRD overwrites its previous log file, if a file with the same name already exists; there is no way to append the log. Also, these log files can be very large, so use only for specific situations. The form will still function if FRD is unable to write to the log file due to a full file system or other problem, but the logging will not take place.

FRD can be especially valuable when a user is getting unexplained errors in a Form, but is unable to tell you how, or even exactly what the errors are. Enable FRD for that user, and once they report that the problem has recurred, examine the log. You will be able to see exactly what the user was doing in the form when the error was encountered.

CUSTOMIZED LOGGING WITH TEXT_IO

It's possible that none of these logging options exactly suits your purposes. Perhaps you have a form that is encountering data-specific errors, or you want to trace the value of a variable throughout a lengthy loop or while fetching a large volume of data. Stepping through the program or responding to messages would be quite tedious, while other logging methods may either not give the information you need or may be so large that extracting the pertinent data would be difficult. The solution to this type of problem is to use TEXT_IO to create your own customized log file.

You may want to create a package to perform customized logging. The following package, described in Oracle Note:61692.1 FORMS DEBUGGING TECHNIQUES, may serve as a model:

```
PACKAGE dbg IS
  filename  VARCHAR2(2000) := 'debug.out';
  PROCEDURE output(
    p_msg  VARCHAR2
  );
END;

PACKAGE BODY dbg IS
  PROCEDURE output(
    p_msg VARCHAR2
  ) IS
    outf  TEXT_IO.FILE_TYPE;
  BEGIN
    outf := TEXT_IO.FOPEN(filename, 'a');
    TEXT_IO.PUT(outf, NAME_IN('SYSTEM.CURRENT_DATETIME') || ' -
');
    TEXT_IO.PUT_LINE(outf, p_msg);
    TEXT_IO.FCLOSE(outf);
  EXCEPTION
    WHEN OTHERS THEN
      IF TEXT_IO.IS_OPEN(outf) THEN
        TEXT_IO.FCLOSE(outf);
      END IF;
  END;
END;
```

You can pass a string to dbg.output, and the string will be appended to the log file, along with a time stamp. For example:

```
dbg.output(‘The value of salary is ‘ || to_char(empsal));
```

You could even activate this only when a parameter in the form is passed a certain value. To do this, you could add to your form a parameter called, for example, FORM_DEBUG, and amend the code in the dbg package to execute conditionally by placing the following line in the package body after BEGIN:

```
if :PARAMETER.FORM_DEBUG=’YES’ then
```

Put an “end if;” before the “EXCEPTION” line, and you have enabled conditional logging. Under normal circumstances, no log file would be created. The log would be written to only if “FORM_DEBUG=YES” were passed as a parameter to the form, either on the command line or, for a form run on the web, in the serverArgs of the HTML file:

```
D:\ORANT\BIN\ifrun60.EXE module=test_form userid=scott/tiger
FORM_DEBUG=YES

Netscape:  serverArgs="module=testform userid=scott/tiger
FORM_DEBUG=YES"
```

```
Internet Explorer: <PARAM NAME="serverArgs" VALUE="module=testform
userid=scott/tiger FORM_DEBUG=YES">
```

This way, you can enable customized logging only when needed, and output any message you desire. The possibilities are endless!

TROUBLESHOOTING WEB-DEPLOYED FORMS

There are several techniques that are useful for debugging problems encountered when running forms over the web. Two such methods are mentioned here: logging connection information at the Forms Server and selecting options for the client's Java console to obtain more meaningful information. In addition, the common FRM-99999 error is specifically discussed.

CONNECTION ACTIVITY LOGGING

To get information about Forms Server connectivity, you can use Connection Activity Logging beginning with Developer 1.6.0 (Forms 4.5.9.3.0), Developer 1.6.1, 2.1 and 6.0 (all versions). This is enabled when starting your Forms Server:

```
f60ctl ... log=log_file
f60svrm ... log=<logfilePath>
ifsrv60 -listen log=<logfilepath>
```

On NT, you must start the Forms Server manually from the command line, or start the service manually with the log= option, in order to enable Connection Activity Logging. You cannot switch it on if the service starts automatically when NT boots.

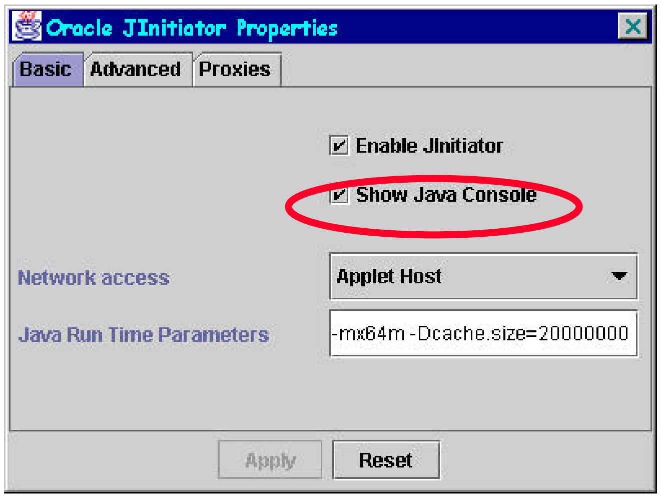
The log will contain all messages produced by the Forms Server, including:

- start-up of the Forms Server
- connection requests, establishment, and disconnection
- abnormal terminations
- IP addresses, port numbers, and process ID information

It is a good idea to keep Connection Activity Logging enabled. The log file is relatively short compared with other types of logs, and the information contained is very useful for system administrators to diagnose Forms Server problems. Later versions of Forms include a stack trace in the connection activity log if there is a crash, making this type of logging even more valuable.

THE JINITIATOR CONTROL PANEL

The JInitiator Control Panel has options which can yield data about forms running on the web. To bring up the control panel: START -> Programs -> Oracle JInitiator Control Panel.



☑ Enable JInitiator

☑ Show Java Console

Network access: Applet Host

Java Run Time Parameters: -mx64m -Dcache.size=20000000

Figure 3 -- JInitiator Control Panel

Make sure that "Show Java Console" is checked. Enabling the Java console will display detailed Java messages as the Forms application runs on the web.

You also may be able to get additional information by clicking on the "Advanced" tab and checking the "Enable Debug" checkbox.

There is a new feature beginning with JInitiator 1.1.7.15.1 which enables you to create a log of JAR file cache activity. You can enable this in the "runtime parameters" field of the JInitiator control panel, using the following options:

- -Dcache.verbose=true outputs all cache operations to the JInitiator console window, including cache hits, cache directory, cache size, file additions to and deletions from the cache.
- -Dcache.verbose.hit=true outputs file retrievals from the cache.
- -Dcache.verbose.miss=true outputs when file is not able to be retrieved from the cache.
- -Dcache.logfile specifies log file which will be appended each time JInitiator is run.
- No setting means logging will not be performed

THE FRM-99999 ERROR

The most common error encountered when running forms over the web is FRM-99999, a generic error number. When you receive this error, you must obtain more meaningful information before a direction to resolving the problem will be evident.

Connection activity logging and/or enabling the Java console from the JInitiator Control Panel may provide the information you need to resolve FRM-99999 problems. The Connection Activity log will tell you whether the user disconnected because of a problem on the client or a problem on the Form Server, and in later Forms versions will record a stack trace in the log file if the user caused the Form Server to crash..

An additional place to check is to see if stack traces have been generated on the client. If the client crashes, a file is generated with the .rpt extension concatenated to the name of the executable that launched the applet. This file is created in the same directory as the executable. If you are running JInitiator, the executable is the browser executable; if you are running the appletviewer, the executable is "appletviewer".

Another diagnostic technique available from the appletviewer is the thread dump, which is generated by pressing Ctrl-Break while in the DOS window where you started the appletviewer. This will print out to the screen all active threads, which can help detect deadlock. It is not possible to generate a thread dump from JInitiator.

A variety of conditions can cause the ubiquitous FRM-99999; these techniques should enable you to access more meaningful information about the error. You will then be able to use that information in a MetaLink query or to provide a better problem definition to the Oracle Support analyst working on your Technical Assistance Request.

SELECTING THE APPROPRIATE TOOL

With all these troubleshooting techniques at your disposal, which tool is the right one for a given situation? You should use good coding habits to begin with, such as checking for form_success after calling any Forms built-in. If errors and/or incorrect results still occur, here are some general guidelines for selecting the right method for diagnosis:

- "Quick and dirty" -- If the form is relatively simple, and you just need to ascertain where a problem is occurring, one of the "quick and dirty" techniques may be helpful: commenting out suspicious code, using the "message" built-in, or very simple invocation of the Forms debugger by using the "break" built-in. If you have no idea where the source of the problem is, you may first want to use "debug_messages=yes" on the command line to display a message for each trigger that fires; once the scope is narrowed, you could use a different technique to pinpoint it further.

- Forms Debugger -- If you need to monitor or modify the runtime environment while the form is running, use the Forms debugger. You can even use this in combination with other methods, such as some type of logging.

- Logging/Tracing -- If you need a log of Forms runtime behavior, use one of the logging or tracing techniques. These methods are especially useful for complex situations where the ability to analyze a file would be beneficial. The log should provide detailed error information containing keywords for querying in MetaLink. Also, your Oracle Support analyst may be able to resolve your issue more quickly if you are able to submit a log of the form's activity.

- Web Forms diagnostics – If the problematic behavior occurs only when running the form on the web, Connection Activity Logging or options available from the JInitiator Control Panel may help you diagnose the problem. If you are running the appletviewer and encountering FRM-99999 errors, producing a thread dump may be useful.

These tools should give you a start toward becoming an expert Forms troubleshooter!

REFERENCES

Forms Debugger

- Chapter 21, Oracle Developer/2000 Forms 4.5 Developer's Guide Manual
- Chapters 2 and 5, Oracle Developer/2000 Procedure Builder 1.5 Developer's Guide Manual
- Note:39322.1 GETTING STARTED WITH THE ORACLE FORMS 4.5 DEBUGGER
- Note:30941.1 Forms 4.5 Debugger Tutorial

DEBUG_SLFIND

- *PR:1011276.6 DEBUG_SLFIND: HOW TO DEBUG CDE TOOLS PROBLEMS WITH RESOURCE FILES*

SQL Trace

- *Appendix B of the ORACLE7 Server Application Developer's Guide (V7)*
- *Pr:1011728.6 SQL TRACING: HOW TO DEBUG CDE TOOLS PROBLEMS*
- *BUL 101128.035 QUERY AND APPLICATION TUNING USING EXPLAIN AND TKPROF UTILITY*

SQL*Net Trace

- *PR:1012290.6 SQL*NET TRACING: HOW TO DEBUG CDE TOOLS PROBLEMS*
- *Note:34022.1 SQLNet Version 2 Tracing for Client Tools*

Forms Runtime Diagnostics, Connection Activity Logging:

- *Note:62664.1 FORMS SERVER AND FORMS RUNTIME LOGGING by Duncan Mills*
- *Technical White Paper: Developer Forms Server Frequently Asked Questions, April 1999*

TEXT_IO for customized logging

- *Note:61692.1 FORMS DEBUGGING TECHNIQUES*

MetaLink

- *PR:1016352.4 ORACLEMETALINK: ORACLE'S PREMIER WEB SUPPORT SERVICE - REGISTRATION AND FEATURES*

Forms Server

- *Developer Server Troubleshooting Tips - available on Oracle Technology Network http://technet.oracle.com/products/developer/pdf/developertips.pdf*

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
+1.650.506.7000
Fax +1.650.506.7200
http://www.oracle.com/