

# Foreword

This document is intended to be a general technical reference for the features and capabilities of Haiku's integrated debugger, and how to use them. This will entail walking through the different user interface features, how they work, and how best to potentially make use of them. It should be noted that most of the capabilities described herein require the target binary to be built with DWARF debugging information. If this is not the case, only a very limited subset of these capabilities will be available. Unless otherwise indicated, this document represents the most current set of capabilities available in the official Haiku tree.

Last Updated: *2015-07-25 00:01:15 -0400*

# Table of Contents

[Teams Window](#)

[Team Window](#)

[Thread List](#)

[Stack Trace View](#)

[Images](#)

[Breakpoints](#)

[Execution Control](#)

[Source View](#)

[Breakpoint condition configuration](#)

[Variables View](#)

[Registers View](#)

[Output Capture View](#)

[Main Window Menus](#)

[Team Settings Window](#)

[Signals](#)

[Images](#)

[Exceptions](#)

[Inspector Window](#)

[Editing Memory](#)

[Advanced Topics](#)

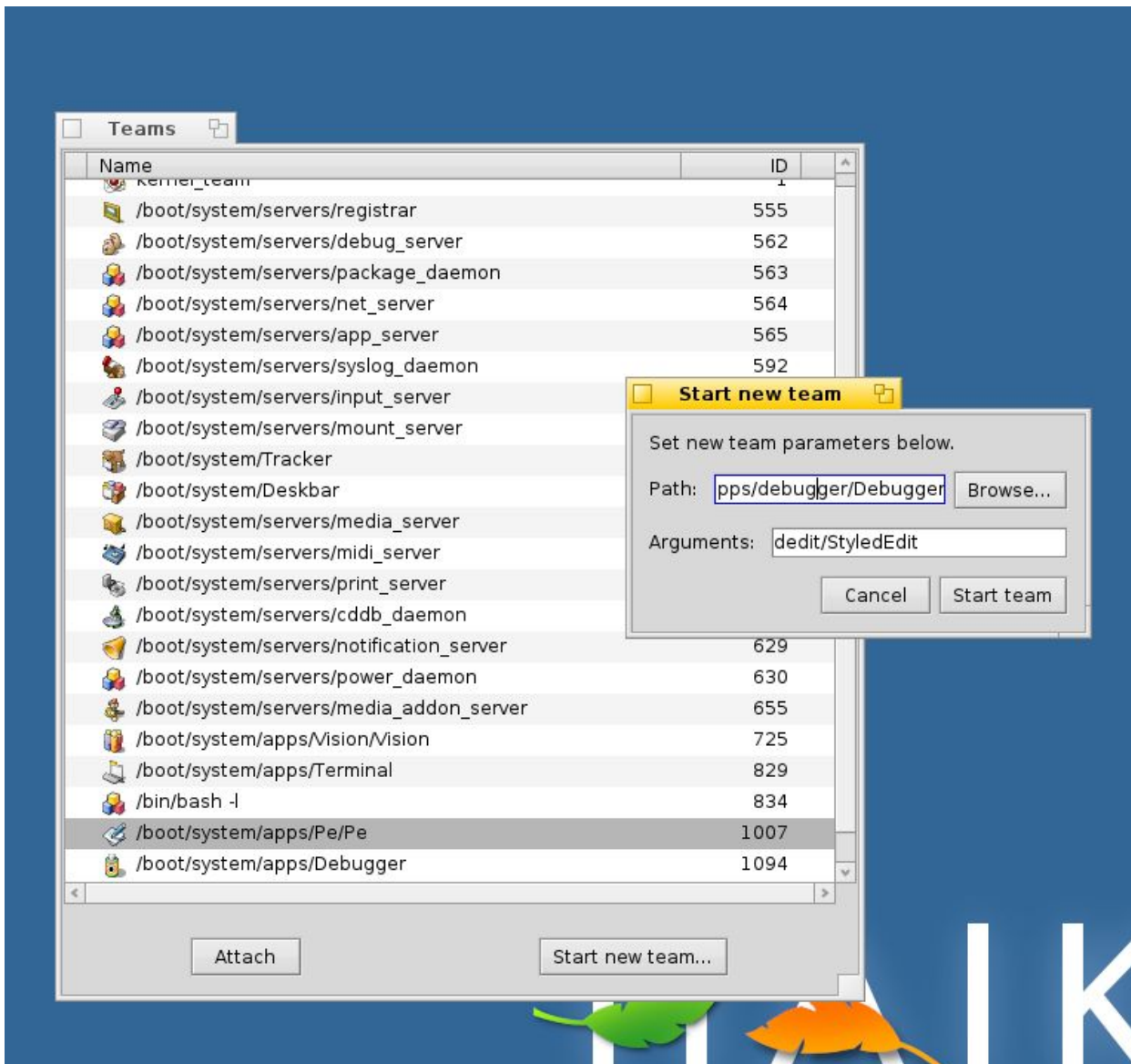
[Expression Evaluation](#)

[Special System Type Handling](#)

[Return values](#)

[Separate Debug Information Files](#)

## Teams Window



Starting the debugger without any command line arguments will cause it to start by showing you the Teams window seen above on the left. This allows you to view the teams currently running, and select/attach to one if desired. If the team in question is one that is intended to be freshly started for this debugging session, then clicking the “Start new team” button will lead to the window shown on the right, where one can choose a path to the target executable, as well as any (optional) command line arguments that should be passed in when loading it.

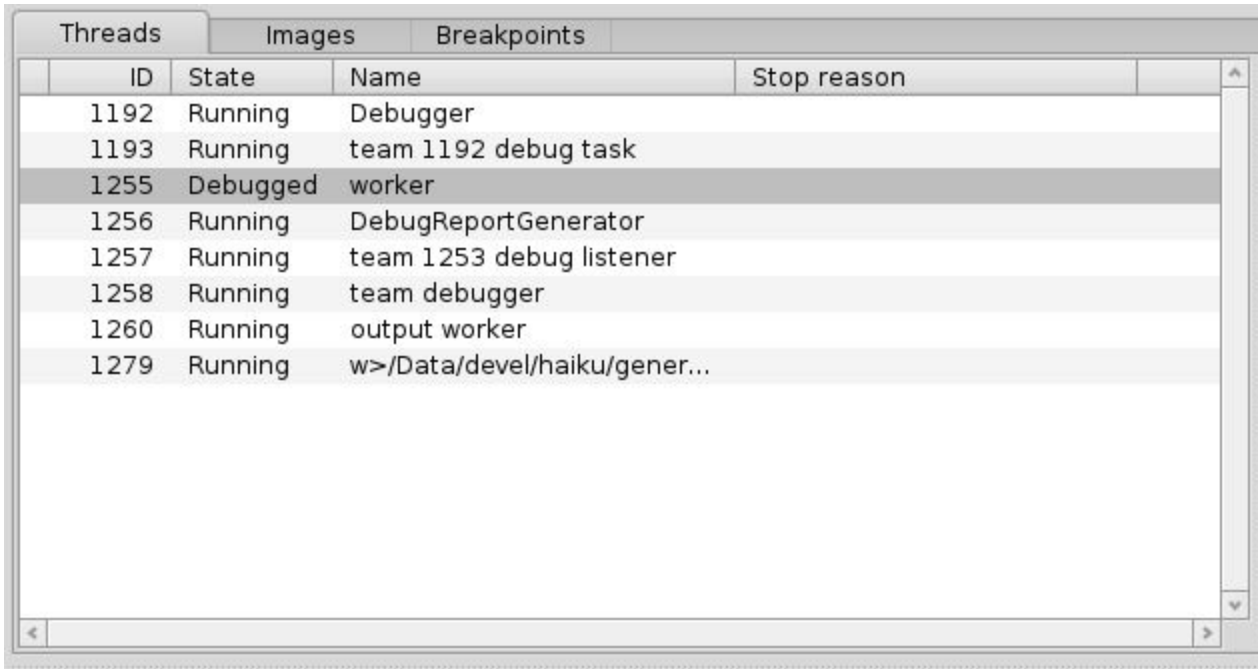
# Team Window

The screenshot displays a debugger window titled "Debugger (1192)". The interface is divided into several panes:

- Threads Pane:** Shows a list of threads with columns for ID, State, Name, and Stop reason. Thread 1255 is highlighted as "Debugged worker".
- Code Pane:** Displays C++ source code for "DwarfFile.cpp". The current execution point is at a line with a yellow highlight: `return B_BAD_VALUE;`. Other lines are also highlighted in yellow and cyan.
- Registers Pane:** Shows a table of registers with columns for Variable, Value, and Type. The "this" register is highlighted.
- Variables Pane:** Shows a table of variables with columns for Variable, Value, and Type. The "this" variable is highlighted.
- Control Panel:** Includes buttons for "Run", "Step over", "Step into", and "Step out".
- Output Pane:** Located at the bottom, it contains checkboxes for "Stdout" and "Stderr", and a "Clear" button.

After a team has been either created or attached to, a window like the above will be opened. In order to illustrate several features, this window is in a state resulting from having already started execution and stopped at a breakpoint. Here we can see most of the main functional areas of the debugger, which will now be detailed individually. The top of the window is grouped into three tabs, the first of which is the Threads tab, which exposes the following functionality.

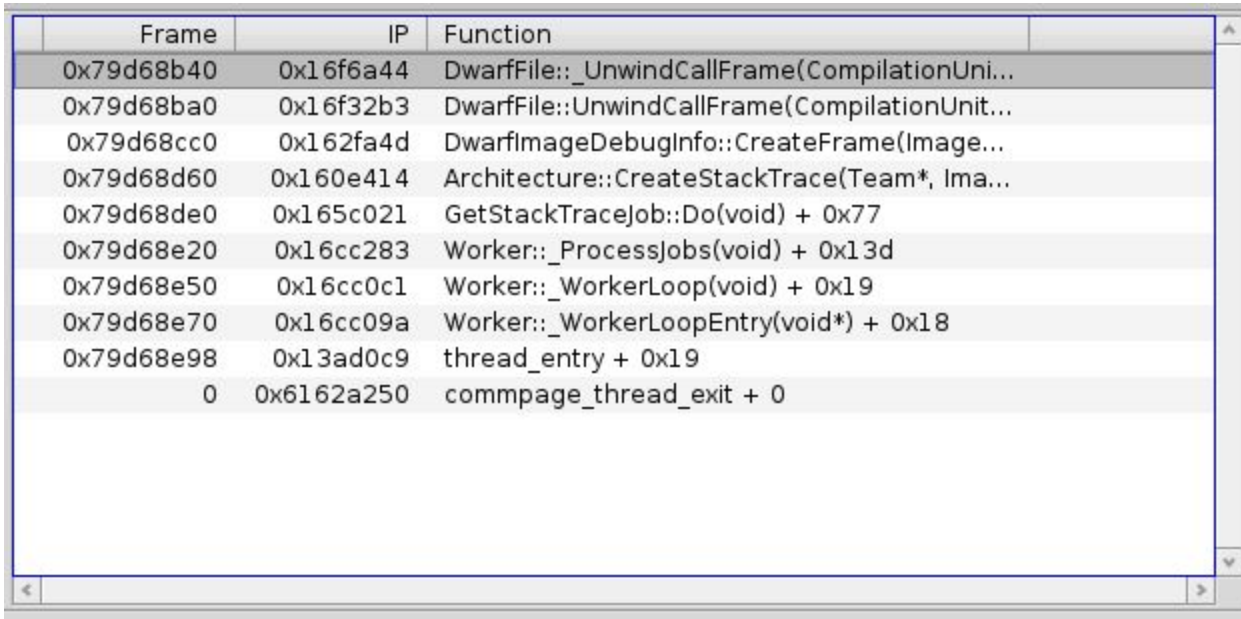
## Thread List



| ID   | State    | Name                         | Stop reason |
|------|----------|------------------------------|-------------|
| 1192 | Running  | Debugger                     |             |
| 1193 | Running  | team 1192 debug task         |             |
| 1255 | Debugged | worker                       |             |
| 1256 | Running  | DebugReportGenerator         |             |
| 1257 | Running  | team 1253 debug listener     |             |
| 1258 | Running  | team debugger                |             |
| 1260 | Running  | output worker                |             |
| 1279 | Running  | w>/Data/devel/haiku/gener... |             |

This view shows a list of the threads that currently exist in the team, as well as their respective states, which be one of Running, Debugged or Exception. Selecting a thread here establishes it as the active thread of interest, which, if the thread is not in a running state, also causes its stack trace to be made visible as well as its topmost stack frame.

## Stack Trace View

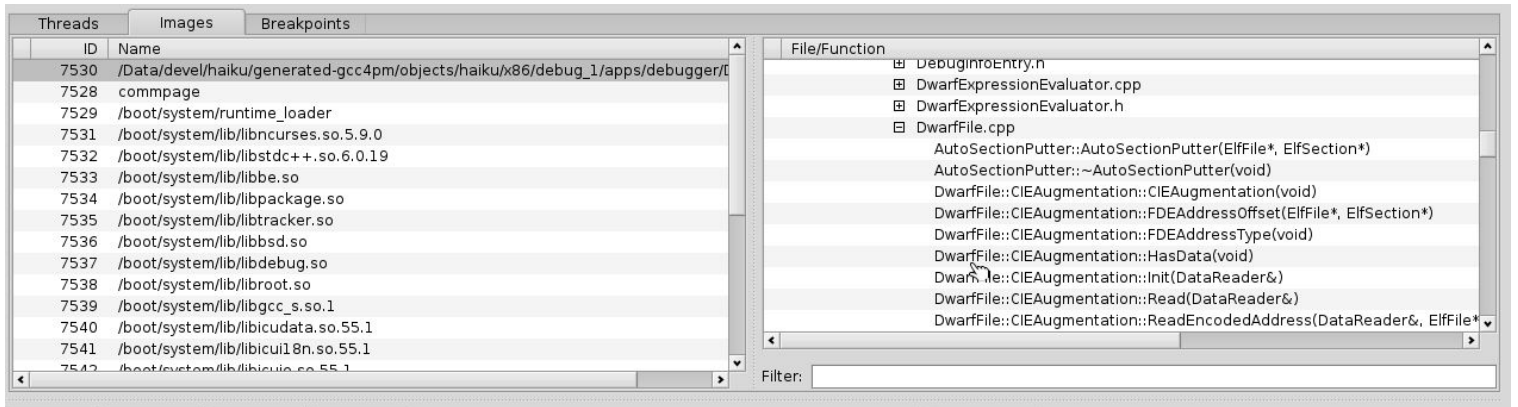


| Frame      | IP         | Function                                       |
|------------|------------|--|
| 0x79d68b40 | 0x16f6a44  | DwarfFile::_UnwindCallFrame(CompilationUni...  |
| 0x79d68ba0 | 0x16f32b3  | DwarfFile::_UnwindCallFrame(CompilationUnit... |
| 0x79d68cc0 | 0x162fa4d  | DwarfImageDebugInfo::CreateFrame(Image...      |
| 0x79d68d60 | 0x160e414  | Architecture::CreateStackTrace(Team*, Ima...   |
| 0x79d68de0 | 0x165c021  | GetStackTraceJob::Do(void) + 0x77              |
| 0x79d68e20 | 0x16cc283  | Worker::_ProcessJobs(void) + 0x13d             |
| 0x79d68e50 | 0x16cc0c1  | Worker::_WorkerLoop(void) + 0x19               |
| 0x79d68e70 | 0x16cc09a  | Worker::_WorkerLoopEntry(void*) + 0x18         |
| 0x79d68e98 | 0x13ad0c9  | thread_entry + 0x19                            |
| 0          | 0x6162a250 | commpage_thread_exit + 0                       |

This view shows a stopped thread's active stack trace, and lets you select which frame is currently of interest, which in turn will cause other views to update accordingly (i.e. the variables/registers views will switch over to that frame's state, and the source view will jump to the appropriate line of execution).

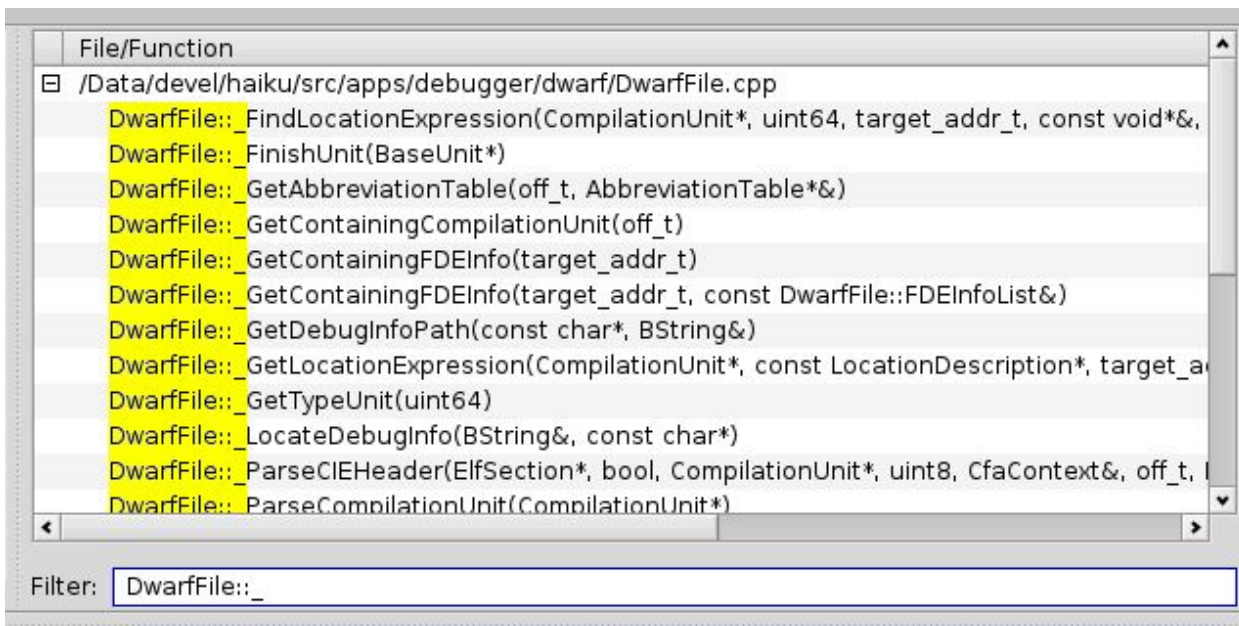
## Images

The Images Tab contains two views, allowing the user to browse the list of loaded executable images, and select functions within.

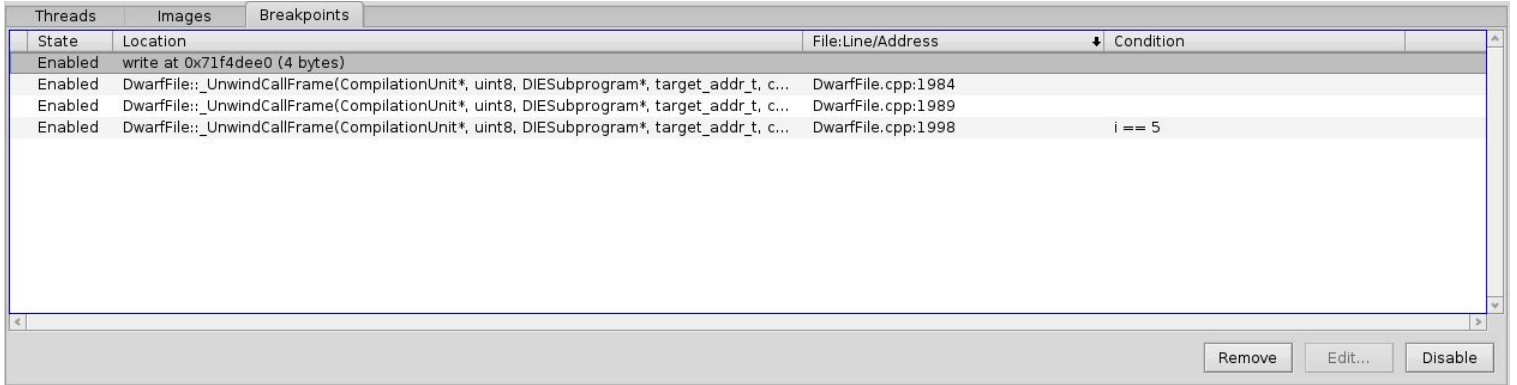


The left view contains the a list of all images currently loaded. Selecting one causes the view on the right to show a list of all functions found within that image, organized hierarchically by source location. In the case of an image without debug information available, this consists simply of a list of all symbols found in the image. Selecting a function makes it visible in the source view, where one can then e.g. set breakpoints or ask to execute to that point.

As the list of functions can often be large/unwieldy for complex code, the functions list allows a partial match filter to be applied to reduce the list to the functions of interest:



# Breakpoints



The breakpoints tab shows the list of currently installed breakpoints, their locations, and their states. Breakpoints can optionally have an expression applied to them, which causes the breakpoint to only be triggered if said expression evaluates to a non-zero result (i.e. if a particular variable in the function has a certain value, such as “i == 5”). In addition, the list will also show any installed memory watchpoints, as shown above.

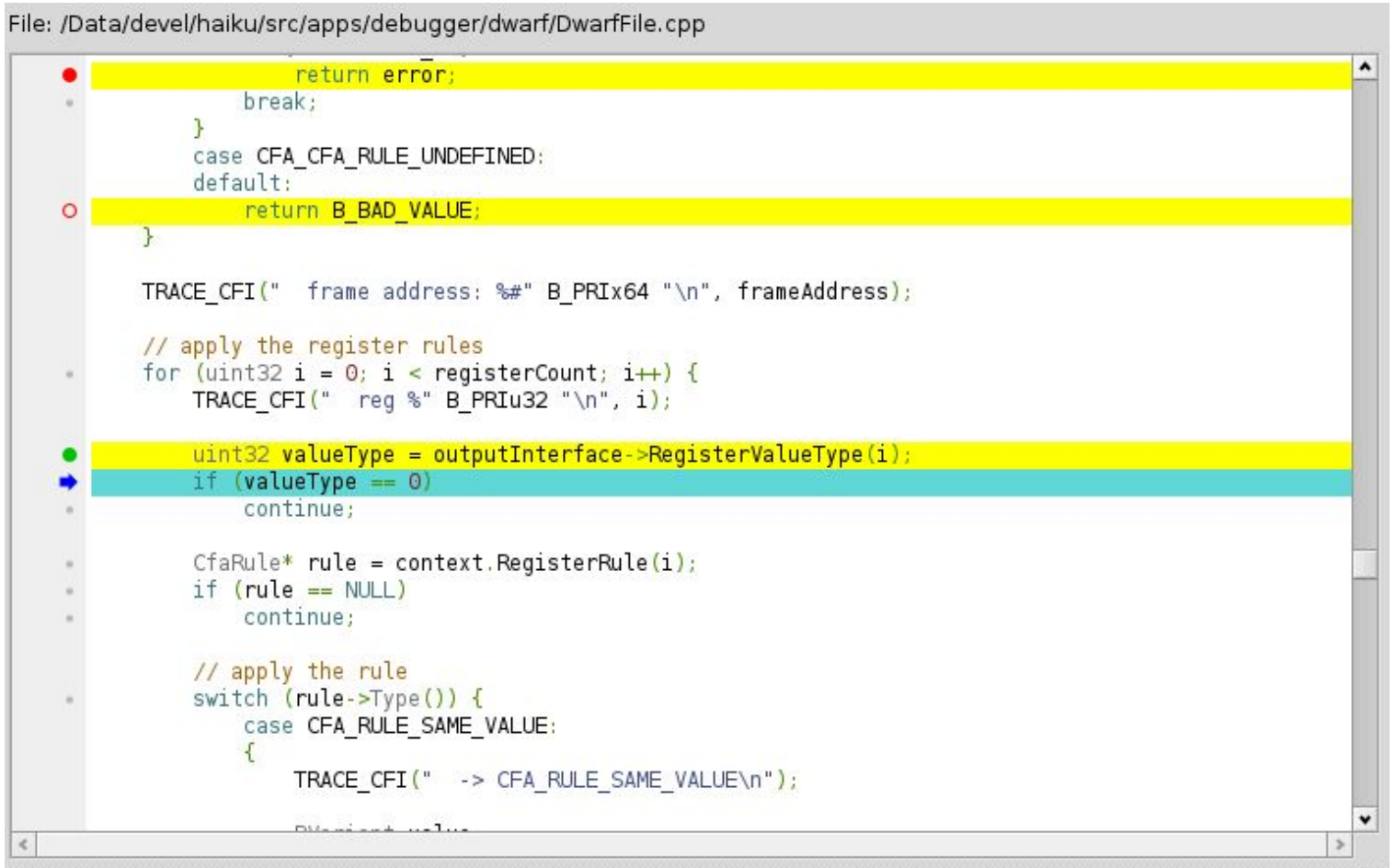


## Execution Control



If a thread is currently stopped, its execution can be manipulated via the above buttons, and their corresponding keyboard shortcuts (F5, F10, F11 and shift-F11 respectively). If the thread is running, the button corresponding to Run will instead read Debug, allowing you to halt its execution wherever it currently may be. These buttons are visible regardless of which tab is selected from the top tab group.

## Source View



The screenshot shows a source code editor window titled "File: /Data/devel/haiku/src/apps/debugger/dwarf/DwarfFile.cpp". The code is as follows:

```
    return error;
    break;
}
case CFA_CFA_RULE_UNDEFINED:
default:
    return B_BAD_VALUE;
}

TRACE_CFI(" frame address: %#" B_PRIx64 "\n", frameAddress);

// apply the register rules
for (uint32 i = 0; i < registerCount; i++) {
    TRACE_CFI(" reg %" B_PRIu32 "\n", i);




    uint32 valueType = outputInterface->RegisterValueType(i);
    if (valueType == 0)
        continue;

    CfaRule* rule = context.RegisterRule(i);
    if (rule == NULL)
        continue;

    // apply the rule
    switch (rule->Type()) {
        case CFA_RULE_SAME_VALUE:
        {
            TRACE_CFI("  -> CFA_RULE_SAME_VALUE\n");
            Divergent value
```

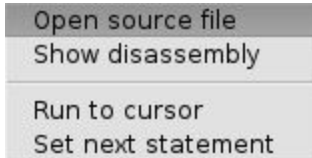
The code is annotated with several markers on the left side: a red dot (active breakpoint) on the first line, a gray dot on the second line, a red circle (disabled breakpoint) on the third line, a green dot (conditional breakpoint) on the line containing `uint32 valueType = outputInterface->RegisterValueType(i);`, and a blue arrow (execution cursor) on the line containing `if (valueType == 0)`. The line with the blue arrow is highlighted in turquoise. Other lines are highlighted in yellow.

The source view shows the currently active source code (or that which was most recently selected via either a stack trace or the image/function list). In the example pictured above, execution is currently stopped at the line highlighted in turquoise. Also pictured are the several types of supported breakpoints:

|   |   |
|---|---|
|  | Standard breakpoint. Always halts execution when encountered.   |
|  | Disabled breakpoint. While tracked and persisted in settings, it won't actually be triggered during execution while in this state. This state can be toggled by hovering over the breakpoint marker and left clicking while holding the shift key.        |
|  | Conditional breakpoint. These have an expression associated with them, and will only trigger if that expression evaluates to a non-zero result at the point when they're encountered. Conditions can be configured by right clicking a breakpoint marker. |

Left-clicking any of the dotted gray line markers on the left allows a breakpoint to be installed, while clicking an existing breakpoint causes that breakpoint to be removed.

The source view also has some functionality accessible by right clicking anywhere in the actual text area:

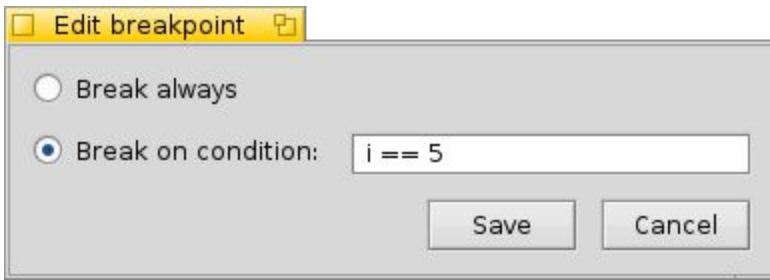


As its name implies, the “Open source file” menu item opens the currently visible source file in the preferred editor configured in the system file types. “Show disassembly” allows one to switch to a view of the low level disassembly version of the function which is currently selected, as this can sometimes be useful.

The remaining two items allow manipulation of execution. “Run to cursor” indicates that the current thread should continue execution until it encounters the source line at which the cursor was right clicked. “Set next statement” on the other hand, allows one to force the current instruction pointer to the line in question. For example, if one steps over a line of code, and the evaluated result/effect on local variables is not what was expected, this can be used to step the instruction pointer back to that line and execute the function again, perhaps stepping into it to evaluate what happened in further detail (assuming the function hasn’t already modified something which will affect its behavior when being executed again). Note that this functionality should generally be used with great care, and only by as few lines as necessary, as the stack pointer may be modified by some lines, especially when switching in and out a scope block; crossing such a boundary could potentially cause program execution to crash or otherwise exhibit undefined behavior due to the stack pointer no longer being correct.

It should also be noted that the top row of the source view which displays the location of the current source file potentially has an additional function: in the case where debug information is available, but the source file cannot be found at the specified location, the description will instead display a prompt to the user asking them to locate the file in question. Clicking it will first attempt a BFS query for matching file names, and display any matches in a pop up menu. In addition, there will also be a menu item allowing manual location via a file panel in case none of the found items are correct (as may be the case for development partitions with no indices). In the case where no matches are found at all, the file panel will be opened directly with no intermediate menu.

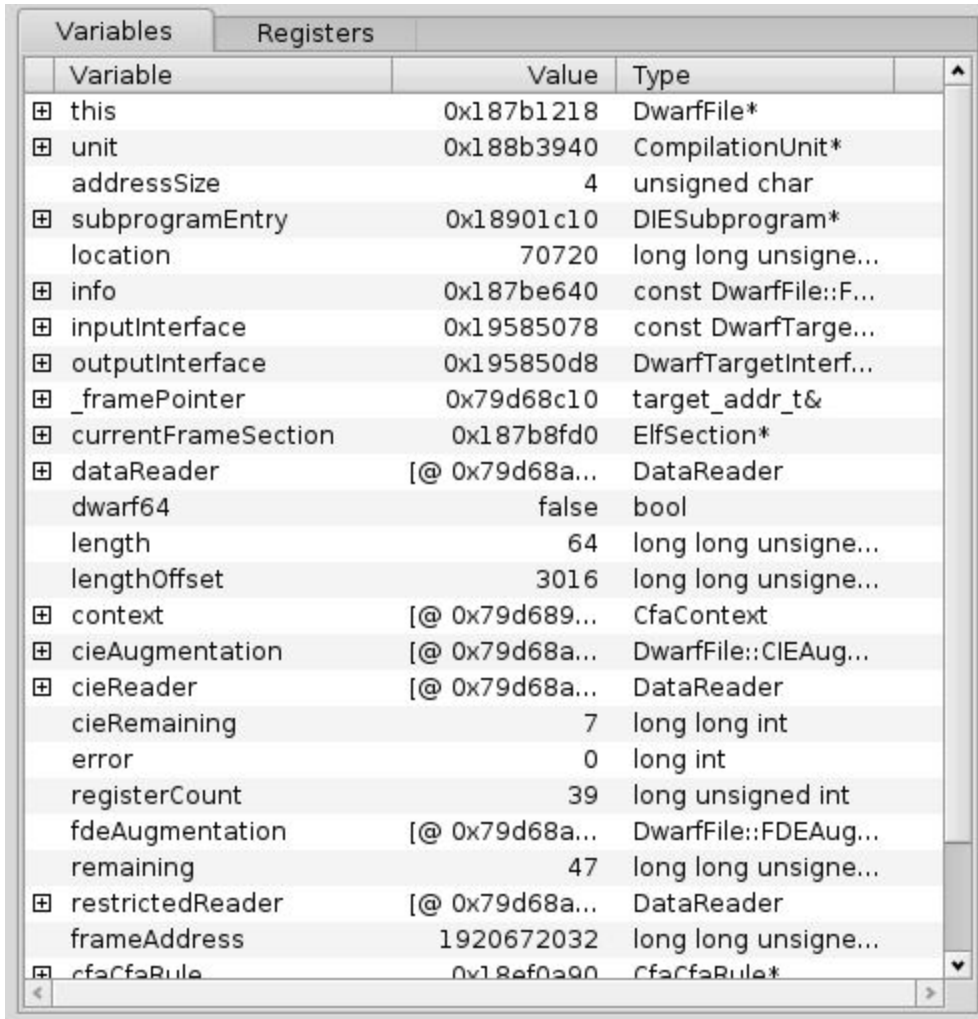
## Breakpoint condition configuration



As mentioned in the breakpoint types table above, right clicking a breakpoint allows configuring it for conditional execution, which is done via the window pictured above. Here, one can either enable and specify a condition to apply to said breakpoint, or revert it back to an ordinary unconditional breakpoint. The condition takes the form of an expression, as detailed in the advanced topics section.

## Variables View

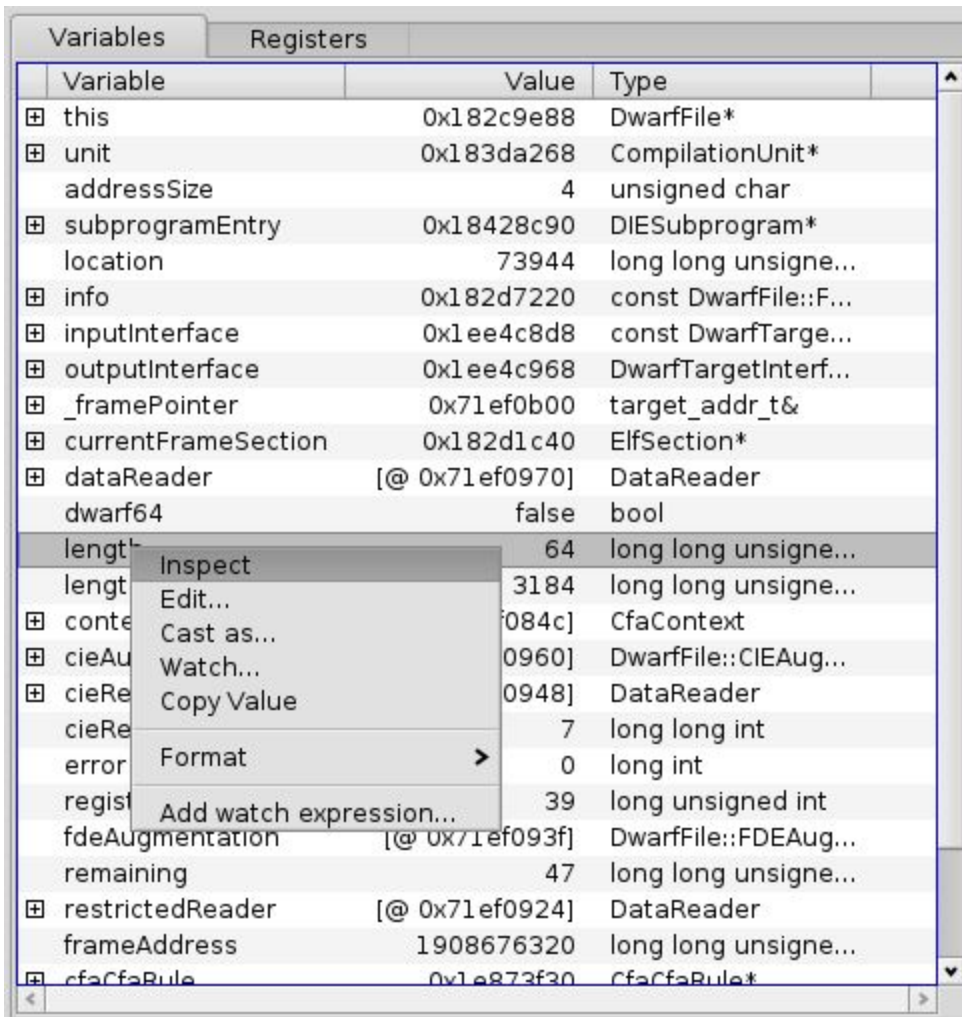
This view shows the variables that are currently in scope where the selected thread/function is stopped. If no thread is currently stopped, no variables will be displayed.



| Variable  | Value          | Type                  |
|---|----------------|-----------------------|
| <input checked="" type="checkbox"/> this                | 0x187b1218     | DwarfFile*            |
| <input checked="" type="checkbox"/> unit                | 0x188b3940     | CompilationUnit*      |
| addressSize   | 4              | unsigned char         |
| <input checked="" type="checkbox"/> subprogramEntry     | 0x18901c10     | DIESubprogram*        |
| location  | 70720          | long long unsigne...  |
| <input checked="" type="checkbox"/> info                | 0x187be640     | const DwarfFile::F... |
| <input checked="" type="checkbox"/> inputInterface      | 0x19585078     | const DwarfTarge...   |
| <input checked="" type="checkbox"/> outputInterface     | 0x195850d8     | DwarfTargetInterf...  |
| <input checked="" type="checkbox"/> _framePointer       | 0x79d68c10     | target_addr_t&        |
| <input checked="" type="checkbox"/> currentFrameSection | 0x187b8fd0     | ElfSection*           |
| <input checked="" type="checkbox"/> dataReader          | [@ 0x79d68a... | DataReader            |
| dwarf64   | false          | bool                  |
| length  | 64             | long long unsigne...  |
| lengthOffset  | 3016           | long long unsigne...  |
| <input checked="" type="checkbox"/> context             | [@ 0x79d689... | CfaContext            |
| <input checked="" type="checkbox"/> cieAugmentation     | [@ 0x79d68a... | DwarfFile::CIEAug...  |
| <input checked="" type="checkbox"/> cieReader           | [@ 0x79d68a... | DataReader            |
| cieRemaining  | 7              | long long int         |
| error   | 0              | long int              |
| registerCount   | 39             | long unsigned int     |
| fdeAugmentation   | [@ 0x79d68a... | DwarfFile::FDEAug...  |
| remaining   | 47             | long long unsigne...  |
| <input checked="" type="checkbox"/> restrictedReader    | [@ 0x79d68a... | DataReader            |
| frameAddress  | 1920672032     | long long unsigne...  |
| <input checked="" type="checkbox"/> cfaCfaRule          | 0x18ef0a90     | CfaCfaRule*           |

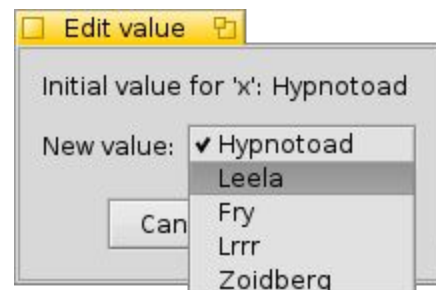
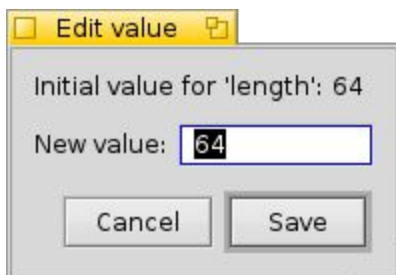
As pictured, the variables are shown with their name, current value and type. In the case of pointer or compound types such as structs/classes/arrays, they can additionally be expanded to show their data members. When stepping a thread, any variables whose values have changed in between steps will be highlighted accordingly.

Variables can also be right-clicked to select various options:



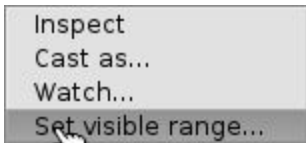
The “Inspect” item opens the debugger’s memory inspector at the memory location of the selected variable (to be described in more detail later).

If the selected variable is in a writable location, the “Edit” item is presented, which opens a window allowing one to edit its respective value. The type of editor presented will vary depending on the type of variable. Integer and floating point values will present a free form editor allowing the user to type a new value as seen on the left, while restricted types such as enums or booleans will present an editor as seen on the right:

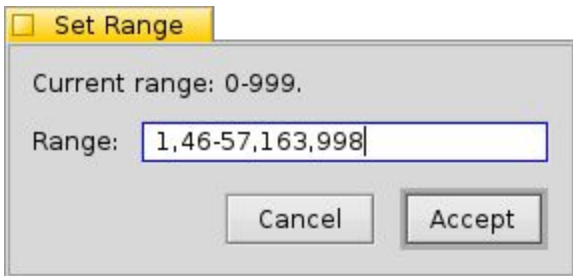


The “Cast as” item brings up a window allowing one to specify a different type to treat the currently selected variable as. For instance, if the current function takes a parameter which is a pointer to a base interface class of some form, but the user knows that the actual instance that was passed in is a specific subclass, this allows the debugger to view the variable as that subclass, and consequently show any additional members that otherwise wouldn't be known.

If the selected variable is an array, then an additional menu item is presented, allowing one to control the range of elements which is made visible:

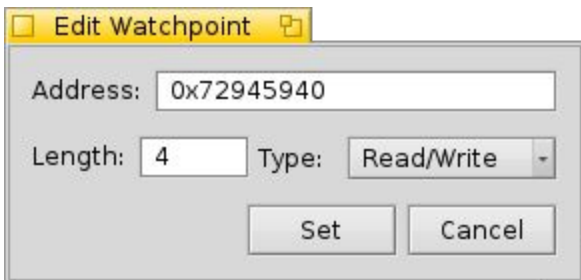


This can be handy since, e.g. a large array may only have a few elements that are actually of interest at present, and hunting for said elements in a list that could potentially encompass thousands of items is typically rather tedious and error-prone. Clicking said item presents you with the following dialog:



The top line of text indicates the overall bounds of the array from which one can select elements. In this particular case, the selected array contains 1000 elements overall. The range list accepts a comma-separated set of either ranges of the form x-y, or individual element indices, as seen above. After clicking Accept, the variables view will only display the ranges specified for the variable in question.

The “Watch” item allows the user to install a watchpoint at the memory address at which the selected variable is located. These are similar to breakpoints, but rather than being triggered when a particular line of code is executed, they instead cause execution to be halted when the corresponding memory location is accessed as configured:



As can be seen here, the watchpoint can be triggered either via a read or a write, and one can specify how many bytes away from the current address can be written to while still triggering the halt.

The “Copy Value” item, as its name implies, simply copies the value of the current variable to the clipboard.

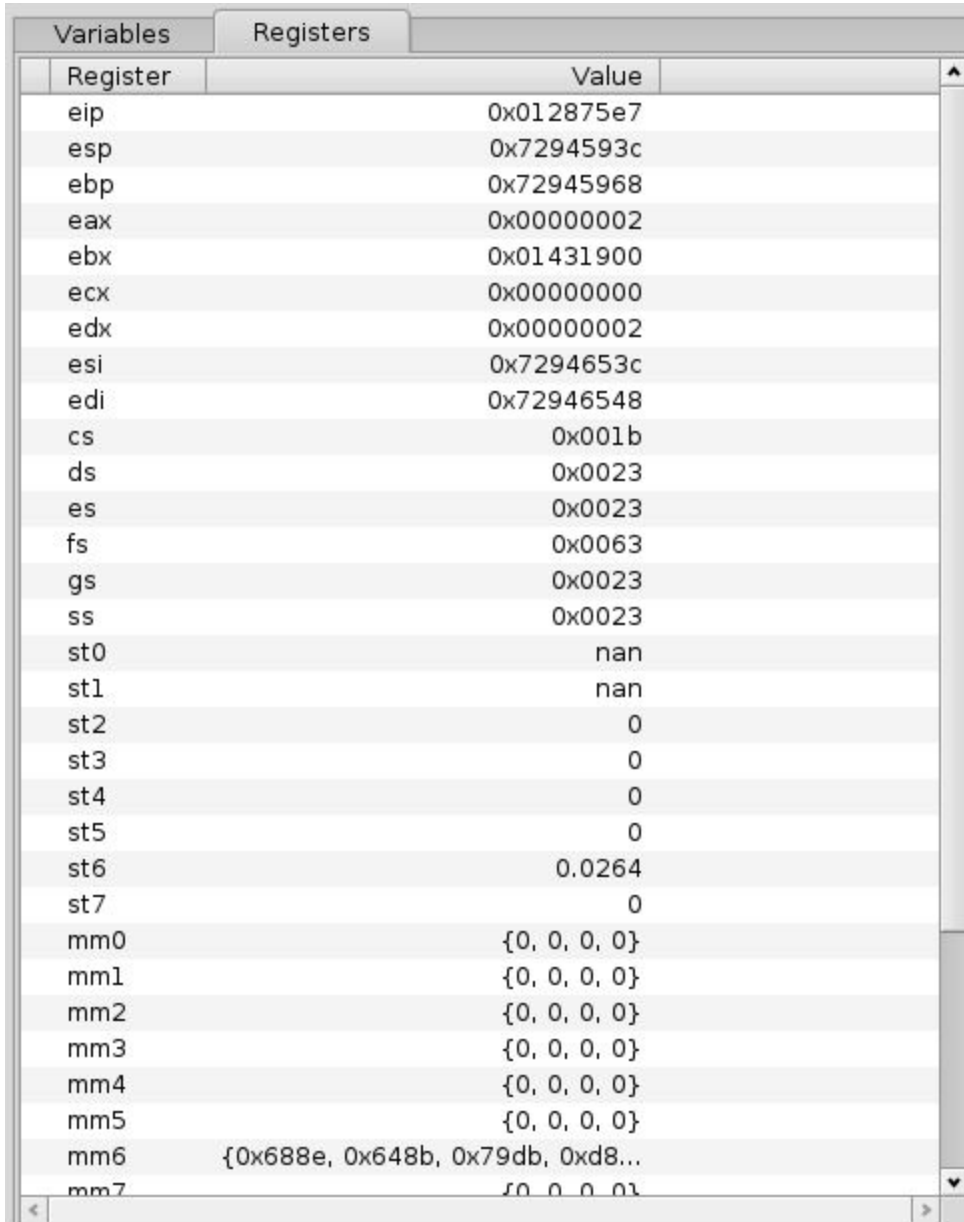
The “Format” submenu allows one to specify an alternative format to view the current variable as. For integers, this allows to switch between signed, unsigned and hexadecimal formats.

The “Add Watch Expression” item allows the user to add a variable to the current function that actually consists of an expression that is re-evaluated at each step. Unlike in the case of conditional breakpoints, the expressions here can be of an arbitrary type.



## Registers View

The registers view shows the values of the CPU's registers at the current stack frame. Three types of registers are supported, depending on the target CPU: integer, floating point, and vector. In the screenshot below, all three types are visible.

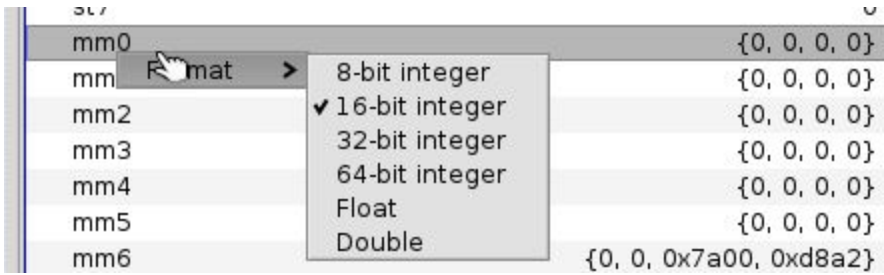


The screenshot shows a debugger window with two tabs: 'Variables' and 'Registers'. The 'Registers' tab is active, displaying a table of CPU registers and their values. The table has two columns: 'Register' and 'Value'. The registers are listed in the following order: eip, esp, ebp, eax, ebx, ecx, edx, esi, edi, cs, ds, es, fs, gs, ss, st0, st1, st2, st3, st4, st5, st6, st7, mm0, mm1, mm2, mm3, mm4, mm5, mm6, and mm7. The values are hexadecimal for integer registers, floating-point for st registers, and vector for mm registers.

| Register | Value                             |
|----------|-----------------------------------|
| eip      | 0x012875e7                        |
| esp      | 0x7294593c                        |
| ebp      | 0x72945968                        |
| eax      | 0x00000002                        |
| ebx      | 0x01431900                        |
| ecx      | 0x00000000                        |
| edx      | 0x00000002                        |
| esi      | 0x7294653c                        |
| edi      | 0x72946548                        |
| cs       | 0x001b                            |
| ds       | 0x0023                            |
| es       | 0x0023                            |
| fs       | 0x0063                            |
| gs       | 0x0023                            |
| ss       | 0x0023                            |
| st0      | nan                               |
| st1      | nan                               |
| st2      | 0                                 |
| st3      | 0                                 |
| st4      | 0                                 |
| st5      | 0                                 |
| st6      | 0.0264                            |
| st7      | 0                                 |
| mm0      | {0, 0, 0, 0}                      |
| mm1      | {0, 0, 0, 0}                      |
| mm2      | {0, 0, 0, 0}                      |
| mm3      | {0, 0, 0, 0}                      |
| mm4      | {0, 0, 0, 0}                      |
| mm5      | {0, 0, 0, 0}                      |
| mm6      | {0x688e, 0x648b, 0x79db, 0xd8...} |
| mm7      | {0, 0, 0, 0}                      |

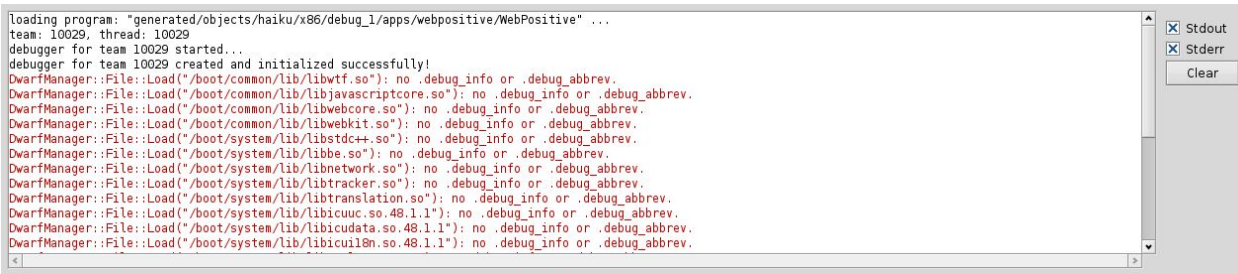
Right-clicking a register brings up a context menu that allows various options to be selected, depending on the type of register. For integer registers, this comprises an Inspect item which, as in the case of variables, brings up the inspector, using the value of the current register as the target memory address.

For vector registers such as mm0-mm7 above, the context menu allows one to specify the type of packed values currently stored in the register, i.e. the various sizes of integer, or floating point types, as this can vary depending on which instructions are currently being used to manipulate them, as shown below:



## Output Capture View

The output capture view displayed at the bottom of the team window serves the purpose of showing console output from the target program, as shown below:

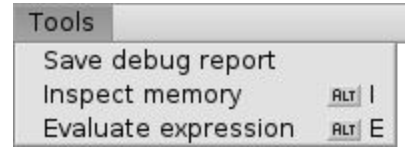
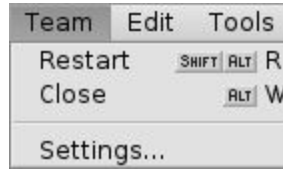
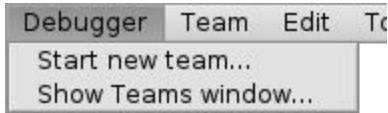


```
loading program: "generated/objects/haiku/x86/debug_1/apps/webpositive/WebPositive" ...
team: 10029, thread: 10029
debugger for team 10029 started...
debugger for team 10029 created and initialized successfully!
DwarfManager::File::Load("/boot/common/lib/libutf.so"): no .debug_info or .debug_abbrev.
DwarfManager::File::Load("/boot/common/lib/libjavascriptcore.so"): no .debug_info or .debug_abbrev.
DwarfManager::File::Load("/boot/common/lib/libwebcore.so"): no .debug_info or .debug_abbrev.
DwarfManager::File::Load("/boot/common/lib/libwebkit.so"): no .debug_info or .debug_abbrev.
DwarfManager::File::Load("/boot/system/lib/libstdc++.so"): no .debug_info or .debug_abbrev.
DwarfManager::File::Load("/boot/system/lib/libbe.so"): no .debug_info or .debug_abbrev.
DwarfManager::File::Load("/boot/system/lib/libnetwork.so"): no .debug_info or .debug_abbrev.
DwarfManager::File::Load("/boot/system/lib/libtracker.so"): no .debug_info or .debug_abbrev.
DwarfManager::File::Load("/boot/system/lib/libtranslation.so"): no .debug_info or .debug_abbrev.
DwarfManager::File::Load("/boot/system/lib/libicuuc.so.48.1.1"): no .debug_info or .debug_abbrev.
DwarfManager::File::Load("/boot/system/lib/libicudata.so.48.1.1"): no .debug_info or .debug_abbrev.
DwarfManager::File::Load("/boot/system/lib/libicu10n.so.48.1.1"): no .debug_info or .debug_abbrev.
```

Captured output is colored based on which descriptor it came from ; black represents stdout, while red is used for stderr. Which descriptors the view shows output from can be individually toggled as needed, depending on what is currently of interest. Furthermore, if the target program is known to emit no console output of interest, the view can be collapsed entirely via the splitter above it.

## Main Window Menus

Some functionality is also accessible via the menu bar, as shown below:

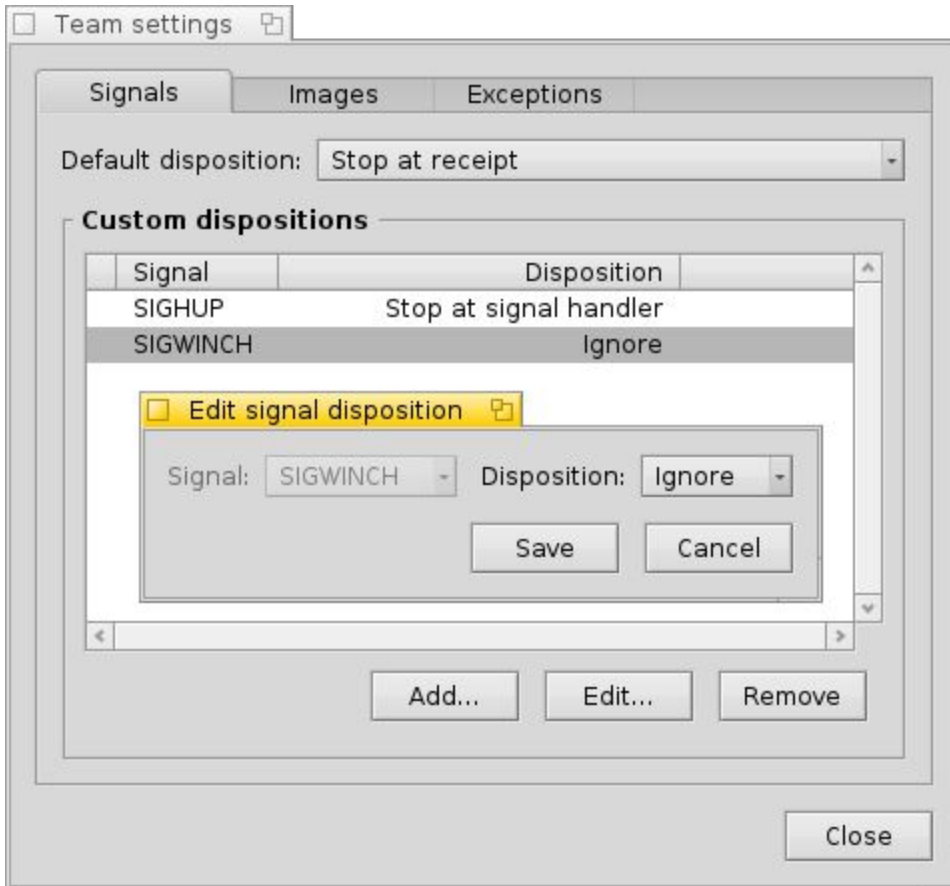


|                     |   |
|---------------------|---|
| Start new team      | This item offers the same functionality as the similarly named button previously described in the Teams window. The new team will be started in a separate debugger window from the current one.  |
| Show Teams window   | As its name implies, this brings the previously described Teams window up, in case an encountered situation might require attaching to another running team.  |
| Restart             | Resets the currently attached team into an initial state as if it had just been freshly launched.   |
| Close               | Terminates the current debugger window. If the team in question is still running, a dialog will ask if the debugged team should also be terminated or simply detached, or the close request can be ignored entirely in case invoked accidentally. |
| Settings            | Opens the team settings window. This allows one to adjust image-related break conditions, signal behavior, and language-specific settings. For more details, please see the Team Settings window section of this guide.                           |
| Save debug report   | Saves a report detailing the current team's state. This is functionally identical to the reports that can be saved when a program crashes otherwise.  |
| Inspect memory      | Opens the memory inspector window. This allows one to view and edit the contents of memory in the target team. For more details, please see the Inspector window section of this guide.   |
| Evaluate expression | Brings up the expression evaluation window. This allows one to evaluate the results of expressions in a C-like syntax. For more details, please see the expression evaluation section in Advanced Topics.   |

# Team Settings Window

The settings window, which is accessible from the Team menu, allows one to adjust several different types of advanced settings, which are currently split into three categories.

## Signals



The signals tab allows the configuration of how the debugger responds to signal-related events in the target team. Three possible dispositions can be configured:

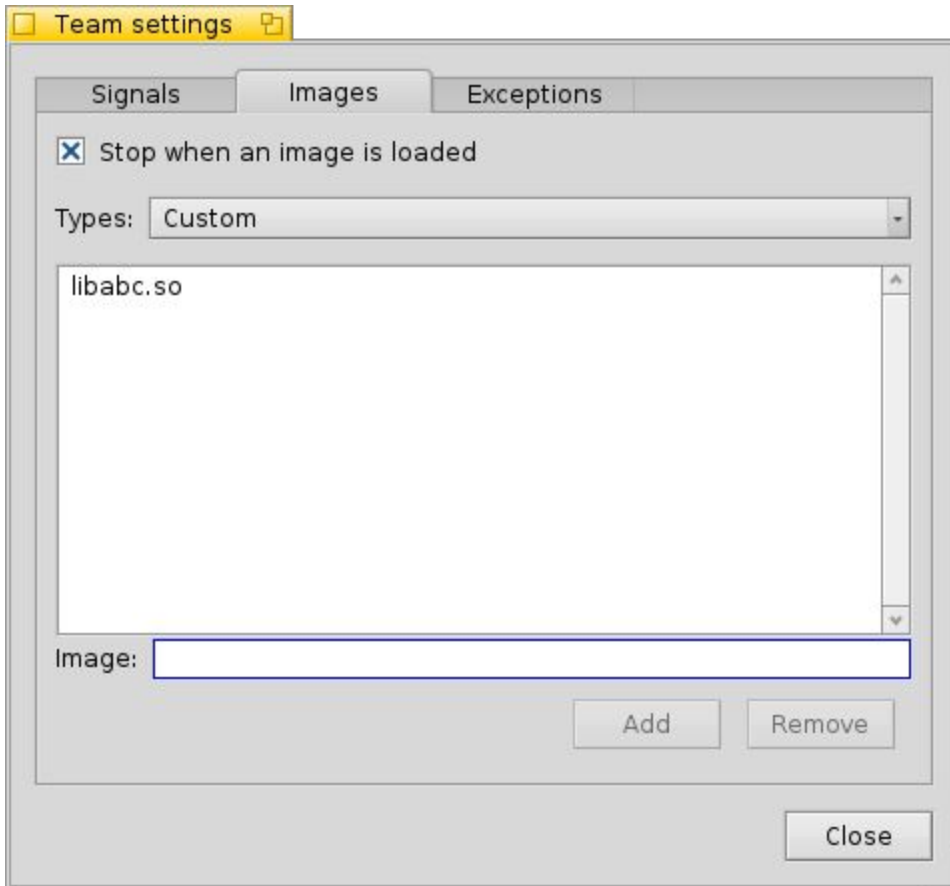
|                        |  |
|------------------------|--|
| Ignore                 | This is the default behavior, and essentially indicates that execution should proceed as usual when a signal is received. Note that certain types of critical signals will result in execution being halted regardless if the application does not handle them in some way, such as SIGSEGV. |
| Stop at receipt        | This causes execution to be halted immediately when the signal is slated to be delivered to the target team. The thread's stop status in the Threads tab will indicate the signal that was received.   |
| Stop at signal handler | If the target team has installed a signal handler for the signal in question via <code>signal()</code> or <code>sigaction()</code> , this causes execution to stop at that handler. Note that if no handler has been installed, the behavior will fall back to that of the "Stop at          |

|  |                  |
|--|------------------|
|  | receipt” option. |
|--|------------------|

The “Default signal disposition” menu field allows the global default to be configured from the above options. In addition, the list in the lower half of the tab allows per-signal overrides to be set up in addition to the global default. This is useful if, for instance, one is only interested in one specific signal, or if most signals are desired to stop execution, but a specific one may frequently occur, but is uninteresting. Choosing to add or edit a signal brings up a window allowing one to choose both the target signal, and the desired disposition. In the case of editing an existing row, the choice of signal will be locked to that of the chosen row.

## Images

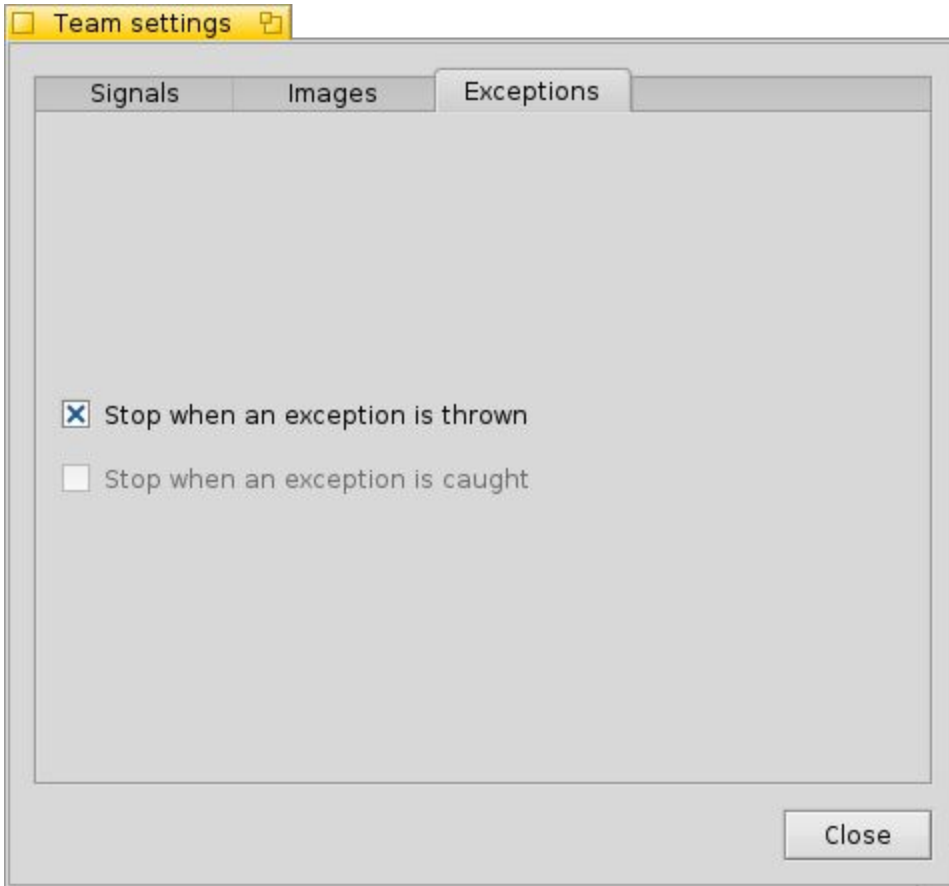
The Images tab allows one to specify that execution should be halted when new executable images are loaded into the target team's address space, typically via the `load_add_on()` system call. This allows the user to access functions within the newly loaded executable image, and configure breakpoints before any of its code is actually executed. By default, this behavior is unconditional, which is to say the debugger stops for any image load. However, the Types menu allows one to also choose to filter the images, so only those matching particular names cause a halt, which can be handy for programs that load many libraries dynamically, but only a handful are of interest:



In this case, one can add or remove image filenames to the list, which in turn causes threads to stop only if a newly loaded image's name matches.

## Exceptions

The exceptions tab is specific to the C++ language, and may not necessarily always be visible in the future, when support for other languages has been added.

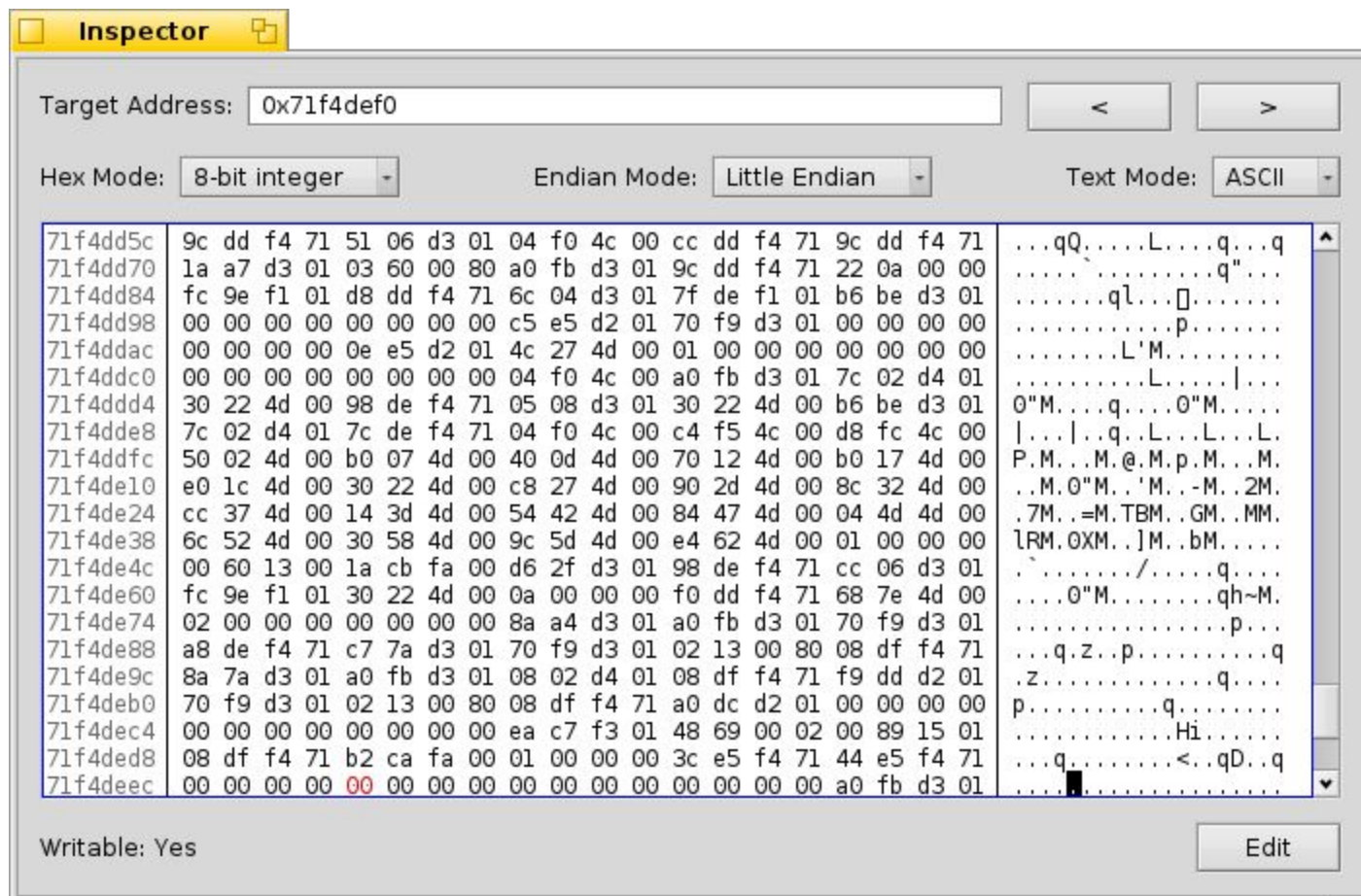


Currently, this only allows configuring the debugger to halt execution when a C++ exception is thrown, though in the future there is planned to be support for halting execution at the point where an exception is caught as well.



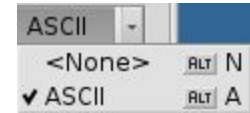
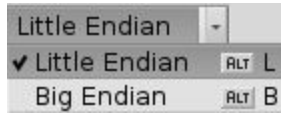
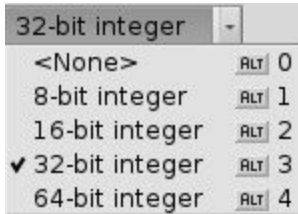
## Inspector Window

Choosing to inspect a variable/register, or invoking the aforementioned “Inspect memory” menu item brings up the memory inspector, whose primary purpose is to allow one to look at the contents of the target team’s memory. This can be useful when troubleshooting various kinds of bugs involving what appears to be data corruption, as seeing the full contents of the relevant block of memory can potentially reveal patterns or clues that might point the way to the culprit. An example of the window is pictured below:

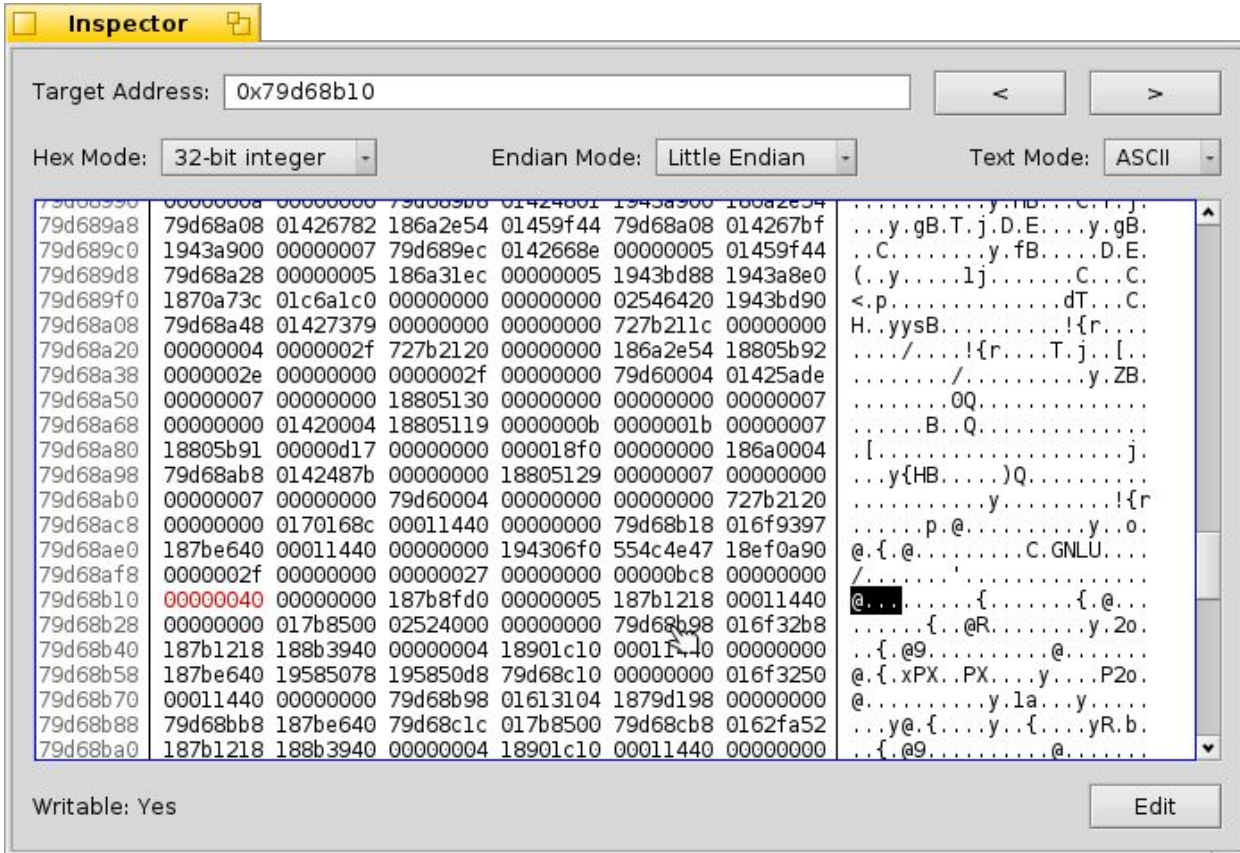


The inspector loads memory one block at a time, where a block is currently specified to be the size of a hardware page (4KB on x86 systems). As can be seen above, the actual memory view is divided into two sections, one of which shows the contents in a hexadecimal format. On the far left is the starting memory address of the each respective line. The target address within the block, as seen in the text control at the top of the window, is highlighted in red in this view. In this particular case, the address in question actually corresponds to the “length” variable that was highlighted in the Variables view section earlier in this guide, and as can be seen, its value of 64 (aka hex 0x40) is located at the target address as expected.

The smaller section on the right of the view shows the same data in printable ASCII format where possible. If a memory value does not correspond to a printable character, it is replaced with a ‘.’. Various aspects of the hex and text sections of the view can be configured via the menu fields shown immediately above it:



The hex mode menu allows one to choose between hiding the hex view entirely, or adjusting the hex block size displayed. Previously, the view was in 8-bit mode. Selecting 32-bit as shown above results in the following display instead:



This functionality is useful if one is looking at specific data types of a well known size, such as pointer addresses.

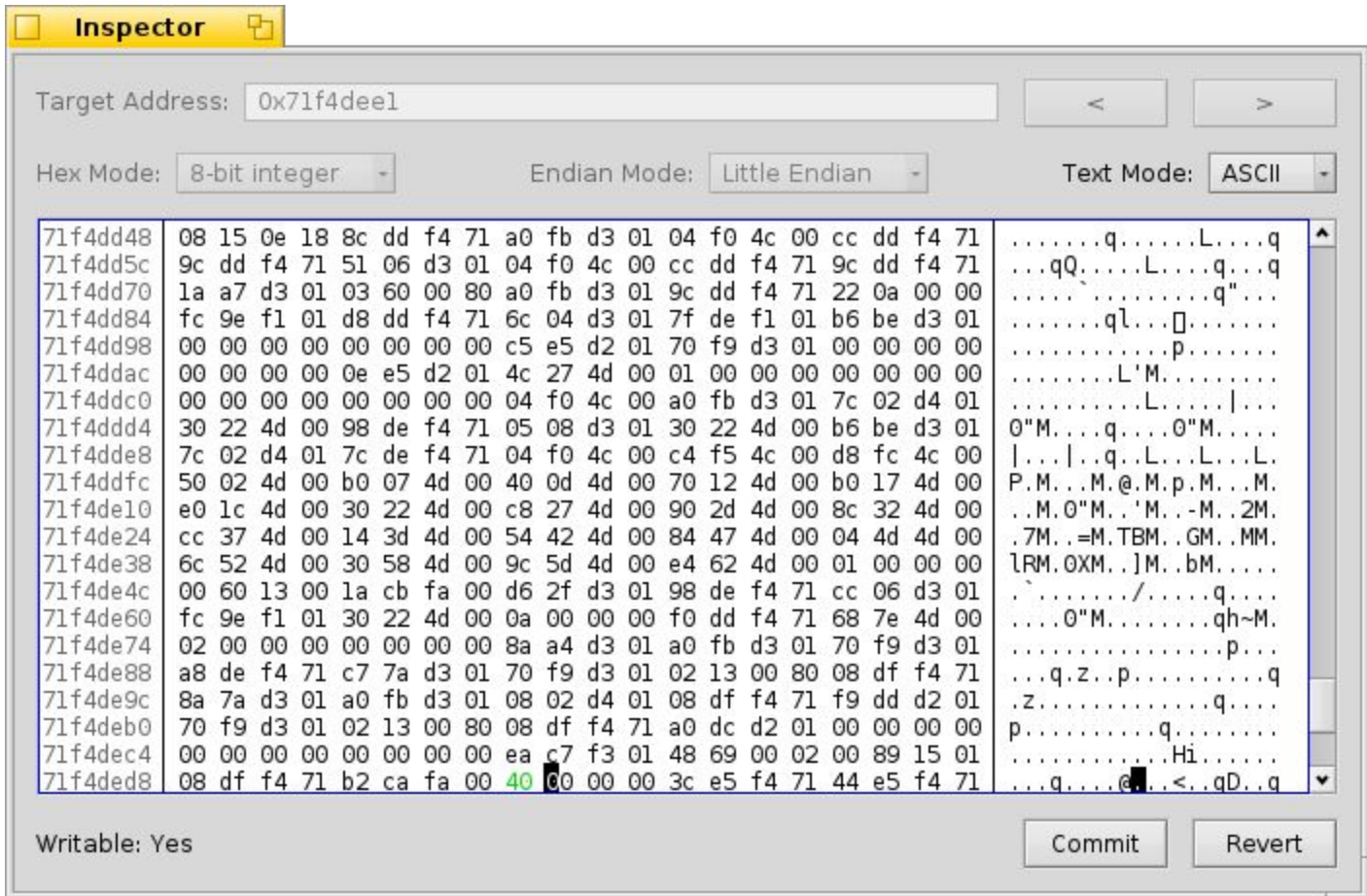
The endian mode menu lets one choose whether to view the data in big or little-endian format. By default, this chooses the architecture's native endian orientation (in the case of x86{-64}, little endian as seen here), but in some circumstances one may be viewing data that's stored in memory in an alternative format. The most commonly encountered example of this that one may see in ordinary application code is when dealing with network code. Frequently, internet addresses and other network-related pieces of information are specified as being stored in big endian format. On x86, this would mean that a raw memory view of them would cause their bytes to be in reversed order from the CPU's native format, resulting in the value being completely different from what one would expect. Switching orientation allows the data to be reinterpreted such that it appears as its normal value.

Finally, the text mode menu allows one to show/hide the text section of the display. This may be extended in the future to allow interpreting the data in a wide character format such as UTF-16 or the various extended 8-bit character codepages.

The controls along the top of the window allow one to manipulate the location of memory currently being shown. The target address field allows explicitly specifying a new target address as either a plain numerical address or a mathematical expression. The left and right buttons adjacent to it allow navigating to the next or previous memory block. It should also be noted that when the memory view has input focus, the usual arrow, page and home/end keys can be used as an alternative means to navigate within the current block.

## Editing Memory

The final piece of functionality afforded by the inspector window is to actually edit the contents of the target block, assuming the address in question points to writable memory in the target team. As an example of memory where this would not be the case, most executable code such as the shared libraries or the executable image of the target itself are usually mapped as read-only. In the lower left of the window, one can see an indicator which specifies whether the current block is writable or not, as is the case here. If so, one can use the Edit button in the lower right to switch the inspector into editing mode, which results in it adjusting accordingly as seen below:



Here, we have modified the value that was stored at the target address, and the window will highlight all changes made. The Edit button has been replaced with the Commit and Revert buttons, which allow one to either write the changes back to the target team once satisfied, or to simply discard all changes and leave the block in the state it was in before edit mode was enabled.

## Advanced Topics

Several more advanced features are also available in the debugger, which are described in further detail in this section.

### Expression Evaluation

Several of the features described in this guide require the use of what is referred to as an expression, such as conditional breakpoints. These effectively consist of a statement in a subset of the target language that results in a value. For basic mathematical operations, all the C/C++ mathematical expressions are supported, as are most of the bit operations, with the exception of bit shifts. Parentheses to enforce precedence are likewise supported. This subset of functionality can be handy if circumstances require one to perform some simple calculations to determine e.g. a memory offset to inspect. Note that variables in the context of the current stack frame can also be used in such an expression.

In addition to basic mathematical expressions, however, the expression evaluator can also handle expressions more complex types, as well as various operations upon them. For instance, if the variable in question is a pointer, one can dereference it as one would in the actual language. Furthermore, if the pointer refers to a structure or class, one can refer to member variables within said objects as one normally would, i.e. via the `.` and `->` operators. Typecasts are likewise supported. When used in the context of a conditional breakpoint, this allows one to construct breakpoints that, for example, only trigger if the value of a particular member of an object within the target function matches a specific value (or, alternatively, does not match an expected value). For the conditional breakpoint case, the evaluation is concerned primarily with whether the value returned is true or false. Note that an expression evaluation failure, i.e. due to a syntax error, or a failure retrieving variable value will always cause the breakpoint to trigger as a safety precaution.



Outside of the context of conditional breakpoints, such expressions can also directly return objects of arbitrary types. When using the expression evaluation window or a watch expression in the variables view, such evaluation results are displayed as additional rows in the list of variables. This can have a number of uses: for example, one may know the memory address of a variable that one wishes to keep track of, but which isn't visible in the context of the current stack frame, such as an application object. In such a case, one can add a watch expression which casts said memory address to the appropriate type, and consequently have it appear in the variables view with each step, as demonstrated below:

| Frame      | IP        | Function                                   |
|------------|-----------|--|
| 0x7a4f8080 | 0x16eb9d3 | StyledEditWindow::MessageReceived(BMess... |
| 0x7a4f80a0 | 0x26d2227 | BLooper::DispatchMessage(BMessage*, BH...  |
| 0x7a4f84a0 | 0x27bc1cf | BWindow::DispatchMessage(BMessage*, BH...  |
| 0x7a4f8500 | 0x27c033a | BWindow::task_looper(void) + 0x326         |
| 0x7a4f8530 | 0x26d3606 | BLooper:: task0 (void*) + 0x3a             |

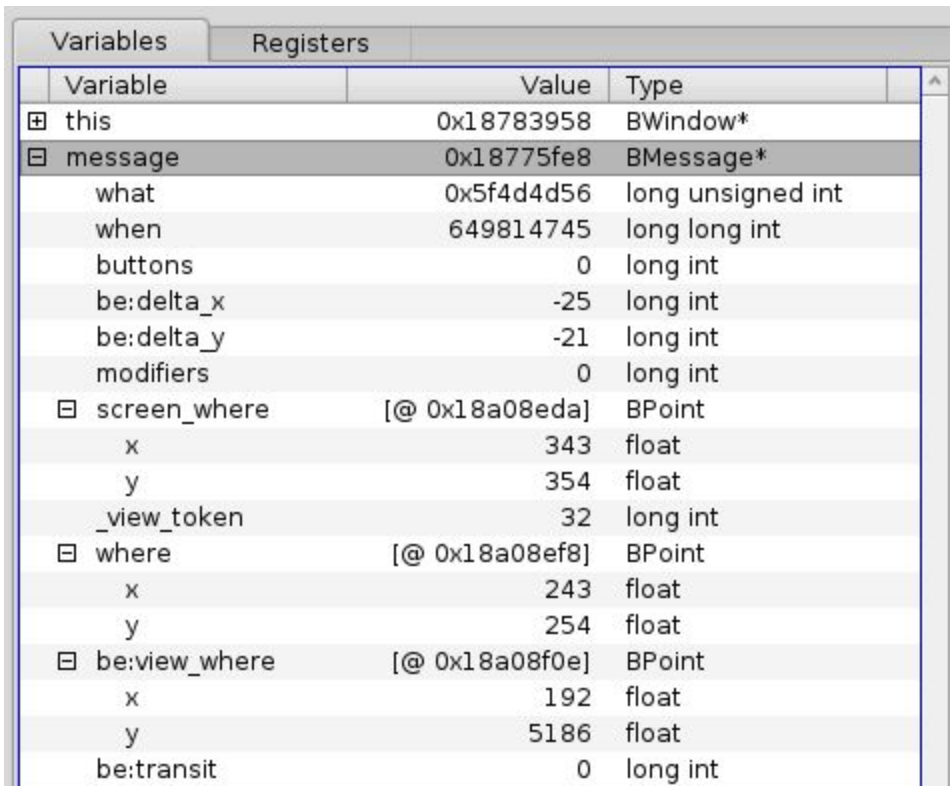
| Variables  |                | Registers   |  |
|--|----------------|-------------|--|
| Variable   | Value          | Type        |  |
| <input checked="" type="checkbox"/> this                       | 0x1826ea48     | StyledEd... |  |
| <input checked="" type="checkbox"/> message                    | 0x1835b840     | BMessa...   |  |
| <input checked="" type="checkbox"/> (StyledEditApp*)0x7206b0f0 | 0x7206b0f0     | StyledEd... |  |
| <input checked="" type="checkbox"/> BApplication               | [@ 0x7206b0f0] | BApplica... |  |
| <input checked="" type="checkbox"/> fOpenPanel                 | 0x18250f48     | BFilePan... |  |
| <input checked="" type="checkbox"/> fOpenPanelEncodingMenu     | 0x18280cf8     | BMenu*      |  |
| <input checked="" type="checkbox"/> fOpenAsEncoding            | 0              | long uns... |  |
| <input checked="" type="checkbox"/> fWindowCount               | 1              | long int    |  |
| <input checked="" type="checkbox"/> fNextUntitledWindow        | 2              | long int    |  |
| <input checked="" type="checkbox"/> fBadArguments              | false          | bool        |  |

## Special System Type Handling

Several API-provided classes are used quite heavily throughout the system. As a consequence, one will quite frequently run into situations where it can be useful during the course of debugging to be able to see the contents of one which is currently being dealt with. This is in fact possible, provided the target application has been loaded with a debug build of libbe. In future stable Haiku releases, this will be more easily provided via a debug information package, but as of the moment this requires building it yourself from the Haiku source tree.

Currently, the supported classes for this feature include [BMessage](#), [BList](#) and [BObjectList](#). Under normal circumstances, debug information would merely show you the low level class structure, as with other structs within your own program. For these types however, the debugger is capable of reconstructing them from the target team's memory, and instead showing their contents.

As an example, the screenshot below illustrates this feature for the contents of the B\_MOUSE\_MOVED message which is used to trigger the [BView::MouseMoved\(\)](#) API call:



The screenshot shows a debugger's 'Variables' window with a tree view of a BMessage object. The 'message' variable is expanded to show its fields: what, when, buttons, be:delta\_x, be:delta\_y, modifiers, screen\_where (with sub-fields x and y), \_view\_token, where (with sub-fields x and y), and be:view\_where (with sub-fields x and y). The 'be:transit' field is also visible at the bottom.

| Variable        | Value          | Type              |
|-----------------|----------------|-------------------|
| ⊞ this          | 0x18783958     | BWindow*          |
| ⊞ message       | 0x18775fe8     | BMessage*         |
| what            | 0x5f4d4d56     | long unsigned int |
| when            | 649814745      | long long int     |
| buttons         | 0              | long int          |
| be:delta_x      | -25            | long int          |
| be:delta_y      | -21            | long int          |
| modifiers       | 0              | long int          |
| ⊞ screen_where  | [@ 0x18a08eda] | BPoint            |
| x               | 343            | float             |
| y               | 354            | float             |
| _view_token     | 32             | long int          |
| ⊞ where         | [@ 0x18a08ef8] | BPoint            |
| x               | 243            | float             |
| y               | 254            | float             |
| ⊞ be:view_where | [@ 0x18a08f0e] | BPoint            |
| x               | 192            | float             |
| y               | 5186           | float             |
| be:transit      | 0              | long int          |

For the case of the BList and BObjectList classes, similar behavior is used to present those respective classes as if they were arrays, as shown below:

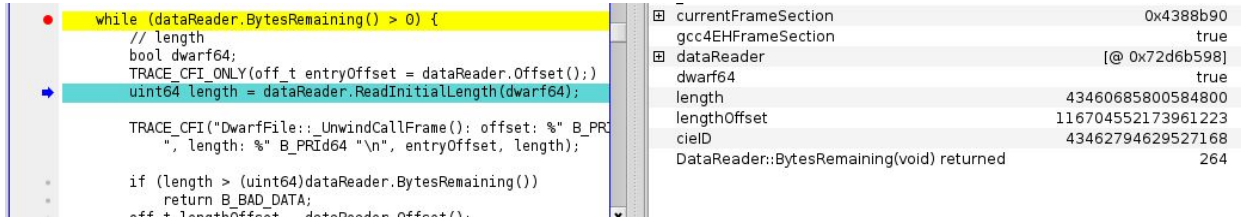
| Variables          |                   | Registers                             |  |
|--------------------|-------------------|---------------------------------------|--|
| Variable           | Value             | Type                                  |  |
| [-] this           | 0x195cac50        | CompoundValueNode*                    |  |
| [-] ValueNode      | [@ 0x195cac50]    | ValueNode                             |  |
| [-] fType          | 0x191a0200        | CompoundType*                         |  |
| [-] fChildren      | [@ 0x195cac74]    | BObjectList<CompoundValueNode::Child> |  |
| Capacity           | 7                 | long int                              |  |
| [-] [0]            | 0x1960c610        | CompoundValueNode::Child*             |  |
| [-] ValueNodeChild | [@ 0x1960c610]    | ValueNodeChild                        |  |
| [-] fParent        | 0x195cac50        | CompoundValueNode*                    |  |
| [-] fName          | [@ 0x1960c62c]    | BString                               |  |
| fPrivateData       | "BApplication"    | char*                                 |  |
| [-] [1]            | 0x1960c5e0        | CompoundValueNode::Child*             |  |
| [-] [2]            | 0x1960c5b0        | CompoundValueNode::Child*             |  |
| [-] [3]            | 0x1960c580        | CompoundValueNode::Child*             |  |
| [-] ValueNodeChild | [@ 0x1960c580]    | ValueNodeChild                        |  |
| [-] fParent        | 0x195cac50        | CompoundValueNode*                    |  |
| [-] fName          | [@ 0x1960c59c]    | BString                               |  |
| fPrivateData       | "fOpenAsEncoding" | char*                                 |  |
| [-] [4]            | 0x1960c550        | CompoundValueNode::Child*             |  |
| [-] [5]            | 0x1960c520        | CompoundValueNode::Child*             |  |
| [-] [6]            | 0x1960c4f0        | CompoundValueNode::Child*             |  |

As can be seen here, for a BObjectList, the contained type information is available, and as such, the individual array elements are exposed as their actual type. In the case of BList, however, this is not known, so the latter simply exposes an array of pointers. Given that the user knows what the type actually is however, typecasting can be used on the individual elements in order to present them appropriately.



## Return values

When stepping through code, it is not uncommon to step over a statement which contains a nested function call returning a value used by the remainder of the statement. As such, it would be nice to be able to see the results of such calls as one steps over the statement rather than having to step into each one individually in order to determine this. The debugger can in fact support this capability, as illustrated below:



```
while (dataReader.BytesRemaining() > 0) {
    // length
    bool dwarf64;
    TRACE_CFI_ONLY(off_t entryOffset = dataReader.Offset();)
    uint64 length = dataReader.ReadInitialLength(dwarf64);

    TRACE_CFI("DwarfFile::UnwindCallFrame(): offset: %" B_PRI
              ", length: %" B_PRId64 "\n", entryOffset, length);

    if (length > (uint64)dataReader.BytesRemaining())
        return B_BAD_DATA;
    off_t lengthOffset = dataReader.Offset();
}
```

|   |                    |
|---|--------------------|
| currentFrameSection                       | 0x4388b90          |
| gcc4EHFrameSection                        | true               |
| dataReader                                | [@ 0x72d6b598]     |
| dwarf64                                   | true               |
| length                                    | 43460685800584800  |
| lengthOffset                              | 116704552173961223 |
| cielD                                     | 43462794629527168  |
| DataReader::BytesRemaining(void) returned | 264                |

Note that this capability is currently only supported for full function calls ; calls that have been inlined are not yet able to be handled in this way.

## Separate Debug Information Files

The ELF binary format used by Haiku allows for the possibility of an executable's debug information file to be stored in a separate file from the executable itself. This capability is supported by the debugger as well, and under normal circumstances will be handled transparently. If, however, the debug information file cannot be located, then the user may be prompted for its location. In the event that the executable in question comes from a package, it is potentially possible that the debug information is available in a separate package as well. This is currently the case for the command line tool 'sed' and several other packages in the HaikuPorts repository. Should such a situation be detected, before prompting the user, the debugger will first search the package repository to see if a matching debug information package can be found. If so, the user will instead be prompted if they wish to install said package, and the situation will be handled automatically from there.