4D Advanced Debugging Techniques – Part 2

By Josh Fletcher, Technical Support Engineer, 4D Inc.

Technical Note 06-33

Abstract

The purpose of this 2-part Technical Note is twofold:

- To provide a reference for some of the advanced debugging features that are available in 4th Dimension (4D) 2004. This was covered in part 1.
- To provide an alternative technique for debugging, based on text file logging, for when the 4D features are not available or not ideal. This will be covered in part 2.

The basic concepts of debugging were presented in part 1, as well as how they are implemented in 4D. Additionally some advanced 4D debugging techniques were highlighted.

Part 2 of this Technical Note presents a debugging technique based on text file logging and includes a 4D component that can be used to facilitate text file logging in any database. Instructions for installing and using this component, as well as an example database, are included.

Introduction

Part 1 of this Technical Note covered the basic concepts of debugging and how 4D fits into the debugging picture.

Recall that part 1 asserted the steps of debugging can be broken down as follows:

1. Acknowledge that a bug exists
2. Determine the pattern that reproduces the bug
3. Determine the source of the bug
4. Determine the cause of the bug
5. Develop a fix for the bug
6. Apply the fix
7. Test the fix

4D provides several different features that support the process of debugging:

- The debug/trace window
- The Runtime Explorer
- The debug log file
- Error messages

What if 4D cannot help?

It is important to note that, with the exception of the debug log file, none of these features are necessarily helpful in determining the pattern that reproduces a bug. These features are primarily focused on isolating the source of the bug once it has already been reproduced.

Additionally in part 1 it was suggested to follow the debugging steps in order. Note that the process of determining the pattern to reproduce the bug should ideally be completed before attempting to isolate the bug. It was also mentioned that this can be the most critical and time-consuming step.

Finally, most of these features are also not available in compiled databases so, when investigating a bug that only occurs in a compiled database, the debugging options are obviously more limited.

One alternative is to create a log file with customized information that can help the developer isolate the problem.

Advanced Debugging Part 2 – Logging

This section discusses a debugging technique that can be useful for when the 4D tools are not available or not ideal.

Introduction

The idea behind using a custom log file for debugging purposes is to avoid the shortcomings inherent in other debugging tools. For example, the use of a log file can greatly aid in step 2 of the debugging steps listed previously; determine the pattern that reproduces the bug. If the bug is possibly related to the order of and/or the timing of certain events a log file often makes it much easier to spot this kind of pattern.

Another shortcoming that the use of a log file avoids is that the log file will work in either compiled or interpreted databases.

In summary, using a customized log file has several distinct advantages over a tool like a debugger:

- Pattern recognition.
    - Can be filtered (e.g. with a text editor or spreadsheet).
    - Can compare logs to one another.
- Detecting problems related to timing/synchronization.
- Works interpreted or compiled.

These advantages are discussed in greater detail in the following sections.

Pattern Recognition

One of the primary benefits to using a log file is pattern recognition. Here is a simple example:

```
Handle Creation. Errhp: 135853116
Handle Creation. Errhp: 135890816
Handle Creation. Errhp: 135888944 <- Leak
Handle Free.     Errhp: 135890816
Handle Free.     Errhp: 135853116
Handle Creation. Errhp: 136015956
Handle Creation. Errhp: 136014084 <- Leak
Handle Free.     Errhp: 136015956
```

(Note that the text *<- Leak* was added for emphasis.)

This example text log was created from a 4D database that uses the 4D for OCI plug-in. This particular log contains the creation and deletion of 4D for OCI "error handles", which are represented in 4D code as long integer values.

This simple example demonstrates how easy it can be to recognize a memory leak an application by logging all allocation and de-allocation. In this case notice that two error handles are never freed.

Another great benefit of logging to a text file (be it plain text, XML, HTML, etc.) is that the file can be opened in another application, like a text editor or spreadsheet, and the results can be filtered. This allows the developer to log as much information as possible without concern for the complexity since the information can later be altered to highlight the important parts. This process is often iterative (read time consuming), but it often allows patterns to emerge that might not be evident from other debugging tools.

Finally a log file gives the developer a snapshot in time of the execution of the database. This is important in that, when compared to a debugger session for example, there will be a history of what was done. The debugging session will be "lost" when execution continues or the database is closed. The log file (or files) can be archived for later comparison to more recent log files. Once again this allows patterns to emerge that might be evident with other debugging tools.

Detecting Problems Related to Timing/Synchronization

Another benefit of debugging with a log file is when reproducing and/or analyzing a problem depends on the timing in the application. The debugger might not be useful because the timing is too dramatically altered (by stopping processes with breakpoints and TRACE statements the developer is drastically changing the timing of process execution). By using a log file the application can be run normally* and the timing observed after the fact by analyzing the log file.

Here is a small (and slightly contrived) example. This sample logging records a recursive factorial method that computes the factorial of the value in the interprocess variable "<>n". In this case a second process comes along and modifies <>n while the factorial method is running:

```
PID<6> (0)Start Factorial of: <>n = 20
PID<7> Changed <>n to 4 just the screw things up...
PID<6> (0)Before:   test_Factorial( 4 )
PID<6> (1)  Method Start: test_Factorial
PID<6> (1)  Before:   test_Factorial( 3 )
PID<6> (2)    Method Start: test_Factorial
PID<6> (2)    Before:   test_Factorial( 2 )
PID<6> (3)      Method Start: test_Factorial
PID<6> (3)      Before:   test_Factorial( 1 )
PID<6> (4)        Method Start: test_Factorial
PID<6> (4)        test_Factorial done!
PID<6> (4)        Method End:   test_Factorial
PID<6> (3)      After:  2:=test_Factorial( 1 )
PID<6> (3)      Method End:   test_Factorial
PID<6> (2)    After:  6:=test_Factorial( 2 )
PID<6> (2)    Method End:   test_Factorial
PID<6> (1)  After:  24:=test_Factorial( 3 )
PID<6> (1)  Method End:   test_Factorial
PID<6> (0)After:  24:=test_Factorial( 4 )

4! = 24
```

Of course the factorial of 20 is not 24. Also notice that the process ID's are logged. This makes the problem easier to spot as it can be seen that another process came along and altered the variable.

*Of course, any logging code will alter the timing of the application since it will add overhead. This is something to keep in mind if, after adding the logging code, the problem being investigated goes away and/or changes. Still, even in this case, there is benefit to using the log file because, by slowing down the application with the logging code, the source of the problem has been isolated to timing/synchronization. Thus this gives the developer a path to investigate.

Interpreted vs. Compiled

Of course another advantage to any sort of log file is it can be made to work in compiled databases. This is especially helpful if a bug is not reproducible in the interpreted database but occurs in the compiled database. The only choice is to debug the compiled database, but this can be a daunting task as the 4D debugging features are limited in compiled databases. The debugger/trace window is not available, the Runtime Explorer has limited functionality, and many of the error messages available in Interpreted mode do not exist when running compiled. The built-in 4D log file is available of course, but it is not customizable and may not contain the desired information.

Also in both cases, Interpreted and Compiled, the log file is useful for the reasons already mentioned; pattern recognition and detecting problems related to timing/synchronization.

Conclusion

The technique of debugging with a log file can be a powerful tool under the right circumstances. When searching for patterns or dealing with synchronization issues this technique is invaluable. Of course it is also one of the few techniques available when debugging compiled databases. When coupled with the debugging techniques covered in part 1 of this Technical Note the 4D developer has a potent debugging arsenal.

4D Logger 2004-1 Component Reference

The "4D Logger 2004-1" component (referred to as "Logger" for short) provided with this Technical Note can be used to facilitate text file logging in any database. It is presented "open source" and the developer is free to alter it as needed.

Important General Notes

This section details some of the general concepts that went into the design of the Logger component.

Stack Level

When logging debug messages the concept of a "stack frame" is used in some cases. This is referred to as the "stack level" in the component code. The stack level is indicated in the log file by both an integer value and indentation using spaces. Each time the start of a method is logged, for example, the stack level for that process is increased. Thus subsequent log entries are indented further. When the end of the method is logged, the stack level is decreased.

Note that this has nothing to do with the actual stack used by 4D. This is done only to make the log file easier to read. Here is an example:

```
PID<1> (0)Method Start: OD Execute SQL
PID<1> (1)      Method Start: OCI_Exist_Login
PID<1> (1)      Method End: OCI_Exist_Login
PID<1> (1)      Method Start: OD Create cursor
PID<1> (2)              Method Start: OCI_Exist_Login
PID<1> (2)              Method End: OCI_Exist_Login
PID<1> (2)              Before: OCIHandleAlloc( 129735744; 0; ...
PID<1> (2)              After: 0:=OCIHandleAlloc( 129735744; 129796272; ...
PID<1> (1)      Method End: OD Create cursor
PID<1> (0)Method End: OD Execute SQL
```

The stack level is only adjusted in certain cases (see the next section, "What gets logged?"). The stack level can also be adjusted manually by the developer.

What Gets Logged?

The Logger provides 6 types of logging, with appropriate message formatting. The idea of having different commands for message logging is to make the logging code more generic and yet easier to understand. For example, one could simply write:

```
Log_a_message("Before: OCIHandleAlloc( 129735744; 0)")
```

However this is not very general purpose and the values are all hard coded.

In the Logger different commands are provided to log function calls, procedure calls, etc. that automatically parse the parameters and format the message. This makes the 4D code easier to understand and makes the log file look more consistent.

Functions:

For functions, the Logger provides commands to log information before and after a function call, including parameters, the function name, and the result. The stack level is increased during the "before" call and decreased during the "after" so that log entries that occur in between these calls are indented appropriately. Here is an example:

```
PID<7> (3)      Before:    test_Factorial( 2 )
PID<7> (4)                 Before:    test_Factorial( 1 )
PID<7> (5)                            test_Factorial done!
PID<7> (4)                 After:  2:=test_Factorial( 1 )
PID<7> (3)      After:  6:=test_Factorial( 2 )
```

Procedures:

Procedure logging is the same as for functions except no return value exists and is therefore not logged.

Methods:

For methods, the Logger provides commands to log the start and end of a method. The format of the log entry is like:

PID<*process number*> (*stack level*) *indentation* Method Start: *method name*

For example:

```
PID<1> (2)      Method Start: OCI_Exist_Cursor
PID<1> (2)      Method End: OCI_Exist_Cursor
```

These commands are meant to be used at the beginning and end of methods in the database (as opposed to before and after a method call). As such the stack level is not adjusted since these commands are simply meant to note the beginning and end of a method call.

Messages with stack level:

Used for more general purpose messages where the stack level is still desirable. For example:

```
PID<1> (3)      Message: Foo
```

```
PID<1> (4)          Message: Bar
PID<1> (5)                Message: Baz
```

Messages without a stack level:

Used for more general purpose messages where only the process ID of the process the message came from, but no stack level, is logged e.g.:

```
PID<5> A random message from process 5.
```

Unformatted messages:

Used for messages that are logged "as is". The only formatting applied is to ensure that the message appears on a new line.

What Parameter Types Are Supported?

Currently the logging code only supports integer (C_INTEGER, C_LONGINT) and string (C_STRING, C_TEXT) parameters. Other types of parameters will appear as "<n/a>" in the log file. However, the parameter logging functionality is easy to extend. See the component method **DL_Tool_ParamToString**.

Debug Log Monitor Process

Any form of text file logging runs the risk of hard disk saturation. As more messages are logged the log file will grow continuously.

The Logger provides a 4D process that can be used to monitor the debug log file in order to prevent hard disk saturation. Note: if this process is not used, the debug log file will grow continuously. The process is started by default (assuming **DL_INIT** has been executed).

Debug Log Files

The debug log files created by the component are plain text files.

To prevent hard disk saturation, the debug log files created by the component are capped at 500KB by default. Log files that exceed this size are "archived" as separate files. The maximum log file size may be changed by the developer.

Only the last 6 archived log files are kept, by default. This is also done to prevent hard disk saturation. Note that this is only maintained "per session", i.e. each time the database is run. Archived log files from a previous session will not be deleted. The maximum number of archives to keep may be changed by the developer.

Note that these features only work if the log monitor process is running. Otherwise the log file will grow indefinitely.

Semaphores are used to allow multiple processes to access the debug log file.

Finally note that the log file is not created until a message is actually logged, i.e. when DL_INIT runs it does not actually create the log file, just the file name. This is done in case the developer wants to move the location of the log file (to prevent empty log files popping up where they may not be wanted).

Debug Log Flags

The Logger implements a "flagging" system so that the debug logging can be turned on or off as needed. The details of the system will be covered later but the general idea is to provide an easy way for the developer to globally turn logging code on or off. Multiple debug flags may be used so that the logging code can be broken into multiple chunks that can be turned on or off independent of one-another. For example one flag could be set for function calls and another for diagnostic messages.

This feature is entirely optional. It does not prevent the use of the other logging features.

Internal Members and Methods

None of the variables ("members" for the OOP folks) in the Logger are meant to be accessed directly. There are methods provided to change the values of the appropriate variables. However there is no way to prevent access to these variables in a component made of public objects. Use caution when modifying the variables directly.

In general the methods prefixed with "DL_Tool" are not meant to be used by the 4D Developer. Use caution when calling these methods and be sure to understand their purpose.

Important Method Notes

This section covers important information about specific methods in the Logger component.

Method: **DL_INIT**

This method takes care of the initialization of the logging code and starts the log monitor process. This method should be placed in the database startup code and must be executed before any logging can be done. **Also note that the delay used by the component for checking semaphores is set**

here (the variable is called "<>DL_K_DELAY"). By default this is set to 10 ticks.

Method: **DL_LOGFILE_NEW**

To modify the name of the debug log file, modify this method. An interface to modify the log file name is not provided in the Logger component (other than directly modifying the interprocess variable, which is not recommended). By default the name is of the form:

Log Created MM-DD-YYYY hh_mm_ss.txt

Method: **DL_Tool_ParamToString**

This method controls how parameters are formatted for logging. For example if a text parameter that was passed to "DL_Log_FunctionBefore" is greater than 20 characters long it is truncated and an ellipsis is added to it to improve readability of the log file.

To add support for the logging of other parameter types besides integers and strings, modify this method. The input is a pointer to the parameter and the output should be a string representation of that parameter.

Methods: **DL_Constants_@**

The functions **DL_Constants_ERROR**, **DL_Constants_OFF**, and **DL_Constants_ON** are meant to be treated as constants. This was simply a design decision to allow a bit of "data hiding" as well as avoid the overhead of needing to install 4DK# resources for the component. Using these functions in place of the actual interprocess variables (<>DL_K_ERROR, <>DL_K_OFF, <>DL_K_ON) prevents the developer from touching the component variables directly. The methods are primarily concerned with the debug log flag feature mentioned above and will be discussed in greater detail later.

4D Logger 2004-1 Command Reference

This section documents the commands available in the Logger.

Themes

The methods in the Logger component are organized into the following themes (indicated by the prefix of the method name):

DL_Constants
Methods used to represent "constants", in lieu of accessing the variables directly.

DL_Flag
Methods concerned with the debug log flag system.

DL_Get / DL_Set
Methods used to access the component variables that the developer might need to modify, in lieu of accessing the variables directly.

DL_LOGFILE
Methods related to the log file.

DL_Log
Methods used for logging messages.

DL_Monitor
Methods to start and stop the log file monitoring process.

DL_Tool
Methods that, in general, should not be called directly by the developer. These are methods that the component uses internally.

Methods

Note that the only methods documented here are the commands useful to the 4D developer. For component methods not covered here, refer to the comments in the source code.

Finally, all of the methods in the component contain header comments that describe the purpose of the method as well as list input/output parameters, as appropriate.

DL_Constants_ERROR
Returns the value of <>DL_K_ERROR, used for comparisons.

DL_Constants_OFF
Returns the value of <>DL_K_OFF, used for comparisons.

DL_Constants_ON
Returns the value of <>DL_K_ON, used for comparisons.

DL_Flag_Create
Use to create a new debug log flag.

DL_Flag_Delete
Use to delete a previously created debug log flag.

DL_Flag_State
Use to check the state of a debug log flag. The possible states are DL_Constants_ON or DL_CONSTANTS_OFF. DL_Constants_ERROR is returned of the flag does not exist, or some other problem occurred.

DL_Flag_TurnOff
Use to turn a debug log flag off.

DL_Flag_TurnOn
Use to turn a debug log flag on.

DL_Get_LogFileName
Returns the file name of the current log file. A semaphore is used to ensure that the current name is returned.

DL_Get_LogFilePath
Returns the path to the current log file. A semaphore is used to ensure that the current path is returned.

DL_Get_LogLimit
Returns the maximum number of archived log files kept.

DL_Get_LogSizeLimit
Returns the maximum size the log file can reach before it is archived.

DL_INIT
Declares the interprocess variables that are used for by the component. Should only be called once, preferably in the database startup code, and ALWAYS before executing any other DL_@ calls.

DL_LOGFILE_DELETECURRENT
Deletes the current log file if it exists. Note: the log file name is un-altered. Subsequent logging calls will write to a new file of the same name as the one that was just deleted. To create a new log file (new name), call DL_LOGFILE_NEW.

DL_LOGFILE_NEW
Creates a new log file name. Subsequent logging will use this new file. Note that the file itself is not created at this time, only the file name.

DL_Log_FunctionAfter
Use to log a message to the log file to indicate that a function call has completed. The format of the message is:

PID<*process ID*> (*stack level*) After: *ReturnVal*:=*FunctionName*(*p1*;*p2*;...;*pn*)

Where p1...pn were the parameters to the function.

DL_Log_FunctionBefore
Use to log a message to the log file to indicate that a function is about to be called. The format of the message is:

PID<*process ID*> (*stack level*) Before: *FunctionName*(*p1*;*p2*;...;*pn*)

Where p1...pn are the parameters to the function.

DL_Log_Message
Use to log a message to the log file. The message is placed on a new line.

DL_Log_MessageNoStack
Use to log a message to the log file. The process ID is prepended to the message. The message format is:

PID<*Process Number*> *message*

DL_Log_MessageWithStack
Use to log a message to the log file. The process ID and stack level are prepended to the message, along with padding to indicate the stack level. The message format is:

PID<*process number*> (*stack level*)*padding message*

DL_Log_MethodEnd
Use to log a message to the log file to indicate that the end of a 4D method has been reached. The message format is:

PID<*process number*> (*stack level*)*padding* Method End: *method name*

DL_Log_MethodStart
Use to log a message to the log file to indicate that the start of a 4D method has been reached. The message format is:

PID<*process number*> (*stack level*)*padding* Method Start: *method name*

DL_Log_ProcedureAfter
Use to log a message to the log file to indicate that a procedure call has completed. The format of the message is:

PID<*process ID*> (*stack level*) After: *ProcedureName*(*p1*;*p2*;...;*pn*)

Where p1...pn were the parameters to the procedure.

DL_Log_ProcedureBefore
Use to log a message to the log file to indicate that a procedure is about to be called. The format of the message is:

PID<*process ID*> (*stack level*) Before: *ProcedureName*(*p1*;*p2*;...;*pn*)

Where p1...pn are the parameters to the procedure.

DL_Monitor_Start
Creates a process that will be used to monitor the log file and archive it if needed.

DL_Monitor_Stop
Use to stop the log monitor process. Sets an interprocess variable so that the log monitoring process can be stopped.

DL_Set_LogFilePath
Use to set the path where the log file will be saved. Default path is the temporary folder (as returned by the **Temporary folder** 4D command).

DL_Set_LogLimit
Use to set the maximum number of archived log files to keep. Default is 6.

DL_Set_LogSizeLimit
Use to set the maximum size the log file can reach, in bytes, before it is archived. Default is 500KB.

The remaining Logger methods are not covered here. These are methods that are primarily used internally by the component. More information can be found in the source code for the methods:

DL_Tool_CheckFlagArrays
DL_Tool_CreateLogFile
DL_Tool_Error
DL_Tool_FormatLogFilePath
DL_TOOL_LOGFILE_ARCHIVE
DL_Tool_LogMessage
DL_Tool_MonitorLog
DL_Tool_ParamCountAlert
DL_Tool_ParamToString

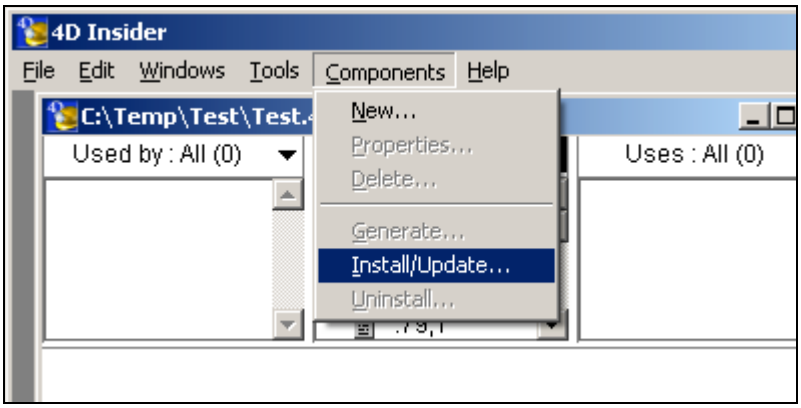
DL_Tool_StackDecrease
DL_Tool_StackIncrease

4D Logger 2004-1 Installation Procedures

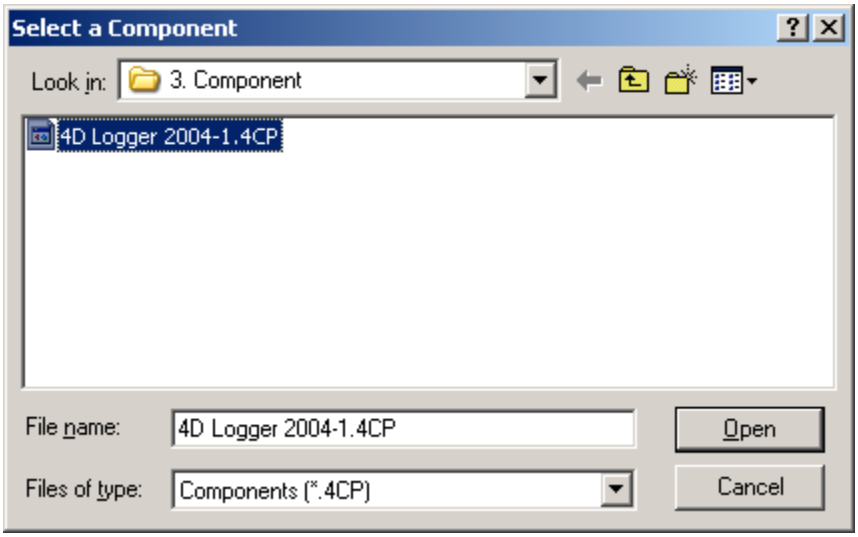
Note that 4D 2004 is required for the 4D Logger 2004-1 component.

Installing the 4D Logger component

1. Open the database with 4D Insider.
2. Open the **Components** menu and select **Install/Update...**:



3. Browse for the "4D Logger 2004-1.4CP" file and click the **Open** button:



4. After the component is installed quit 4D Insider.
5. Open the database with 4D.
6. Add the method **DL_INIT** to the **On Startup** database method (or at least within the context of the startup code.
7. Quit 4D.
8. Finally open the database in 4D again so that DL_INIT will be executed (alternatively execute DL_INIT after step 6).

Using the 4D Logger Component

Basic Usage

After completing the installation steps no further configuration steps are needed to start logging with all of the default settings. Any of the methods in the DL_Log theme may be used to start logging messages to the log file. For example, execute the following to add the message "Hello World!" to the log file on a new line:

```
DL_Log_Message("Hello World!")
```

Use the commands in the DL_Set theme to change the log file settings:

- The location of the log file.
- The maximum number of archived log files to keep.
- The maximum size the log file can reach before archiving.

Use the DL_Monitor commands to start or stop the log monitoring process. **Beware that the log file will grow continuously if the monitor process is stopped.**

In-Depth DL_Log Examples

Parameters passed to the DL_Log methods (when applicable) need to be pointers. For example, if a string literal is being passed to a function and the function call is to be logged, this will not work:

```
DL_Log_FunctionBefore ("someFunction";"someStringLiteral")
SomeFunction ("SomeStringLiteral")
```

Instead declare a local variable that will be used to store the literal for the logging code, e.g.:

```
C_TEXT($debug_StringLiteral)
$debug_StringLiteral:="someStringLiteral"
DL_Log_FunctionBefore ("someFunction";->$debug_StringLiteral)
SomeFunction ("someStringLiteral")
```

Note that it is **not** recommended to use the logging variable in the "real" code as this makes it far more difficult to remove the logging code if needed. For example, do not do this:

```
C_TEXT($debug_StringLiteral)
$debug_StringLiteral:="someStringLiteral"
DL_Log_FunctionBefore ("someFunction";->$debug_StringLiteral)
SomeFunction ($debug_StringLiteral)
```

Note debug variable used instead of the original value.

Instead leave the original code untouched so that the logging code can be easily removed (although, as a side note, the string literal should probably be replaced with a variable so that there are not "magic values" in the database).

It is also recommended to wrap the logging code in comments to make it even easier to remove, if needed. For example:

```
  `-- Begin Debugging code block--------------------------------
DL_Log_MethodStart (Current method name)
  `-- End Debugging code block----------------------------------
```

Surrounding the logging code in some sort of consistent comment block makes it easier to find all of the logging code at a later time (with **Find in Database**, for example) and remove it or alter it as needed.

Function example:

4D code, executed from User/Custom Menus process:

```
C_LONGINT($n;$result)
$n:=1
DL_Log_FunctionBefore ("test_Factorial";->$n)
$result:=test_Factorial ($n)
DL_Log_FunctionAfter (->$result;"test_Factorial";->$n)
```

Resulting messages in log file:

```
1  PID<1> (0)Before:    test_Factorial( 1 )
2  PID<1> (0)After:  1:=test_Factorial( 1 )
3
```

Procedure example:

4D code, executed from User/Custom Menus process:

```
C_LONGINT($foo)
$foo:=100
DL_Log_ProcedureBefore ("some_procedure";->$foo)
Some_Procedure ($foo)
DL_Log_ProcedureAfter ("some_procedure";->$foo)
```

Resulting messages in log file:

```
1  PID<1> (0)Before:     some_procedure( 100 )
2  PID<1> (0)After:      some_procedure( 100 )
3
```

Method example:

4D code, executed from the method "SomeMethod" in the User/Custom Menus process:

```
DL_Log_MethodStart (Current method name)
   ` Code for SomeMethod…
DL_Log_MethodEnd (Current method name)
```

Resulting messages in log file:

```
1  PID<1> (0)Method Start: SomeMethod
2  PID<1> (0)Method End:   SomeMethod
3
```

Messages with stack level example:

4D code, executed from User/Custom Menus process during the 5th execution of the recursive function "test_Factorial" (aka 5th stack level):

```
DL_Log_MessageWithStack (Current method name+" done!")
```

Resulting message in log file:

```
1  PID<1> (5)           test_Factorial done!
2
```

Messages without stack level example:

4D code, executed from User/Custom Menus process:

```
DL_Log_MessageNoStack ("Begin test!")
```

Resulting message in log file:

```
PID<1> Begin test!
```

Messages with no extra formatting example:

4D code, executed from anywhere:

```
DL_Log_Message ("Hello World!")
```

Resulting message in log file:

```
Hello World!
```

Using Log Flags (DL_Flag examples)

To create a log flag (they are turned on by default):

```
DL_Flag_Create ("MyFlag")
```

To use the flag to control logging code:

```
If (DL_Flag_State ("MyFlag")=DL_Constants_ON )
      ` Do logging code…
End if
```

Note that the DL_Constants methods are used when checking the state of a debug log flag (with the possible states being ON, OFF, or ERROR).

To turn a flag off:

```
DL_Flag_TurnOff ("MyFlag")
```

To delete a flag:

```
DL_Flag_Delete ("MyFlag")
```

Why use multiple log flags? Here is an example that uses 2 log flags. One is used to control the logging of informational messages and the other to control the logging of actual code (e.g. a procedure call):

```
C_TEXT(LogCodeFlag;LogMessFlag)
C_TEXT($pl)
LogCodeFlag:="Log Code"
LogMessFlag:="Log messages"
$pl:="A parameter for the MESSAGE command…"
If (DL_Flag_State (LogMessFlag)=DL_Constants_ON )
   DL_Log_MessageWithStack ("COMMENT: End variable declaraions..")
End if
If (DL_Flag_State (LogCodeFlag)=DL_Constants_ON )
   DL_Log_ProcedureBefore ("MESSAGE";->$pl)
End if
MESSAGE($pl)
```

If the flag “Log messages” is turned off, the call to DL_Log_MessageWithStack will not occur. Similarly if the flag “Log Code” is turned off, the call to DL_Log_ProcedureBefore will not occur. Thus the developer can control which sets of logging code get executed by turning log flags on and off.

Putting it all together

The sample database included with this Technical Note (Logger_Test.4DB) includes test code that exercises all of the logger methods. It is a small database (the logging code is probably larger than the real code) but demonstrates the use of all of the logging features.

A Final Word

There may be features of the 4D Logger component that you do not like or that do not highlight a problem that you are trying to debug. If this is the case, remember that you have the source code! Please feel free to alter the design as it suits your database. A great attempt has been made to document the source code of the Logger methods to make it easy should the developer choose to change the design. Hopefully you will find the 4D Logger code useful in any database.