4D Advanced Debugging Techniques – Part 1

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

Technical Note 06-31

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 will be 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.

In part 1 the basic concepts of debugging are presented as well as how they are implemented in 4D. Finally some advanced 4D debugging techniques are highlighted.

Part 2 of this Technical Note will present a debugging technique based on text file logging and will include a 4D component that can be used to facilitate text file logging in any database. Instructions for installing and using this component will be included.

Introduction

This section covers the basic concepts of debugging and how 4D fits into the debugging picture.

Debugging Concepts

The Wikipedia encyclopedia (http://www.wikipedia.org/) defines debugging as follows:

> *"**Debugging** is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware thus making it behave as expected."*
> http://en.wikipedia.org/wiki/Debugging

Perhaps the most important word in the above definition is "methodical". It is very important to approach debugging from a methodical perspective. Not using an organized approach to debugging will make it far more difficult to isolate problems and might even introduce new ones.

The basic 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.

Step 1 in this list is often overlooked but it is important to mentally acknowledge that a bug exists. Of course no one likes to admit that they write buggy code but if debugging is approached with the mindset that the there are no bugs then finding the cause of the bad behavior will be that much more difficult.

Other the other hand it is important to clearly identify what the undesirable behavior is and how to reproduce it, which brings us to step 2. Simply saying "this database crashes" is not sufficient. If a program crashes this is, of course, undesirable but in terms of isolating the problem one must be able to determine **when** and **how** the program crashes before even attempting to address step 3 (the **source**) or step 4 (**why** the program crashes). For example users of the software might say the program crashes "randomly". Saying that a program can crash randomly is a logically false statement. Short of random fluctuations in the electricity flowing through the computer (and some might say even this is not random), there is always a perfectly logical explanation for why a program might crash. After all computers are devices built on logic. What the term "random crashing" really points to is that the problem at hand may be hard to isolate. Establishing the pattern that reproduces the bug can be a daunting task given that there are so many variables and layers that go into any piece of software. Often the most difficult and time-consuming part of debugging is determining the pattern that reproduces the bug, not determining where or why the bug occurs. Thus establishing the pattern is critical to the debugging process.

Once a reproducible pattern has been established for the bug step 3 can be tackled; determine the source of the bug. Depending upon the complexity of the software and, especially in 4D's case, whether or not the database is compiled this task can be anything from trivial to very time consuming. Still the basic procedure is the same: find the specific event that causes the buggy behavior. This could be a single method call or a single button click or a single bad record. The pattern helps limit the search to specific areas of the software but the bug (or bugs) can usually be broken into atomic parts. This step is critical.

Once the source of the bug has been determined it is time to analyze the cause (step 4). Was it bad data in the record? Was it an unexpected input or return value? This is where step 3 becomes critical because modern software is often made up of many modules that the developer may or may not have access to. If the bug can be isolated to a single line of code and that single line of code happens to be a function call to a third-party library it may be impossible to completely determine the cause of the bug (without the help of the third party of course).

When developing bug fixes (step 5) there is one important point that should be acknowledged: if a fix does not actually stop the problem, be sure to back out the changes before trying another fix. If any non-useful alterations are left in they might introduce further bugs. Of course there are situations where the fix might need to be applied in layers so it might not **always** be appropriate to back out all changes. This point is not meant to be a "hard" rule but more a mental exercise that more often than not speeds the debugging process.

Step 6 (apply the fix) is straightforward, with some minor points. If the bug is in the 4D code then the source code must be available (i.e. a copy of the interpreted database must be available). If the bug was caused by bad data the behavior can at least be avoided by fixing the data (although the code should still be changed to prevent the bug from happening again). In other words the fix might not always be in the 4D code, or in any other code, but simply a matter of maintenance.

Testing the bug fix (step 7) often simply involves repeating the pattern established in step 2 to see if the undesirable behavior stops.

Note that the numbering of these steps is not meant to imply that this process is serial. The process of debugging often involves much iteration through these steps but, in general, it is best to try to execute them in order, i.e. do not jump from step 2 (pattern) to step 5 (fix) because you think you might know the cause of the bug. If you are wrong you might have made more work for yourself by needing to back the fix out and re-evaluating the pattern to make sure it is valid. Sticking with the methodical approach will more often than not make the process go smoother.

Where does 4D fit in?

The techniques presented in this Technical Note will be on focused on steps 2, 3, and 4 from the list above. That is:

2. Determine the pattern that reproduces the bug
3. Determine the source of the bug
4. Determine the cause of the bug

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

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

The **debug/trace window** is an extremely powerful tool for debugging interpreted databases. It provides a host of abilities that make finding the source and cause of a bug much easier. The debug/trace window is not, however, available in compiled databases.

The **Runtime Explorer** is similar to the debug/trace window in usefulness with the added bonus that some of the information in contains is available in both interpreted and compiled databases.

The **debug log file** generated by 4D is essentially a log of the call chain for a 4D database. The primary advantages that 4D's debug log file has over a log file created by the developer are: lower level 4D calls are logged in addition to the logging of the developers methods; and the file access is controlled by 4D, i.e. the developer does not need to manage access to the file.

There are several different kinds of **error messages** that 4D can present depending on a given situation. These messages are often overlooked in terms of debugging and it is important for the 4D developer to acknowledge that, in many cases, the error message can point to the exact line of code and reason that might be causing the problem. However, as with the debug window and the Runtime Explorer, some types of error messages are not available in compiled databases.

What if 4D cannot help?

It is important to note that, with the exception of the debug log file, none of the above features are particularly 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.

Recall that it was suggested to follow the list of debugging steps in order, i.e. the process of determining the pattern to reproduce the bug should ideally be completed before attempting to isolate the bug.

In addition, 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 more limited.

One alternative to alleviate both of these problems is to create a log file with customized information. This file can then be used to recognize patterns in the database, as well as for isolating the source of a bug. Part 2 of this Technical Note will explore this solution in greater depth (as well as provide a 4D component that implements it).

Advanced Debugging Part 1 – 4D Techniques

This section covers some helpful and often overlooked features of 4D that are available for debugging. The techniques are grouped based on the list given above and repeated here:

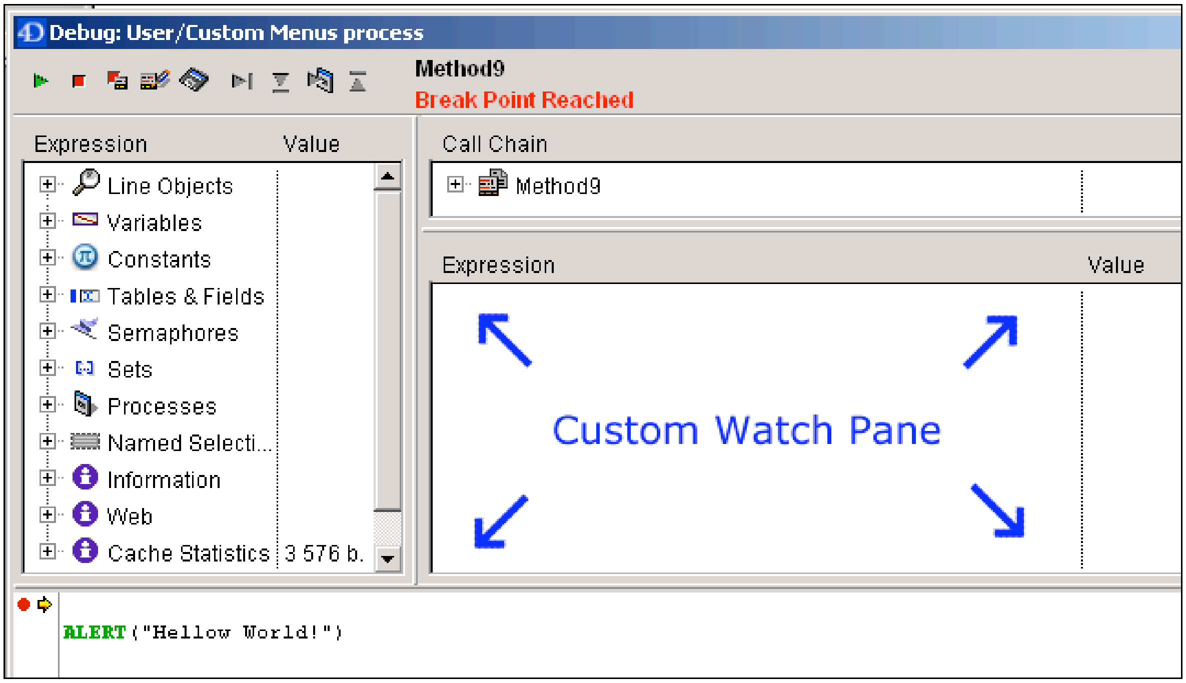
- The debug/trace window
- The Runtime Explorer
- The debug log file
- Error messages

The Debug/Trace Window

The debug/trace window is already well documented in the existing 4D documentation so it is not covered in-depth here. Refer to the "Design Reference" and "Language Reference" for your version of 4D for general information about this feature.

This section aims to cover two of the less used/overlooked features of the debugger: the Custom Watch Pane and the Program Counter.

The Custom Watch Pane is an extremely powerful tool when it comes to debugging. Here is a screenshot of the debugger with the Custom Watch Pane highlighted:



```
ALERT("Hellow World!")
```

Of course the value of variables and expressions can be viewed in this pane but it is important to note that **any** 4D expression can be entered here. 4D commands, project methods, plug-in methods, etc. can be executed.

Here is a simple example:

Given a project method that adds the two long integer parameters together and returns the result, e.g.:

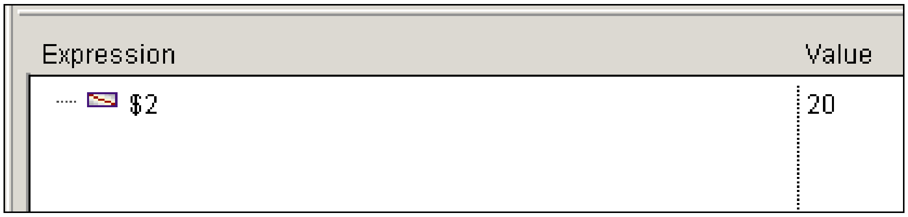
```
` Add two numbers...
C_LONGINT($1;$2;$0)

$0:=$1+$2
```

When debugging this method, after executing the line "$0:=$1+$2" it is determined that the value in the second parameter is invalid. A new value must be entered for testing. The new value will be "5". However, rather than aborting the process and executing the method again, just change the value of $2 in the Custom Watch Pane. To do this the variable $2 must first be added to the Expression list. There are a few ways to accomplish this:

- Right-click in the Expression list and choose "New Expression…" from the context menu. This opens the Formula Editor. Use the Formula Editor to insert $2.
- Double-click in the Expression list to create a new, empty expression and type in "$2" (without the quotes). Press RETURN to create and evaluate the expression.
- Highlight "$2" with the mouse anywhere in the method. Hold the CTRL (Windows) or Command (MacOS) key and single-click on the highlighted text. Note that this technique works for any highlighted text. Even if the highlighted text is not a valid expression it will still be inserted into the Expression list.

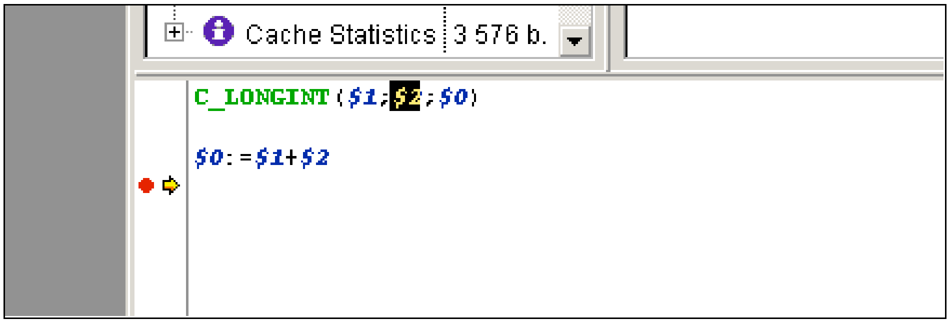
Regardless of the technique used, the expression "$2" will be inserted in the Expression list, as shown here:



Expression	Value
$2	20

To change the value of $2 to "5", single-click on the value under the Value list to edit it (note that ENTER/RETURN must be pressed to make the change stick). At this point the value of $2 has been changed. All subsequent lines of code will use this new value.

Now the addition must be executed again but that line of code has already been executed. Once again, rather than aborting the process and starting over, another of the debugger's excellent features can be used: the Program Counter. This is the small yellow arrow that appears in the Source Pane, as can be seen in this screenshot:

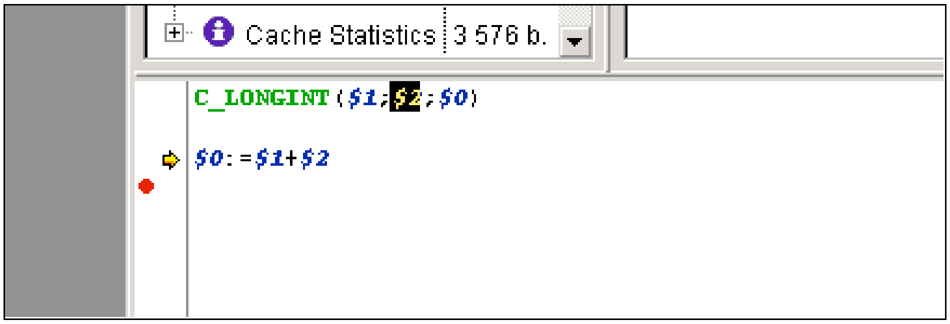


```
C_LONGINT($1;$2;$0)

$0:=$1+$2
```

This arrow indicates which line of code in the debugger will be executed next (e.g. if “Step Over” is chosen). It is important to note that the Program Counter can be moved to any other line in the method, thus allowing the code to execute in any order. This is a very powerful tool!

In this example the Program Counter needs to be moved back one line in order to execute the addition again:



```
C_LONGINT($1;$2;$0)

$0:=$1+$2
```

Then execute a “Step Over” to execute the line of code.

Here is another example to demonstrate the power of the Custom Watch Pane:

Using the same method and same problem as above, rather than adding the parameter $2 to the Expression list, altering its value, and using the Program Counter to re-execute the addition, do the following:

- Double-click the Expression list to insert a new expression.
- Enter “$1+5” (press ENTER/RETURN to force the debugger to evaluate the expression).

In this example the result of performing the previous steps can be seen without actually altering the value of the $2 parameter. Again, **any** 4D expression can be added to the Custom Watch Pane!

There is one important caveat to using expressions in the Custom Watch Pane: anytime the debugger window gains focus or the Program Counter advances, the expressions in the Custom Watch Pane are re-evaluated. This can be dangerous.

For example entering “ALERT(“Hello World!”)” in the Custom Watch Pane can create an endless loop of Alert dialogs since each time the Alert window is closed the debugger will gain focus and the Alert will open again (Incidentally, to get out of this loop, use the spacebar to repeatedly close the alerts while repeatedly clicking on where the expression appears behind the alert. If the timing is right, the expression should be highlighted. Once the expression is highlighted, repeatedly press the delete or backspace key while pressing spacebar to close the alerts. If the timing is right the offending expression will be deleted from the Custom Watch Pane).

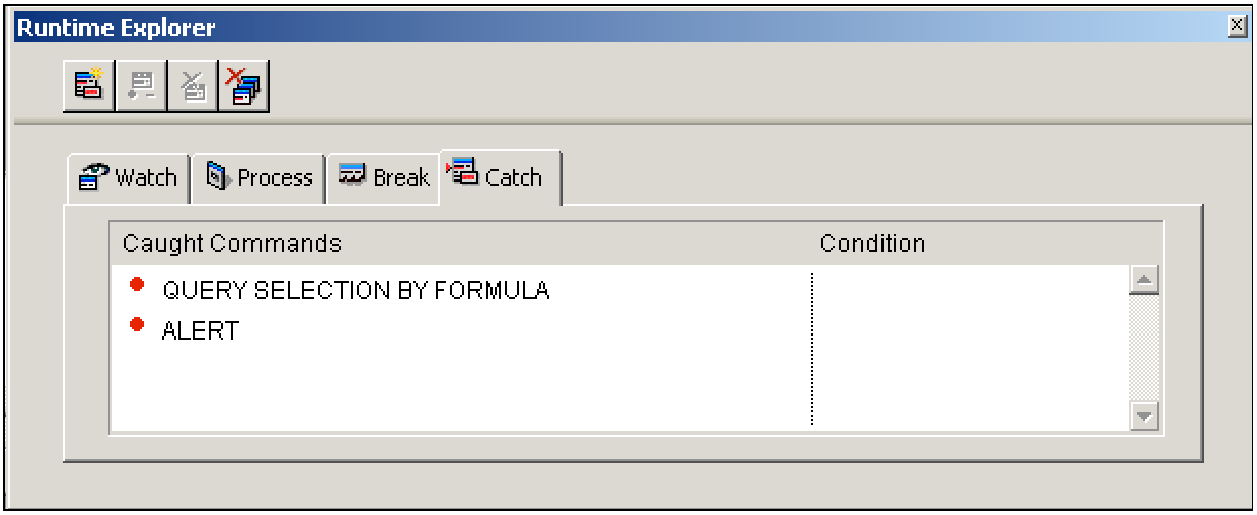
The Custom Watch Pane and Program Counter are two indispensable features of the 4D debugger. With these tools code can be executed in any order the developer chooses and any 4D expression can be executed at any time during the debugging process. This often eliminates the need to stop and re-start processes when debugging, which can lead to tremendous time savings depending upon the design of the database and how complex a pattern is needed to reproduce a given bug.

There are however a few significant drawbacks to the Debug/Trace window: it is not available in compiled databases; if there is a problem in the database due to timing (e.g. when things are executing) the debugger is not well suited to tracking them down since, but its nature (by artificially stopping execution), the timing of execution is being altered; and the debugger is not efficient for spotting larger patterns in the database.

The Runtime Explorer

As with the debugger, the Runtime Explorer is well documented in the existing 4D documentation. Rather than cover all of the Runtime Explorer features, this section will focus on one of the most powerful debugging features of 4D that is often overlooked because it is not a part of the Debug/Trace window: the Caught Commands List.

Here is a screenshot of the Runtime Explorer with the Caught Commands List page open:



Caught Commands	Condition
● QUERY SELECTION BY FORMULA	
● ALERT	

What is so powerful about "caught" commands is that the developer does not need to know the exact location of the code that needs to be debugged. When a command is entered in this list, 4D will automatically open a Debug/Trace window **anytime** that command is executed. This can be useful for establishing a pattern to reproduce the bug; if the developer thinks the bug is related to a specific command but is not sure where the bug is occurring they can set a "catch" for that command and observe all places where the command is used.

However, it is important to reiterate that **any** execution of the commands in the Caught Commands List will open in the debugger. This can get quite overwhelming if the commands are used throughout the database. It is important to keep this in mind when using the Caught Commands List.

As with the debugger, the Caught Commands List is not available in compiled databases. Similarly, while this feature is better than the debugger for investigating patterns, it can still be cumbersome in large, complex databases where the commands being caught are executed in multiple places.

A final note: the "Watch Pane" of the Runtime Explorer supports custom expressions just as the debug/trace window. The same techniques mentioned above can be used to add custom expressions to this pane with the exception of being able to highlight text in the Source Pane since there is no Source Pane in the Runtime Explorer.

The Debug Log File

With the release of 4D 2004.3 a new feature was added called "Debug Log Recording". This feature creates a log file, managed by 4D, which can be useful for debugging. This log is essentially a log of the call chain for the database.

There is already a Technical Note that covers the use of the Debug Log File feature in-depth. Refer to Technical Note 06-01, “The 4DDebugLog.txt File”.

The primary shortcoming of this log file is the information it contains. In some cases, e.g. when the database is crashing, this log file can be useful for isolating the specific point where the database crashes (usually the last entry in the log file is where the crash occurred). At the same time the contents are not customizable and may not be what the developer really needs (hence the purpose of part 2 of this technical note about debugging with a custom log file).

Error Messages

4D can present a variety of error messages like Syntax Errors, Database Engine Errors, Network Errors, etc. This topic may seem self explanatory. Error messages are there to tell the developer when an error occurs. However it is surprising how often it is overlooked that the error message is there to tell **why** the problem occurred as well.

In terms of debugging, error messages can be extremely important and it is vital to understand what the error message means as it may lead to the solution of the problem much more quickly.

At the same time error messages can be misleading and point the developer in the wrong direction. It is, of course, important to be conscious of this. Nonetheless the error messages that 4D presents should never be ignored and care should be taken to make sure whether or not the error message seems correct and might already have isolated the bug for the developer.

Conclusion

This Technical Note explored the process of debugging as well as some of the advanced techniques available to the developer using built-in 4D features.

Part 2 of this Technical Note will present a debugging technique that can be useful for when the 4D features are not available or not ideal. This technique involves the use of text file logging and analyzing the resultant text file. A 4D component will be included that demonstrates this.