Qualcomm Technologies, Inc.

Qualcomm® Hexagon™ LLDB Debugger

User Guide

80-N2040-31 Rev. C

November 6, 2020

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

Figures

Tables

1 Introduction

This document describes a debugger for the Qualcomm® Hexagon™ processor architecture. It is based on the LLDB debugger that was developed as part of the LLVM project. The debugger works with the Hexagon software development tools and utilities to provide a complete programming system for developing high-performance software.

The LLDB debugger runs on the Windows and Linux platforms.

NOTE: The debugger is commonly referred to as LLDB.

1.1 Features

The LLDB debugger offers the following features:

- High performance
- Efficient memory use
- Extensible (using Python)
- Multi-threaded debugging support
- Remote debugging
- Supports C and C++

1.2 Conventions

`Courier` font is used for computer text and code samples, for example, `hexagon_<function_name>()`.

The following notation is used to define command syntax:

- Square brackets enclose optional items, for example, `[label]`.
- **Bold** indicates literal symbols for example, `[comment]`.
- The vertical bar character, `|`, indicates a choice of items.
- Parentheses enclose a choice of items for example, `(add|del)`.
- An ellipsis, `. . .` , follows items that can appear more than once.

1.3 Technical assistance

For more information on the LLDB debugger, see http://llvm.org/.

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/.

If you do not have access to the CreatePoint website, register for access or send email to qualcomm.support@qti.qualcomm.com.

2 Get started

This chapter assumes you are familiar with the GDB debugger commands.

2.1 Start the debugger

To start LLDB from a command line, enter:

`hexagon-lldb` *`program`*

Where `program` is the name of an executable Hexagon file.

Once started, LLDB displays a command line:

```
(lldb)
```

To perform a debug command, type the command name into the command line and press **Return**.

After the command is performed, the command line reappears. It will continue to reappear after every debug command, until you perform a command that exits LLDB.

2.2 Debug commands

Compared to the GDB command set, which is rather free-form, the LLDB command syntax is structured. All commands have the following form:

```
noun verb [-options [option-value]] [argument [argument...]]
```

The command line parsing is performed before the command execution, so it is uniform across all commands. The syntax for basic commands is quite simple: arguments, options, and option values are all separated by white spaces, and double quotes are used to protect white spaces in an argument. To put a backslash or double quote character in an argument, prefix it with a backslash in the argument. This makes the command syntax more regular, but it also means you might need to quote some arguments in LLDB that you would not in GDB.

2.2.1 Command arguments

LLDB attempts to reduce the number of special purpose argument parsers that are required, which sometimes forces you to be more explicit when specifying a command.

For instance, to set a breakpoint in GDB, enter the following to break at line 12 of `foo.c`:

```
(gdb) break foo.c:12
```

And enter the following to break at the function `foo`:

```
(gdb) break foo
```

The argument parser that GDB uses to distinguish `foo.c:12` from `foo` from `foo.c::foo` (which specifies the function `foo` in the file `foo.c`) is complex. As a result, cases exist in GDB where it is virtually impossible to specify the function you want to break on, especially when using C++.

By comparison, LLDB commands are more verbose but also more precise, and they allow for intelligent auto-completion.

To set the same file and line breakpoint in LLDB, you can enter either of the following commands:

```
(lldb) breakpoint set --file foo.c --line 12
(lldb) breakpoint set -f foo.c -l 12
```

To set a breakpoint on a function named foo, you can enter either of the following commands:

```
(lldb) breakpoint set --name foo
(lldb) breakpoint set -n foo
```

You can specify the `--name` option multiple times to set a breakpoint on a group of functions. This is convenient because it enables you to set common conditions or commands without specifying them multiple times. For example:

```
(lldb) breakpoint set --name foo --name bar
```

2.2.2 Breakpoints

In LLDB, you can set a breakpoint at a function using a method name. For instance, to set a breakpoint on all C++ methods named `foo`, you can use either of the following commands:

```
(lldb) breakpoint set --method foo
(lldb) breakpoint set -M foo
```

To limit any breakpoints to a specific executable image, use the command option, `--shlib path` (or `-s path` for short). For example:

```
(lldb) breakpoint set --shlib foo.dylib --name foo
(lldb) breakpoint set -s foo.dylib -n foo
```

You can repeat the `--shlib` option to specify several shared libraries.

2.2.3 Completion

Like GDB, the LLDB command interpreter performs a shortest unique string match on command names, so the following two commands will both execute the same command:

```
(lldb) breakpoint set -n "-[SKTGraphicView alignLeftEdges:]"
(lldb) br s -n "-[SKTGraphicView alignLeftEdges:]"
```

LLDB also supports command completion for source file names, symbol names, file names, and so on. Press the **Tab** key to initiate completion.

NOTE: Tab completion is not supported in Windows.

Individual options in a command can have different *completers*. For example, in the `breakpoint` command, `--file` completes to source files, `--shlib` completes to currently loaded shared libraries, and so on.

You can also do things like this: if you specify `--shlib` and are completing on `--file`, LLDB will list only the source files in the shared library specified by `--shlib`.

2.2.4 Help

The individual LLDB commands are extensively documented. Use the `help` command to get an overview of which commands are available, or to obtain details on specific commands.

There is also an `apropos` command, which will search the help text for all commands for a specified word, and then dump a summary help string for each matching command.

In some cases the LLDB help system can provide information on the individual arguments of a command. For example:

```
(lldb) help break command add
Add a set of commands to a breakpoint, to be executed whenever the
breakpoint is hit.
Syntax: breakpoint command add <cmd-options> <breakpt-id>
etc...
```

When you see command arguments specified with angle brackets in the command syntax (such as `<breakpt-id>` in the example above), this indicates a common argument type that you can get further help on:

```
(lldb) help <breakpt-id>
<breakpt-id> -- Breakpoint ID's consist major and minor numbers; the
major ...
```

2.2.5 Aliases

LLDB provides a mechanism for defining aliases for commonly used commands. For example, instead of entering this command:

```
(lldb) breakpoint set --file foo.c --line 12
```

You can enter the following commands:

```
(lldb) command alias bfl breakpoint set -f %1 -l %2
(lldb) bfl foo.c 12
```

Several aliases are predefined for the commonly used commands (such as `step`, `next`, and `continue`). However, you can customize the LLDB command set in any way, and because LLDB reads the `~/.lldbinit` file at startup, you can store all your aliases in this file so they are available to you.

NOTE: Your aliases are also documented in the `help` command, so you can remind yourself which aliases you have defined.

One of the predefined aliases is a weak emulator of the GDB `break` command. This command does not try to do everything the GDB `break` command can do (for example, it does not handle `foo.c::bar`). But it mostly works, and it makes the transition easier from GDB to LLDB.

For convenience, the `break` command is aliased to `b`. But if you want to learn the LLDB command that is set natively, the `b` alias will get in the way of using the rest of the LLDB breakpoint commands.

Fortunately, if you do not like a predefined alias, you an delete it with the `unalias` command. For example:

```
(lldb) command unalias b
```

Once the command alias is freed, use the following command to run the native LLDB `breakpoint` command with just `b`:

```
(lldb) command alias b breakpoint
```

2.2.6 Raw commands

The LLDB command parser supports *raw* commands where, after any command options are stripped off, the rest of the command string is passed uninterpreted to the command. Raw commands are convenient for commands whose arguments might be a complex expression that would be painful to protect with backslashes.

For example, the `expression` command is a raw command for obvious reasons. The `help` output for a command will tell you if the command is raw or not, so you know what to expect.

When using raw commands, remember that raw commands can still have options. If your command string has any dashes in it, you must indicate that they are not option markers by putting `--` after the command name but before your command string. For example:

```
watch set expr --size 8 -- 0x3fefff00
```

2.2.7 Python interpreter

LLDB has a built-in Python interpreter, which is accessible with the `script` command. All the functionality of the debugger is available as classes in the Python interpreter. Thus, the more complex commands that in GDB you would introduce with the `define` command can be performed in LLDB by writing Python functions using the `lldb-Python` library. Then load the scripts into your running session and access them with the `script` command.

2.3 Load a program

As with GDB, you can use a single command to both start LLDB and specify the file to debug:

```
$ lldb /Projects/Sketch/build/Debug/Sketch.app
Current executable set to '/Projects/Sketch/build/Debug/Sketch.app'
(x86_64).
```

Or, you can us the `file` command to specify the file after the fact:

```
$ lldb
(lldb) file /Projects/Sketch/build/Debug/Sketch.app
Current executable set to '/Projects/Sketch/build/Debug/Sketch.app'
(x86_64).
```

2.4 Set breakpoints

To see all the options available for setting a breakpoint, use the `help breakpoint set` command. For example:

```
(lldb) breakpoint set --selector alignLeftEdges:
Breakpoint created: 1: name = 'alignLeftEdges:', locations = 1, resolved
= 1
```

You can list the breakpoints that you already set:

```
(lldb) breakpoint list
Current breakpoints:
1: name = 'alignLeftEdges:', locations = 1, resolved = 1
  1.1: where = Sketch`-[SKTGraphicView alignLeftEdges:] + 33
       at /Projects/Sketch/SKTGraphicView.m:1405, address =
0x0000000100010d5b, resolved, hit count = 0
```

Setting a breakpoint creates a *logical* breakpoint, which can resolve to one or more actual locations. For example, *break by selector* sets a breakpoint on all the methods that implement that selector in the classes in your program. Similarly, a file and line breakpoint might result in multiple locations if that file and line are inlined in different places in your code.

A logical breakpoint has an integer identifier, and its locations have an identifier within their parent breakpoint. The two identifiers are joined by the a `.` character. For example, see the `1.1` string in the example above.

Also, logical breakpoints remain *live*, so if another shared library is loaded and it has another implementation of the `alignLeftEdges:` selector, the new location is added to breakpoint `1` (in this case, a breakpoint named `1.2` is set on the newly loaded selector).

The other piece of information provided in a breakpoint listing is whether the breakpoint location was *resolved* or not. A location is resolved when the file address to which it corresponds is loaded into the program you are debugging. For example, if you set a breakpoint in a shared library that subsequently is unloaded, that breakpoint location remains but it is no longer be resolved.

GDB users should note that LLDB works like GDB does after using this command:

```
(gdb) set breakpoint pending on
```

That is, LLDB always makes a breakpoint from your specification, even if it cannot find any locations that match the specification. You can tell whether an expression was resolved by checking the locations field in `breakpoint list`.

A breakpoint is reported as `pending` when you set it, so you can more easily determine if you have made a typo (if that was the reason no locations were found):

```
(lldb) breakpoint set --file foo.c --line 12
Breakpoint created: 2: file ='foo.c', line = 12, locations = 0 (pending)
WARNING: Unable to resolve breakpoint to any actual locations.
```

You can delete, disable, set conditions on, and ignore counts either on all the locations generated by your logical breakpoint, or on one location your specification resolved to. For example, the following command prints a backtrace when the breakpoint is hit:

```
(lldb) breakpoint command add 1.1
Enter your debugger command(s). Type 'DONE' to end.
> bt
> DONE
```

By default, `breakpoint command add` accepts LLDB commands on the command line. You can also specify this explicitly by passing the `--command` option. To implement your breakpoint command using the Python script, use the `--script` option instead.

2.5 Set watchpoints

In addition to breakpoints, you can use the `help watchpoint` command to see all the commands for watchpoint manipulations.

For example, the following command watches a variable named `global` for a write operation, but it stops only if the condition `(global==5)` is true:

```
(lldb) watch set var global
Watchpoint created: Watchpoint 1: addr = 0x100001018 size = 4
    state = enabled type = w
    declare @ '/Volumes/data/lldb/svn/ToT/test/functionalities/
    watchpoint/watchpoint_commands/condition/main.cpp:12'
(lldb) watch modify -c '(global==5)'
(lldb) watch list
Current watchpoints: Watchpoint 1: addr = 0x100001018 size = 4 state =
enabled type = w
    declare @ '/Volumes/data/lldb/svn/ToT/test/functionalities/
      watchpoint/watchpoint_commands/condition/main.cpp:12'
    condition = '(global==5)'
(lldb) c
Process 15562 resuming
(lldb) about to write to 'global'...
Process 15562 stopped and was programmatically restarted.
Process 15562 stopped and was programmatically restarted.
Process 15562 stopped and was programmatically restarted.
Process 15562 stopped and was programmatically restarted.
Process 15562 stopped
* thread #1: tid = 0x1c03, 0x0000000100000ef5 a.out`modify + 21 at
main.cpp:16, stop reason = watchpoint 1
frame #0: 0x0000000100000ef5 a.out`modify + 21 at main.cpp:16
   13
   14  static void modify(int32_t &var) {
   15      ++var;
-> 16  }
   17
   18  int main(int argc, char** argv) {
   19      int local = 0;
(lldb) bt
* thread #1: tid = 0x1c03, 0x0000000100000ef5 a.out`modify + 21 at
main.cpp:16, stop reason = watchpoint 1
   frame #0: 0x0000000100000ef5 a.out`modify + 21 at main.cpp:16
   frame #1: 0x0000000100000eac a.out`main + 108 at main.cpp:25
   frame #2: 0x00007fff8ac9c7e1 libdyld.dylib`start + 1
(lldb) frame var global
(int32_t) global = 5
(lldb) watch list -v
Current watchpoints:
Watchpoint 1: addr = 0x100001018 size = 4 state = enabled type = w declare
@ '/Volumes/data/lldb/svn/ToT/test/functionalities/watchpoint/
watchpoint_commands/condition/main.cpp:12'
   condition =    '(global==5)'
   hw_index = 0  hit_count = 5     ignore_count = 0
(lldb)
```

2.6 Start your program

To launch a program in LLDB, use the `process launch` command or one of its built-in aliases. For example:

```
(lldb) process launch
(lldb) run
(lldb) r
```

NOTE: The `process launch` command automatically launches the Hexagon simulator and connects LLDB to it.

After you launch a process, your process might stop somewhere:

```
(lldb) process launch
Hexagon-sim INFO: the rev_id used in the simulation is 0x00002105 (v5a)
hexagon-sim WARNING: StartGDBserver:Setting up GDB server on port 10437
Process 1 stopped
Process 1 launched: './factorial' (hexagon)
Process 1 stopped
* thread #1: tid = 0x0001, 0x00004130 factorial`main(argc=1,
argv=0x0000b100) + 28 at factorial.c:32, stop reason = breakpoint 1.1
   frame #0: 0x00004130 factorial`main(argc=1, argv=0x0000b100) + 28 at
factorial.c:32
```

NOTE: After a process is launched, the simulator output will be mixed with the LLDB output.

2.7 Control your program

After a program is launched, it can execute until it hits a breakpoint. The basic commands for controlling execution are all variations of the `thread` command. For example:

```
(lldb) thread continue
Resuming thread 0x2c03 in process 46915
Resuming process 46915
(lldb)
```

Currently you can only operate on one thread at a time. However, in the future LLDB will support commands of the form *step over the function in thread 1*, *step into the function in thread 2*, and *continue thread 3*, and so on. When LLDB eventually supports some threads running while others are stopped, this will be particularly important.

For convenience, all the stepping commands have simple command aliases. For instance, the alias for thread continue is just the letter `c`.

The other program stepping commands are mostly the same as in GDB:

LLDB command	GDB command
`thread step-in`	`step` `s`
`thread step-over`	`next` `n`
`thread step-out`	`finish` `f`

By default, LLDB defines aliases for all the common GDB control commands (`s`, `step`, `n`, `next`, `finish`). If any are missing, you can add them to your `~/.lldbinit` file using the `command alias` command.

LLDB also supports the *step by instruction* command variants:

LLDB command	GDB command
`thread step-inst`	`stepi` `si`
`thread step-over-inst`	`nexti` `ni`

Finally, LLDB includes a *run until line or frame exit* stepping mode:

```
(lldb) thread until 100
```

This command either runs the thread in the current frame until it reaches line 100 in this frame, or it stops if it leaves the current frame. This behavior is almost equivalent to the GDB `until` command.

By default, a process will share the LLDB terminal with the inferior process. In this mode (like when debugging with GDB), when the process is running, anything you type goes to the standard input of the inferior process. To interrupt the inferior program, press **Ctrl+C**.

If you launch a process with the `--no-stdin` option, the command interpreter is always available to enter commands. This allows you to set a breakpoint, for example. without explicitly interrupting the program you are debugging:

```
(lldb) process continue
(lldb) breakpoint set --name stop_here
```

NOTE: For GDB users, it might seem odd to always have the `(lldb)` prompt.

Many LLDB commands will not work while a program is running; the command interpreter should notify you when this is the case. (If you find any instances where the command interpreter is not doing its job, please file a bug report.)

The commands that work while a program is running include interrupting the process to halt execution (`process interrupt`), getting the process status (`process status`), breakpoint setting and clearing (`breakpoint [set|clear|enable|disable|list] ...`), and memory reading and writing (`memory [read|write] ...`).

The issue of disabling `stdio` while running is a good opportunity to show how to set debugger properties in general. If you always want to run in the `--no-stdin` mode, you can set it as a generic process property using the LLDB `settings` command, which is equivalent to the GDB `set` command. For example, in this case you would specify the following command:

```
(lldb) settings set target.process.disable-stdio true
```

Over time, the GDB set command was expanded into many disordered options, such that there were useful options that even experienced GDB users did not know about because these options were too hard to find.

By contrast, LLDB organizes the settings hierarchically, using the structure of the basic entities in the debugger. Anywhere you can specify a setting on a generic entity (such as threads), you can also apply the option to a specific instance, which can be convenient in some cases.

You can view the available settings with `settings list`, and the `settings` command provides help explaining how the command works.

2.8 Examine the thread state

When you stop a program, LLDB chooses a current thread (usually the one that stopped for a reason), and a current frame in that thread (on stop, this is always the bottom-most frame). Many of the commands that are used to inspect state choose their current thread/frame in this way.

To inspect the current state of your process, start with listing the threads:

```
(lldb) thread list
Process 46915 state is Stopped
* thread #1: tid = 0x2c03, 0x00007fff85cac76a, where =
libSystem.B.dylib`__getdirentries64 +10, stop reason = signal =
SIGSTOP, queue = com.apple.main-thread
  thread #2: tid = 0x2e03, 0x00007fff85cbb08a, where =
libSystem.B.dylib`kevent + 10, queue = com.apple.libdispatch-manager
  thread #3: tid = 0x2f03, 0x00007fff85cbbeaa, where =
libSystem.B.dylib`__workq_kernreturn + 10
```

The `*` in this listing indicates that thread 1 is the current thread.

To list the backtrace for that thread, use the thread backtrace command:

```
(lldb) thread backtrace
thread #1: tid = 0x2c03, stop reason = breakpoint 1.1, queue = com.apple.
main-thread
frame #0: 0x0000000100010d5b, where = Sketch`-[SKTGraphicView
alignLeftEdges:] + 33 at /Projects/Sketch/SKTGraphicView.m:1405
frame #1: 0x00007fff8602d152, where = AppKit`-[NSApplication sendAction:
to:from:] + 95
frame #2: 0x00007fff860516be, where = AppKit`-[NSMenuItem
_corePerformAction] + 365
frame #3: 0x00007fff86051428, where = AppKit`-[NSCarbonMenuImpl
performActionWithHighlightingForItemAtIndex:] + 121
frame #4: 0x00007fff860370c1, where = AppKit`-[NSMenu
performKeyEquivalent:] + 272
frame #5: 0x00007fff86035e69, where = AppKit`-[NSApplication
_handleKeyEquivalent:] + 559
frame #6: 0x00007fff85f06aa1, where = AppKit`-[NSApplication sendEvent:]
+ 3630
frame #7: 0x00007fff85e9d922, where = AppKit`-[NSApplication run] + 474
frame #8: 0x00007fff85e965f8, where = AppKit`NSApplicationMain + 364
frame #9: 0x0000000100015ae3, where = Sketch`main + 33 at /Projects/
Sketch/SKTMain.m:11
frame #10: 0x0000000100000f20, where = Sketch`start + 52
```

You can also provide a list of threads to backtrace, or the `all` keyword to see all threads:

```
(lldb) thread backtrace all
```

You can select the current thread (which will be used by default in all the commands in Section 2.9) with the `thread select` command:

```
(lldb) thread select 2
```

Where the thread index is only the one shown in the `thread list` listing.

2.9 Examine the stack frame state

Use the `frame variable` command to display a stack frame's arguments and local variables. For example:

```
(lldb) frame variable
self = (SKTGraphicView *) 0x0000000100208b40
_cmd = (struct objc_selector *) 0x000000010001bae1
sender = (id) 0x00000001001264e0
selection = (NSArray *) 0x00000001001264e0
i = (NSUInteger) 0x00000001001264e0
c = (NSUInteger) 0x00000001001253b0
```

As shown in this example, if you do not specify any variable names, all arguments and locals are displayed. If you invoke the `frame variable` command with the names of specific locals, only those variables are printed. For example:

```
(lldb) frame variable self
(SKTGraphicView *) self = 0x0000000100208b40
```

You can also specify the path to a sub-element of one of the available locals, and that sub-element is printed:

```
(lldb) frame variable self.isa
(struct objc_class *) self.isa = 0x0000000100023730
```

The `frame variable` command is not a full expression parser, but it does support a few basic operators such as `&`, `*`, `->`, `[]` (no overloaded operators). Use the array brackets on pointers to treat pointers as arrays:

```
(lldb) frame variable *self
(SKTGraphicView *) self = 0x0000000100208b40
(NSView) NSView = {
(NSResponder) NSResponder = {
...

(lldb) frame variable &self
(SKTGraphicView **) &self = 0x0000000100304ab

(lldb) frame variable argv[0]
(char const *) argv[0] = 0x00007fff5fbffaf8
"/Projects/Sketch/build/Debug/Sketch.app/Contents/MacOS/Sketch"
```

The `frame variable` command also performs *object printing* operations on variables. Enable this feature by using the `-O` option with the `frame variable` command:

```
(lldb) frame variable -O self (SKTGraphicView *)
         self = 0x0000000100208b40
<SKTGraphicView: 0x100208b40>
```

Use the `frame` **`select command to select another stack frame to display:`**

```
(lldb) frame select 9
frame #9: 0x0000000100015ae3, where = Sketch`function1 + 33 at
   /Projects/Sketch/SKTFunctions.m:11
```

You can also move up or down the stack by using the `--relative` (`-r`) option with the `frame select` command. Also, the predefined command aliases `u` and `d` work like their GDB equivalents.

To view more complex data or change program data, use the `expression` command. It takes an expression and evaluates it in the scope of the currently selected frame. For example:

```
(lldb) expr self
$0 = (SKTGraphicView *) 0x0000000100135430
(lldb) expr self = 0x00
$1 = (SKTGraphicView *) 0x0000000000000000
(lldb) frame var self
(SKTGraphicView *) self = 0x0000000000000000
```

You can also call functions:

```
(lldb) expr (int) printf ("I have a pointer 0x%llx.\n", self)
$2 = (int) 22
I have a pointer 0x0.
```

The `expression` command is one of the raw commands, which means you are not required to quote your entire expression, backslash protect the quote marks, and so on.

Finally, the results of the expressions are stored in persistent variables (of the form `$[0-9]+`) that you can use in more expressions:

```
(lldb) expr self = $0
$4 = (SKTGraphicView *) 0x0000000100135430
```

3 Use the debugger

The LLDB debugger debugs C and C++ programs written for the Hexagon processor.

3.1 Start the debugger

To start LLDB from a command line, enter:

`hexagon-lldb program` *`[option...]`*

Where:

- `program` is the name of an executable Hexagon file
- `option` indicates the LLDB options (see Section 3.2)

NOTE: The program file can alternatively be specified on the command line using the LLDB `--file` option.

3.1.1 Command line arguments

Command line arguments in LLDB are complicated by the fact that arguments might be required to be passed to three different entities:

- The LLDB itself
- The Hexagon simulator (which is launched by LLDB)
- The program that is being debugged

To distinguish these sets of command line items, all the LLDB arguments and options are specified first on the command line, followed by all the simulator arguments and options, followed by all the program arguments and options. Each set of items is separated on the command line by an instance of the `--` character sequence. For example:

`hexagon-lldb program` *`[option...]`* **`-- sim_args -- program_args`**

NOTE: The program name specified for LLDB is automatically passed to the simulator, so you can specify it only once on the command line.

In the following example, `--` is used to separate the LLDB `foo.elf` argument from the simulator `-mv5` command option from the program `bar` argument.

```
hexagon-lldb foo.elf -- -mv5 -- bar
```

`foo.elf` is automatically passed to the simulator as the name of the file to run.

3.1.2 Command files

When LLDB starts up, it reads debug settings, commands, and command aliases from the following three command files, in the specified order:

1. `.lldbinit-debugger`

   When using a command line interface, this file is named `.lldbinit-hexagon-lldb`. When using LLDB in Eclipse, it is named `.lldbinit-hexagon-lldb-mi`.

   This file is a useful place for storing settings that you want to apply only when a given command interpreter is used.

2. `~/.lldbinit`

3. `.lldbinit`

   This file is assumed to be stored in the current working directory (that is, where LLDB was started).

3.1.3 Debug a remote application

LLDB can debug an application already running on the Hexagon simulator, if the simulator was launched with a unique port number.

1. Launch the simulator that is running the application to be debugged, using the `-G` switch to identify a unique port number:

   `hexagon-sim -G Port# program sim_args`

   The simulator loads the program with the `sim_args`, but it pauses the program at the beginning.

2. Start LLDB from the command line in a separate terminal or machine:

   `hexagon-lldb program` `[option...]`

   Where program is the same file loaded by the simulator.

3. At the (lldb) prompt, enter the command:

   `(lldb)` **`gdb-remote host:Port#`**

   Where:

   - `host:` is the name of the machine on which hexagon-sim is running, if the machine is different from where LLDB is running
   - P`ort#` is the port number used when launching the simulator

4. When the debugger establishes a connection to the simulator, proceed with typical debug operations.

NOTE: All input and output performed by the application occurs where the simulator is running.

3.2 Debug options

LLDB supports the following debug options to control various LLDB features from the command line when LLDB is first launched.

```
-e          | --editor
-f filename | --file filename
-h          | --help
-o cmdname  | --one-line cmdname
-O cmdname  | --one-line-before-file cmdname
-Q          | --source-quietly
-s filename | --source filename
-S filename | --source-before-file filename
-v          | --version
-x          | --no-lldbinit
```

Descriptions

-e
--editor

Open all program source files using the host's external editor mechanism.

-f *filename*
-file *filename*

Specify the executable file to launch.

If this option is not specified, LLDB uses the first argument on the command line as the name of the executable file to launch.

-h
--help

Print the usage information for the debugger.

-o *cmdname*
--one-line *cmdname*

Execute the specified LLDB command immediately *after* LLDB loads the executable file specified on the command line.

The specified command must fit on one line.

You can use this option multiple times. The specified commands are executed in the order (from left to right) that they appear on the command line.

-O *cmdname*
--one-line-before-file *cmdname*

Execute the specified LLDB command immediately *before* LLDB loads the executable file specified on the command line.

The specified command must fit on one line.

You can use this option multiple times. The specified commands are executed in the order (from left to right) that they appear on the command line.

-Q
--source-quietly

Suppress the display output generated by any LLDB commands that are specified with the -o, -O, -s, or -S commands.

-s *filename*
-source *filename*

Execute the specified LLDB commands immediately *after* LLDB loads the executable file specified on the command line.

The specified file is a text file containing a list of the command names to execute.

You can use this option multiple times. The specified commands are executed in the order (from left to right) that they appear on the command line.

-S *filename*
-source-before-file *filename*

Execute the specified LLDB commands immediately *before* LLDB loads the executable file specified on the command line.

The specified file is a text file containing a list of the command names to execute.

You can use this option multiple times. The specified commands are executed in the order (from left to right) that they appear on the command line.

-v
--version

Print the version number of the debugger.

-x
--no-lldbinit

Do not automatically parse any `.lldbinit` files.

3.3 Debug commands

This section lists the LLDB commands by category, along with their GDB equivalents (which in many cases are identical).

Command names can be shortened in LLDB if they uniquely identify the command. For example, to invoke the `breakpoint set` command, enter `br se`.

NOTE: The `tlb`, `pagetable`, `pv8`, `pv16`, and `pv32` commands are specific to the Hexagon version of LLDB. They are implemented in the `hexagon_utils.py` script.

3.3.1 Command options

Many LLDB debug commands accept options that resemble the command line options used to start the debugger itself. These options are command-specific: they work only with the debug commands for which they are defined.

You can specify debug command options anywhere on the command line, but if any command arguments begin with a `-` character, the LLDB will be unable to distinguish the command options from the command arguments. In this case, all the options must be specified before the arguments, and the two sets of items must be separated with the `--` character sequence.

In this example, `--` is used to separate the `--stop-at-entry` command option from the `-program-arg` command argument:

```
(lldb) process launch --stop-at-entry -- -program_arg value
```

3.3.2 Execution commands

Figure 3-1 LLDB execution commands

GDB	LLDB
Launch `process` with no arguments	

`(gdb) run`
`(gdb) r`
	`(lldb) process launch`
	`(lldb) run`
	`(lldb) r`

Launch `process` with arguments	
`(gdb) run args` `(gdb) r args`	`(lldb) process launch -- args` `(lldb) r args`
Launch `process` with arguments `a.out 1 2 3`, without supplying arguments every time	

`% gdb --args a.out 1 2 3`
`(gdb) run`
`...`
`(gdb) run`
`...`
	`% lldb -- a.out 1 2 3`
	`(lldb) run`
	`...`
	`(lldb) run`
	`...`

Figure 3-1 LLDB execution commands (cont.)

GDB	LLDB
Or	
(gdb) set args 1 2 3 (gdb) run ... (gdb) run ...	(lldb) settings set target.run-args 1 2 3 (lldb) run ... (lldb) run ...
Set the nvironment variables for `process` **before launching**	
(gdb) set env DEBUG 1	(lldb) settings set target.env-vars DEBUG=1 (lldb) set se target.env-vars DEBUG=1 (lldb) env DEBUG=1
Unset the environment variables for `process` **before launching**	
(gdb) unset env DEBUG	(lldb) settings remove target.env-vars DEBUG (lldb) set rem target.env-vars DEBUG
Show the arguments that will be or were passed to the program when run	
(gdb) show args Argument list to give a program being debugged when it is started is `1 2 3`.	(lldb) settings show target.run-args target.run-args (array of strings) = [0]: "1" [1]: "2" [2]: "3"
Set the environment variables for `process`**, and launch** `process` **in one command**	
	(lldb) process launch -v DEBUG=1
Attach to the remote GDB protocol server running on system `eorgadd`**, port 8000**	
(gdb) target remote eorgadd:8000	(lldb) gdb-remote eorgadd:8000
Attach to the remote GDB protocol server running on local system, port 8000	
(gdb) target remote localhost:8000	(lldb) gdb-remote 8000
Attach to the Darwin kernel in `kdp` **mode on system** `eorgadd`	
(gdb) kdp-reattach eorgadd	(lldb) kdp-remote eorgadd
Launch the simulator with the current target, and connect to it	
	(lldb) run
Do a source level single step in the currently selected thread	
(gdb) step (gdb) s	(lldb) thread step-in (lldb) step (lldb) s
Do a source level single step over in the currently selected thread	
(gdb) next (gdb) n	(lldb) thread step-over (lldb) next (lldb) n
Do an instruction-level single step in the currently selected thread	
(gdb) stepi (gdb) si	(lldb) thread step-inst (lldb) si
Do an instruction-level single step over in the currently selected thread	
(gdb) nexti (gdb) ni	(lldb) thread step-inst-over (lldb) ni

Figure 3-1 LLDB execution commands (cont.)

GDB	LLDB
Step out of the currently selected frame	
`(gdb) finish`	`(lldb) thread step-out` `(lldb) finish`
Return immediately from the currently selected frame, with an optional return value	
`(gdb) return` *expression*	`(lldb) thread return` *expression*
Backtrace and disassemble every time you stop	

	`(lldb) target stop-hook add`
	
	Enter your stop hook commands. Enter `DONE` to end.
	`> bt`
	`> disassemble --pc`
	`> DONE`
	`Stop hook #1 added.`

3.3.3 Breakpoint commands

Table 3-1 LLDB breakpoint commands

GDB	LLDB
Set a breakpoint at all functions named `main`	

`(gdb) break main`	`(lldb) breakpoint set --name main`
	`(lldb) br s -n main`
	`(lldb) b main`

Set a breakpoint in the `test.c` **file at line 12**	

`(gdb) break test.c:12`	`(lldb) breakpoint set --file test.c --line 12`
	`(lldb) br s -f test.c -l 12`
	`(lldb) b test.c:12`

Set a breakpoint at all C++ methods whose base name is `main`	
`(gdb) break main` (Hope that no C functions are named `main`)	`(lldb) breakpoint set --method main` `(lldb) br s -M main`
Set a breakpoint by a regular expression on the function name	

`(gdb) rbreak regular-expression`	`(lldb) breakpoint set --func-regex regular-expression`
	
	`(lldb) br s -r regular-expression`

Ensure that breakpoints by file and line work for `#included .c/.cpp/.m` **files**	

`(gdb) b foo.c:12`	`(lldb) settings set target.inline-breakpoint-strategy always`
	
	`(lldb) br s -f foo.c -l 12`

Set a breakpoint by a regular expression on the source file contents	
`(gdb) shell grep -e -n pattern source-file` `(gdb) break source-file: CopyLineNumbers`	`(lldb) breakpoint set --source-pattern regular-expression --file SourceFile` `(lldb) br s -p regular-expression -f file`

Table 3-1 LLDB breakpoint commands (cont.)

GDB	LLDB
Set a conditional breakpoint	
`(gdb) break foo if strcmp(y,"hello") == 0`	`(lldb) breakpoint set --name foo --condition '(int)strcmp(y,"hello") == 0'` `(lldb) br s -n foo -c '(int)strcmp(y,"hello") == 0'`
List all breakpoints	
`(gdb) info break`	`(lldb) breakpoint list` `(lldb) br l`
Delete a breakpoint	
`(gdb) delete 1`	`(lldb) breakpoint delete 1` `(lldb) br del 1`

3.3.4 Watchpoint commands

Table 3-2 LLDB watchpoint commands

GDB	LLDB
Set a watchpoint on a variable when it is written to	
`(gdb) watch global_var`	`(lldb) watchpoint set variable global_var` `(lldb) wa s v global_var`
Set a watchpoint on a memory location when it is written to	

`(gdb) watch -location g_char_ptr`	`(lldb) watchpoint set expression -- my_ptr`
	`(lldb) wa s e -- my_ptr`
	
	The size of the region to watch for defaults to the pointer size if no `-x byte_size` is specified.
	This command takes raw input, evaluated as an expression returning an unsigned integer pointing to the start of the region, after the `--` option terminator.

Set a condition on a watchpoint	
	`(lldb) watch set var global` `(lldb) watchpoint modify -c '(global==5)'` `(lldb) c` `...` `(lldb) bt` `* thread #1: tid = 0x1c03, 0x0000000100000ef5 a.out`modify + 21 at main.cpp:16, stop reason = watchpoint 1` `frame #0: 0x0000000100000ef5 a.out`modify + 21 at main.cpp:16` `frame #1: 0x0000000100000eac a.out`main + 108 at main.cpp:25` `frame #2: 0x00007fff8ac9c7e1 libdyld.dylib`start + 1` `(lldb) frame var global` `(int32_t) global = 5`

Table 3-2 LLDB watchpoint commands (cont.)

GDB	LLDB
List all watchpoints	
`(gdb) info break`	`(lldb) watchpoint list` `(lldb) watch l`
Delete a watchpoint	
`(gdb) delete 1`	`(lldb) watchpoint delete 1` `(lldb) watch del 1`

3.3.5 Examine the variables

Table 3-3 LLDB commands for examining variables

GDB	LLDB
Show arguments and local variables for the current frame	

`(gdb) info args`
 	
And:
`(gdb) info locals`
	`(lldb) frame variable`
	`(lldb) fr v`

Show the local variables for the current frame	
`(gdb) info locals`	`(lldb) frame variable --no-args` `(lldb) fr v -a`
Show the contents of the `bar` local variable	

`(gdb) p bar`	`(lldb) frame variable bar`
	`(lldb) fr v bar`
	`(lldb) p bar`

Show contents of the `bar` local variable formatted as hex	
`(gdb) p/x bar`	`(lldb) frame variable --format x bar` `(lldb) fr v -f x bar`
Show the contents of the `baz` global variable	
`(gdb) p baz`	`(lldb) target variable baz` `(lldb) ta v baz`
Show the global/static variables defined in the current source file	
`N/A`	`(lldb) target variable` `(lldb) ta v`
Display the `argc` and `argv` variables every time you stop	

`(gdb) display argc`
`(gdb) display argv`
	`(lldb) target stop-hook add --one-liner "frame variable argc argv"`
	`(lldb) ta st a -o "fr v argc argv"`
	`(lldb) display argc`
	`(lldb) display argv`

Display the `argc` and `argv` variables only when you stop in the `main` function	
	`(lldb) target stop-hook add --name main --one- liner "frame variable argc argv"` `(lldb) ta st a -n main -o "fr v argc argv"`

Table 3-3 LLDB commands for examining variables (cont.)

GDB	LLDB
Display the `*this` variable only when you stop in the `MyClass` C class	
	`(lldb) target stop-hook add --classname MyClass --one-liner "frame variable *this"` `(lldb) ta st a -c MyClass -o "fr v *this"`

3.3.6 Evaluate the expressions

Table 3-4 LLDB commands for evaluating expressions

GDB	LLDB
Evaluate a generalized expression in the current frame	

`(gdb) print (int) printf ("Print nine: %d.", 4 + 5)`
 	
Or, if you do not want to see void returns:
`(gdb) call (int) printf ("Print nine: %d.", 4 + 5)`
	`(lldb) expr (int) printf ("Print nine: %d.", 4 + 5)`
	
	Or, using the `print` alias:
	`(lldb) print (int) printf ("Print nine: %d.", 4 + 5)`

Create and assign a value to the convenience variable	

`(gdb) set $foo = 5`
`(gdb) set variable $foo = 5`
 	
Or, using the `print` command:
`(gdb) print $foo = 5`
 	
Or, using the `call` command
`(gdb) call $foo = 5`
 	
To specify the type of the variable:
`(gdb) set $foo = (unsigned int) 5`
	In LLDB, you evaluate a variable declaration expression as you would write it in C:
	`(lldb) expr unsigned int $foo = 5`

Print the dynamic type of the result of an expression	

`(gdb) set print object 1`
`(gdb) p someCPPObjectPtrOrReference`
 	
Only works for C++ objects
	`(lldb) expr -d 1 -- [SomeClass returnAnObject]`
	`(lldb) expr -d 1 -- someCPPObjectPtrOrReference`
	
	Or, set dynamic type printing to be the default:
	`(lldb) settings set target.prefer-dynamic run-target`

Call a function so you can stop at a breakpoint in the function	
`(gdb) set unwindonsignal 0` `(gdb) p function_with_a_breakpoint()`	`(lldb) expr -i 0 -- function_with_a_breakpoint()`
Call a function that crashes, and stop when the function crashes	
`(gdb) set unwindonsignal 0` `(gdb) p function_which_crashes()`	`(lldb) expr -u 0 -- function_which_crashes()`

3.3.7 Examine the thread state

Table 3-5 LLDB commands for examining the thread state

GDB	LLDB
Show the stack backtrace for the current thread	
`(gdb) bt`	`(lldb) thread backtrace` `(lldb) bt`
Show the stack backtraces for all threads	
`(gdb) thread apply all bt`	`(lldb) thread backtrace all` `(lldb) bt all`
Backtrace the first five frames of the current thread	

`(gdb) bt 5`	`(lldb) thread backtrace -c 5`
	`(lldb) bt 5 (lldb-169 and later)`
	`(lldb) bt -c 5 (lldb-168 and earlier)`

Select a different stack frame by index for the current thread	

`(gdb) frame 12`	`(lldb) frame select 12`
	`(lldb) fr s 12`
	`(lldb) f 12`

List information about the currently selected frame in the current thread	
	`(lldb) frame info`
Select the stack frame that called the current stack frame	
`(gdb) up`	`(lldb) up` `(lldb) frame select --relative=1`
Select the stack frame that is called by the current stack frame	

`(gdb) down`	`(lldb) down`
	`(lldb) frame select --relative=-1`
	`(lldb) fr s -r-1`

Select a different stack frame using relative offset	

`(gdb) up 2`
`(gdb) down 3`
	`(lldb) frame select --relative 2`
	`(lldb) fr s -r2`
	
	`(lldb) frame select --relative -3`
	`(lldb) fr s -r-3`

Show the general purpose registers for the current thread	
`(gdb) info registers`	`(lldb) register read`
Write a new decimal value `123` to the current `rax` thread register	
`(gdb) p $rax = 123`	`(lldb) register write rax 123`
Skip 8 bytes ahead of the current program counter (instruction pointer)	
`(gdb) jump *$pc+8`	`(lldb) register write pc ` `$pc+8` `` **NOTE:** Use backticks to evaluate an expression and insert the scalar result in the LLDB.

Table 3-5 LLDB commands for examining the thread state (cont.)

GDB	LLDB
Show the general purpose registers for the current thread, formatted as signed decimal	

	LLDB tries to use the same format characters as `printf(3)` when possible. Enter `help format` to see a full list of format specifiers:
	`(lldb) register read --format i`
	`(lldb) re r -f i`
	
	LLDB now supports the GDB shorthand format syntax, but no space can appear after the command:
	`(lldb) register read/d`

Show all registers in all register sets for the current thread	
`(gdb) info all-registers`	`(lldb) register read --all` `(lldb) re r -a`
Show values for the `rax`, `rsp`, and `rbp` registers in the current thread	
`(gdb) info all-registers rax rsp rbp`	`(lldb) register read rax rsp rbp`
Show values for the `rax` register in the current thread, formatted as binary	

`(gdb) p/t $rax`	`(lldb) register read --format binary rax`
	`(lldb) re r -f b rax`
	
	LLDB now supports the GDB shorthand format syntax but no space can appear after the command:
	`(lldb) register read/t rax`
	`(lldb) p/t $rax`

Show values for the `rax` vector register in the current thread, formatted as data array	

	`(lldb) pv8 rax (8-bit array)`
	`(lldb) pv16 rax (16-bit array)`
	`(lldb) pv32 rax (32-bit array)`

Show values for `tlb` in the current thread	
`(gdb) info tlb`	`(lldb) get tlb info`
Show values for `pagetable` in the current thread	
`(gdb) info pagetable`	`(lldb) get pagetable info`
Read memory from address `0xbffff3c0` and show 4 hex `uint32_t` values	

`(gdb) x/4xw 0xbffff3c0`	`(lldb) memory read --size 4 --format x --count 4 0xbffff3c0`
	`(lldb) me r -s4 -fx -c4 0xbffff3c0`
	`(lldb) x -s4 -fx -c4 0xbffff3c0`
	
	LLDB now supports the GDB shorthand format syntax but no space can appear after the command:
	`(lldb) memory read/4xw 0xbffff3c0`
	`(lldb) x/4xw 0xbffff3c0`
	`(lldb) memory read --gdb-format 4xw 0xbffff3c0`

Table 3-5 LLDB commands for examining the thread state (cont.)

GDB	LLDB
Read memory starting at expression `argv[0]`	
`(gdb) x argv[0]`	`(lldb) memory read `argv[0]``  Any command can inline a scalar expression result (if the target is stopped) by using backticks around any expression: `(lldb) memory read --size `sizeof(int)` `argv[0]``
Read 512 bytes of memory from address `0xbffff3c0` **and save the results to a local file as text**	

`(gdb) set logging on`
`(gdb) set logging file /tmp/mem.txt`
`(gdb) x/512bx 0xbffff3c0`
`(gdb) set logging off`
	`(lldb) memory read --outfile /tmp/mem.txt --count 512 0xbffff3c0`
	`(lldb) me r -o/tmp/mem.txt -c512 0xbffff3c0`
	`(lldb) x/512bx -o/tmp/mem.txt 0xbffff3c0`

Save binary memory data starting at `0x1000` **and ending at** `0x2000` **to a file**	
`(gdb) dump memory /tmp/mem.bin 0x1000 0x2000`	`(lldb) memory read --outfile /tmp/mem.bin --binary 0x1000 0x2000` `(lldb) me r -o /tmp/mem.bin -b 0x1000 0x2000`
Disassemble the current function for the current frame	
`(gdb) disassemble`	`(lldb) disassemble --frame` `(lldb) di -f`
Disassemble any functions named `main`	
`(gdb) disassemble main`	`(lldb) disassemble --name main` `(lldb) di -n main`
Disassemble an address range	
`(gdb) disassemble 0x1eb8 0x1ec3`	`(lldb) disassemble --start-address 0x1eb8 --end-address 0x1ec3` `(lldb) di -s 0x1eb8 -e 0x1ec3`
Disassemble 20 instructions from a given address	
`(gdb) x/20i 0x1eb8`	`(lldb) disassemble --start-address 0x1eb8 --count 20` `(lldb) di -s 0x1eb8 -c 20`
Show mixed source and disassembly for the current function for the current frame	
N/A	`(lldb) disassemble --frame --mixed` `(lldb) di -f -m`
Disassemble the current function for the current frame and show the opcode bytes	
N/A	`(lldb) disassemble --frame --bytes` `(lldb) di -f -b`
Disassemble the current source line for the current frame	
N/A	`(lldb) disassemble --line` `(lldb) di -l`

3.3.8 Executable and shared library query commands

Figure 3-2 LLDB commands for querying executable and shared libraries

GDB	LLDB
List the main executable and all dependent shared libraries	
`(gdb) info shared`	`(lldb) image list`
Look up information for the raw address in the executable or any shared libraries	
`(gdb) info symbol 0x1ec4`	`(lldb) image lookup --address 0x1ec4` `(lldb) im loo -a 0x1ec4`
Look up functions matching the regular expression in a binary	

`(gdb) info function <FUNC_REGEX>`	This command finds debug symbols:
	`(lldb) image lookup -r -n <FUNC_REGEX>`
	
	This command finds non-debug symbols:
	`(lldb) image lookup -r -s <FUNC_REGEX>`
	
	Provide a list of binaries as arguments to limit the search.

Find full source line information	
`(gdb) info line 0x1ec4`	This command is a bit messy, currently. Enter: `(lldb) image lookup -v --address 0x1ec4` Look for the `LineEntry` line, which will have the full source path and line range information.
Look up information for an address in `a.out` only	
	`(lldb) image lookup --address 0x1ec4 a.out` `(lldb) im loo -a 0x1ec4 a.out`
Look up information for a `type Point` by name	
`(gdb) ptype Point`	`(lldb) image lookup --type Point` `(lldb) im loo -t Point`
Dump all sections from the main executable and any shared libraries	
`(gdb) maintenance info sections`	`(lldb) image dump sections`
Dump all sections in module `a.out`	
	`(lldb) image dump sections a.out`
Dump all symbols from the main executable and any shared libraries	
	`(lldb) image dump symtab`
Dump all symbols in `a.out` and `liba.so`	
	`(lldb) image dump symtab a.out liba.so`

3.3.9 Miscellaneous commands

Figure 3-3 Miscellaneous LLDB commands

GDB	LLDB
Echo text to the screen	
`(gdb) echo Here is some text\n`	`(lldb) script print "Here is some text"`
Remap the source file pathnames for the debug session	
`(gdb) set pathname-substitutions /buildbot/path /my/path`	`(lldb) settings set target.source-map /buildbot/path /my/path` If your source files are no longer in the same location as when the program was built (typically because the program was built on a different computer), LLDB must know how to find the sources at their local file path instead of the build system's file path.
Supply a catchall directory to search for source files in	
`(gdb) directory /my/path`	No equivalent command; use the `source-map` instead.

4 Frame and thread formatting

LLDB lets you customize how frame and thread information is displayed in the debugger.

4.1 Stack frame and thread format

LLDB lets you define the format used for displaying information on threads and stack frames.

Typically, when your program stops at a breakpoint, LLDB displays a line explaining why your thread stopped:

```
* thread #1: tid = 0x2e03, 0x0000000100000e85 a.out`main + 4, stop reason
    = breakpoint 1.1
```

Stack `backtrace` frames display a similar line:

```
(lldb)* thread backtrace *thread #1: tid = 0x2e03, stop reason =
breakpoint 1.1
  frame #0: 0x0000000100000e85 a.out`main + 4 at test.c:19
  frame #1: 0x0000000100000e40 a.out`start + 52
```

Use the `settings set` command to change the display formats used in these two lines:

```
(lldb) settings set frame-format STRING *
(lldb) settings set thread-format STRING
```

4.2 Format strings

Format strings can contain the following elements:

- Plain text

  Includes any text that does not contain a `{`, `}`, `$`, or `\` character.

- Control characters
- Variables (which access the current program state)

  Specified in a format string with the `${` prefix and `}` suffix. For example: `${frame.pc}`.

4.2.1 Variables

Table 4-1 Format string variables

Variable Name	Description
`file.basename`	Base name of the current compile unit file for the current frame.
`file.fullpath`	Full path of the current compile unit file for the current frame.
`frame.index`	Frame index (0, 1, 2, 3...).
`frame.pc`	Generic frame register for the program counter.
`frame.sp`	Generic frame register for the stack pointer.
`frame.fp`	Generic frame register for the frame pointer.
`frame.flags`	Generic frame register for the flags register.
`frame.reg.NAME`	Access to any platform-specific register by name (replace `NAME` with the name of the intended register).
`function.name`	Name of the current function or symbol.
`function.name-with-args`	Name of the current function with arguments and values, or the name of the symbol.
`function.pc-offset`	Program counter offset within the current function or symbol.
`line.file.basename`	Base name of the line table entry to the file for the current line entry in the current frame.
`line.file.fullpath`	Full path of the line table entry to the file for the current line entry in the current frame.
`line.number`	Line number of the line table entry for the current line entry in the current frame.
`line.start-addr`	Starting address of the line table entry for the current line entry in the current frame.
`line.end-addr`	Ending address of the line table entry for the current line entry in the current frame.
`module.file.basename`	Base name of the current module (shared library or executable).
`module.file.fullpath`	Base name of the current module (shared library or executable).
`process.file.basename`	Base name of the file for the process.
`process.file.fullpath`	Full name of the file for the process.
`process.id`	Process ID native to the system on which the inferior runs.
`process.name`	Name of the process at runtime.
`thread.id`	Thread identifier for the current thread.
`thread.index`	Unique ID of the one-based thread index that is guaranteed to be unique as threads come and go.
`thread.name`	Name of the thread if the target operating system supports naming threads.
`thread.queue`	Queue name of the thread if the target operating system supports dispatch queues.

Table 4-1 Format string variables (cont.)

Variable Name	Description
`thread.stop-reason`	Text explanation of why each thread stopped.
`thread.return-value`	Return value of the latest step operation (currently only for step-out).
`thread.completed-expression`	Expression result for a thread that just finished an interrupted expression evaluation.
`target.arch`	Architecture of the current target.
`target.script:/python_func/`	Use a Python function to generate a piece of text output.
`process.script:/python_func/`	
`thread.script:/python_func/`	
`frame.script:/python_func/`	

4.2.2 Control characters

Control characters include `{`, `}`, and `\`.

The `{` and `}` characters are used for scoping blocks, while `\` is used to desensitize control characters and emit non-printable characters.

4.2.3 Desensitizing characters

The backslash control character, `\`, lets you enter the typical `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`, and `\\` characters. It also allows you to specify characters in octal representation (`\0123`) and hexadecimal representation (`\xAB`).

The `\` character enables you to enter escape characters into your format strings, and it allows colorized output for terminals that support color.

4.2.4 Scoping

In many cases, the information that you might have configured to appear in your prompt might not be available, and you do not want the prompt to print out the information if it is not valid. To handle this situation, you can enclose everything that must be resolved into a *scope*.

A scope is defined to begin with `{` and end with `}`. For example, the following scope displays the base name and line number for the current frame line table entry, but only when this information is available for the current frame:

```
{ at {$line.file.basename}:${line.number}}
```

The scope in this example consists of the following parts:

- The start of the scope:

  ```
  {
  ```

- The format whose content is displayed only if all information is available:

```
at {$line.file.basename}:${line.number}
```

- The end of the scope:

```
}
```

4.3 Format example

The following information line is an example of what can be displayed when your program is stopped in a frame:

```
frame #0: 0x0000000100000e85 a.out`main + 4 at test.c:19
```

You can specify this information line with the following format specifier:

```
settings set frame-format #${frame.index}:${frame.pc}
{${module.file.basename} `{${function.name}{${function.pc-offset}} {at
${line.file.basename}:${line.number}}\n
```

The format specifier in this example consists of the following parts:

- Always print the frame index and frame PC:

```
#${frame.index}: ${frame.pc}
```

- Only print the module followed by a tick if a valid module exists for current frame:

```
{${module.file.basename}`}
```

- Print the function name with an optional offset:

```
{${function.name}{${function.pc-offset}}}
```

- Print the line information, if available:

```
{ at ${line.file.basename}:${line.number}}
```

- End the information line with a newline character:

```
\n
```

4.4 User-defined formats

When modifying your own format strings, it is useful to start with the default values for the frame and thread format strings. To access the default settings, use the `settings show` command. For example:

```
(lldb) settings show thread-format
thread-format (string) = 'thread #${thread.index}:
tid = ${thread.id}{, ${frame.pc}}{
${module.file.basename}`${function.name}{${function.pc-offset}}}{, stop
reason = ${thread.stop-reason}}{, name = ${thread.name}}{, queue =
${thread.queue}}\n'
(lldb) settings show frame-format frame-format (string) = 'frame
#${frame.index}: ${frame.pc}{ ${module.file.basename} `${function.name}
{${function.pc-offset}}}{at ${line.file.basename}:${line.number}}\n'
```

When defining a thread format, you must use scopes (Section 4.2.4) to contain any information that comes from a stack frame, because the thread format does not always show frame information.

When displaying the backtrace for a thread, you are not required to duplicate the information for frame zero in the thread information. For example:

```
(lldb)* thread backtrace
thread #1: tid = 0x2e03, stop reason = breakpoint 1.1 2.1
  frame #0: 0x0000000100000e85 a.out`main + 4 at test.c:19
  frame #1: 0x0000000100000e40 a.out`start + 52 |
```

The frame-related variables are:

```
${file.*}
${frame.*}
${function.*}
${line.*}
${module.*}
```

In the default thread format definition, all frame-related information is in scopes, so you can omit it when the thread information is displayed in contexts where frame information is not applicable. For example:

```
|'thread #${thread.index}: tid = ${thread.id}{, ${frame.pc}}{
${module.file.basename}`${function.name}{${function.pc-offset}}}{, stop
reason = ${thread.stop-reason}}{, name = ${thread.name}}{, queue =
${thread.queue}}\n'
```

In both thread and frame format definitions, use the `${X.script:/python_func/}` variables to specify Python functions that generate the display text (Section 4.2.1).

In all cases, the signature of `python_func` is expected to be:

```
def /python_func/(/object/,unused):
...
return /string/
```

Where `object` is an instance of the SB class associated with the keyword you are using.

For example, assuming your function looks like the following:

```
def thread_printer_func (thread,unused):
  return "Thread %s has %d frames\n" % (thread.name, thread.num_frames)
```

And you set it up with the following command:

```
(lldb) settings set thread-format "${thread.script:thread_printer_func}”
```

You will see the following output:

```
Thread main has 21 frames
```

5 Symbolication

LLDB performs symbolication on program crash logs to convert them into the file and symbol names used in the program's source code.

5.1 Manual symbolication with LLDB

Use LLDB to symbolicate your crash logs; it can often provide more information than other symbolication programs:

- Inlined functions
- Variables that are in scope for an address, along with their locations

The simplest form of symbolication is to load an executable:

```
(lldb) target create --no-dependents --arch x86_64 /tmp/a.out
```

The `--no-dependents` option is used with the `target create` command, so you do not load all the dependent shared libraries from the current system. When you symbolicate, you are often symbolicating a binary that was running on another system, and even though the main executable might reference shared libraries in `/usr/lib`, you often do not want to load the versions on the current computer.

The `image list` command displays a list of all shared libraries associated with the current target. As expected, there is only a single binary:

```
(lldb) image list
[  0] 73431214-6B76-3489-9557-5075F03E36B4 0x0000000100000000 /tmp/a.out
/tmp/a.out.dSYM/Contents/Resources/DWARF/a.out
```

Now you can look up an address:

```
(lldb) image lookup --address 0x100000aa3
Address: a.out[0x0000000100000aa3] (a.out.__TEXT.__text + 131)
Summary: a.out`main + 67 at main.c:13
```

Because no slide (Section 5.2) or load addresses were specified for individual sections in the binary, the address used here is a *file address*. A file address refers to a virtual address as defined by each object file.

If you do not use `--no-dependents` with the target create command, LLDB loads all the dependent shared libraries:

```
(lldb) image list
[  0] 73431214-6B76-3489-9557-5075F03E36B4 0x0000000100000000 /tmp/a.out/
tmp/a.out.dSYM/Contents/Resources/DWARF/a.out
[  1] 8CBCF9B9-EBB7-365E-A3FF-2F3850763C6B 0x0000000000000000/usr/lib/
system/libsystem_c.dylib
[  2] 62AA0B84-188A-348B-8F9E-3E2DB08DB93C 0x0000000000000000/usr/lib/
system/libsystem_dnssd.dylib
[  3] C0535565-35D1-31A7-A744-63D9F10F12A4 0x0000000000000000/usr/lib/
system/libsystem_kernel.dylib
...
```

Performing a lookup using a file address can result in multiple matches because most shared libraries have a virtual address space that starts at zero:

```
(lldb) image lookup -a 0x1000
Address: a.out[0x0000000000001000] (a.out.__PAGEZERO + 4096)
Address: libsystem_c.dylib[0x0000000000001000](libsystem_c.dylib.__TEXT.
__text + 928)
Summary: libsystem_c.dylib`mcount + 9

Address:
libsystem_dnssd.dylib[0x0000000000001000](libsystem_dnssd.dylib.__TEXT.
__text + 456)
Summary: libsystem_dnssd.dylib`ConvertHeaderBytes + 38

Address:
libsystem_kernel.dylib[0x0000000000001000](libsystem_kernel.dylib.__TEXT
.__text + 1116)
Summary: libsystem_kernel.dylib`clock_get_time + 102
...
```

To avoid getting multiple file address matches, specify the name of the shared library to limit the search:

```
(lldb) image lookup -a 0x1000 a.out
Address: a.out[0x0000000000001000] (a.out.__PAGEZERO + 4096)
```

5.2 Define load addresses for sections

When symbolicating your crash logs, it can be tedious if you must always adjust your crash log addresses into file addresses. To avoid performing any conversion, you can set the load address for the sections of the modules in your target. Once you set any section load address, lookups will switch to using load addresses.

You can also slide all sections in the executable by the same amount, or set the load address for individual sections. Use the `target modules load --slide` command to set the load address for all sections.

The following example slides all sections in file `a.out` by adding `0x123000` to each section's file address:

```
(lldb) target create --no-dependents --arch x86_64 /tmp/a.out
(lldb) target modules load --file a.out --slide 0x123000
```

It is often easier to specify the actual load location of each section by name.

5.3 Load multiple executables

Often you have more than one executable involved when you need to symbolicate a crash log. When this happens, create a target for the main executable or one of the shared libraries, and then add more modules to the target using the target modules `add` command.

For example, consider a crash log which contains the following images:

```
Binary Images:
0x100000000    - 0x100000ff7 <A866975B-CA1E-3649-98D0-6C5FAA444ECF>/tmp/
a.out
0x7fff83f32000 - 0x7fff83ffefe7 <8CBCF9B9-EBB7-365E-A3FF-2F3850763C6B>/
usr/lib/system/libsystem_c.dylib
0x7fff883db000 - 0x7fff883e3ff7 <62AA0B84-188A-348B-8F9E-3E2DB08DB93C>/
usr/lib/system/libsystem_dnssd.dylib
0x7fff8c0dc000 - 0x7fff8c0f7ff7 <C0535565-35D1-31A7-A744-63D9F10F12A4>/
usr/lib/system/libsystem_kernel.dylib
```

To symbolicate this log, first create the target using the main executable, and then add any extra shared libraries:

```
(lldb) target create --no-dependents --arch x86_64 /tmp/a.out
(lldb) target modules add /usr/lib/system/libsystem_c.dylib
(lldb) target modules add /usr/lib/system/libsystem_dnssd.dylib
(lldb) target modules add /usr/lib/system/libsystem_kernel.dylib
```

If you have debug symbols in standalone files, you can specify their paths using the `--symfile` option for the `target create` and `target modules add` commands:

```
(lldb) target create --no-dependents --arch x86_64 /tmp/a.out --symfile
/tmp/a.out.dSYM
(lldb) target modules add /usr/lib/system/libsystem_c.dylib --symfile
/build/server/a/libsystem_c.dylib.dSYM
(lldb) target modules add /usr/lib/system/libsystem_dnssd.dylib --symfile
/build/server/b/libsystem_dnssd.dylib.dSYM
(lldb) target modules add /usr/lib/system/libsystem_kernel.dylib
--symfile /build/server/c/libsystem_kernel.dylib.dSYM
```

Next, set the load addresses for each __TEXT section (note the colors of the load addresses above and below) using the first address from the Binary Images section for each image:

```
(lldb) target modules load --file a.out 0x100000000
(lldb) target modules load --file libsystem_c.dylib 0x7fff83f32000
(lldb) target modules load --file libsystem_dnssd.dylib 0x7fff883db000
(lldb) target modules load --file libsystem_kernel.dylib 0x7fff8c0dc000
```

Any stack backtraces that have not yet been symbolicated can now be symbolicated using the `image lookup` command with the raw backtrace addresses.

Given the following raw backtrace:

```
Thread 0 Crashed:: Dispatch queue: com.apple.main-thread
0   libsystem_kernel.dylib   0x00007fff8a1e6d46 __kill + 10
1   libsystem_c.dylib        0x00007fff84597df0 abort + 177
2   libsystem_c.dylib        0x00007fff84598e2a __assert_rtn + 146
3   a.out                    0x0000000100000f46 main + 70
4   libdyld.dylib            0x00007fff8c4197e1 start + 1
```

The following load addresses can now be symbolicated:

```
(lldb) image lookup -a 0x00007fff8a1e6d46
(lldb) image lookup -a 0x00007fff84597df0
(lldb) image lookup -a 0x00007fff84598e2a
(lldb) image lookup -a 0x0000000100000f46
```

5.4 Get variable information

If you use the `--verbose` option with the `image lookup --address` command, you can get verbose information which often includes the locations of some of your local variables:

```
>(lldb) image lookup --address 0x100123aa3 --verbose
Address: a.out[0x0000000100000aa3] (a.out.__TEXT.__text + 110)
Summary: a.out`main + 50 at main.c:13
Module: file = "/tmp/a.out", arch = "x86_64"
CompileUnit: id = {0x00000000}, file = "/tmp/main.c", language = "ISO
C:1999"
Function: id = {0x0000004f}, name = "main", range = [0x0000000100000bc0-
0x0000000100000dc9)
FuncType: id = {0x0000004f}, decl = main.c:9, clang_type = "int (int,
const char **, const char **, const char **)"
Blocks: id = {0x0000004f}, range = [0x100000bc0-0x100000dc9)
        id = {0x000000ae}, range = [0x100000bf2-0x100000dc4)
LineEntry: [0x0000000100000bf2-0x0000000100000bfa): /tmp/main.c:13:23
Symbol: id = {0x00000004}, range = [0x0000000100000bc0-
0x0000000100000dc9), name="main"
Variable: id = {0x000000bf}, name = "path", type= "char [1024]", location
= DW_OP_fbreg(-1072), decl = main.c:28
Variable: id = {0x00000072}, name = "argc", type= "int", *location =
r13*, decl = main.c:8
Variable: id = {0x00000081}, name = "argv", type= "const char **",
*location = r12*, decl = main.c:8
Variable: id = {0x00000090}, name = "envp", type= "const char **",
*location = r15*, decl = main.c:8
Variable: id = {0x0000009f}, name = "aapl", type= "const char **",
*location = rbx*, decl = main.c:8
```

The interesting part of this listing is the variables: they are the parameters and local variables that are in scope for the address that was specified.

Crash logs often include register information for the first frame in each stack. Being able to reconstruct one or more local variables can help you decipher more information from a crash log than you typically would be able to. In reality, this is only useful for the first frame, and only if your crash logs have register information for your threads.

5.5 Us Python API to symbolicate

You can perform all the commands demonstrated in the previous sections using the Python script bridge. The following code recreates the target and adds the three shared libraries that were added in the crash log example in Section 5.4:

```
triple = "x86_64-apple-macosx"
platform_name = None
add_dependents = False
target = lldb.debugger.CreateTarget("/tmp/a.out", triple, platform_name,
add_dependents, lldb.SBError())
if target: # Get the executable module module =
target.GetModuleAtIndex(0)
target.SetSectionLoadAddress(module.FindSection("__TEXT"), 0x100000000)
module = target.AddModule ("/usr/lib/system/libsystem_c.dylib", triple,
None, "/build/server/a/libsystem_c.dylib.dSYM") target.
SetSectionLoadAddress(module.FindSection("__TEXT"), 0x7fff83f32000)
module = target.AddModule ("/usr/lib/system/libsystem_dnssd.dylib",
triple, None, "/build/server/b/libsystem_dnssd.dylib.dSYM")
target.SetSectionLoadAddress(module.FindSection("__TEXT"),
0x7fff883db000) module = target.AddModule ("/usr/lib/system/
libsystem_kernel.dylib", triple, None, "/build/server/c/
libsystem_kernel.dylib.dSYM") target.SetSectionLoadAddress
(module.FindSection("__TEXT"), 0x7fff8c0dc000)

load_addr = 0x00007fff8a1e6d46
# so_addr is a section offset address, or a lldb.SBAddress object so_addr
=   target.ResolveLoadAddress (load_addr)
# Get a symbol context for the section offset address which
# includes a module, compile unit, function, block, line entry, and
# symbol
sym_ctx = so_addr.GetSymbolContext (lldb.eSymbolContextEverything
print sym_ctx
```

5.6 Use built-in Python module to symbolicate

LLDB includes a module in the `lldb` package named `lldb.utils.symbolication`. This module contains numerous symbolication functions that simplify symbolication by allowing you to create objects to represent symbolication class objects such as:

```
lldb.utils.symbolication.Address
lldb.utils.symbolication.Section
lldb.utils.symbolication.Image
lldb.utils.symbolication.Symbolicator
lldb.utils.symbolication.Address
```

This class represents an address that will be symbolicated. It will cache any information that has been looked up: module, compile unit, function, block, line entry, symbol. It does this by having a `lldb.SBSymbolContext` as a member variable.

lldb.utils.symbolication.Section

This class represents a section that might get loaded in `lldb.utils.symbolication.Image`. It includes helper functions which enable you to set it from text that may have been extracted from a crash log file.

lldb.utils.symbolication.Image

This class represents a module that might get loaded into the target used for symbolication. This class contains the executable path, the optional symbol file path, the triple, and the list of sections that must be loaded if you choose to ask the target to load this image. Many of these objects will never be loaded into the target unless they are required by symbolication.

You often have a crash log that has 100 to 200 different shared libraries loaded, but your crash log stack backtraces only use a few of these shared libraries. Only the images that contain stack backtrace addresses are to be loaded in the target for symbolication.

Subclasses of this class are to override the `locate_module_and_debug_symbols` method:

```
class CustomImage(lldb.utils.symbolication.Image):
    def locate_module_and_debug_symbols (self):
      # Locate the module and symbol given the info found in
      # the crash log
```

Overriding this function allows clients to find the correct executable module and symbol files because they might reside on a build server.

lldb.utils.symbolication.Symbolicator

This class coordinates the symbolication process by loading only the `lldb.utils.symbolication.Image` instances that are to be loaded to symbolicate a supplied address.

6 Variable formatting

LLDB supports the custom formatting of variables as they are displayed in the debugger.

6.1 Variable display

LLDB has a data formatters subsystem that lets you define custom display options for variables.

Typically, when you type a frame variable or run some expression, LLDB automatically chooses the way to display your results on a per-type basis. For example:

```
(lldb) frame variable
(uint8_t) x = 'a'
(intptr_t) y = 124752287
```

However, in certain cases you may want to define a different style for the display of certain data types. To do so, you must give hints to the debugger regarding how the variables should be displayed. The LLDB `type` command allows you to do just that.

Using the `type` command, you can change the variable display to look like this:

```
(lldb) frame variable
(uint8_t) x = chr='a' dec=65 hex=0x41
(intptr_t) y = 0x76f919f
```

Several features are related to data visualization: formats, summaries, filters, synthetic children. To reflect this, the `type` command has five subcommands:

- `type format`
- `type summary`
- `type filter`
- `type synthetic`
- `type category`

These subcommands are meant to bind printing options to types. When variables are printed, LLDB first checks if custom printing options have been associated to a variable type, and if so, it uses them instead of the default format.

Each subcommand (except the `type category`) has four subcommands:

- `add` – Associate a new printing option to one or more types
- `delete` – Delete an existing association
- `list` – Provide a listing of all associations
- `clear` – Delete all associations

6.2 Type formats

Type formats enable you to quickly override the default format for displaying primitive types (the usual basic C/C++ types: `int`, `float`, `char`, and so on).

If you want all `int` variables in your program to print out as hexadecimal values, you can add a format to the `int` type. Enter the following at the LLDB command line:

```
(lldb) type format add --format hex int
```

The `--format` option (which can be shortened to `-f`) accepts a format name `<#formatstable>`. Then provide one or more types to apply to the new format.

A frequent scenario is that your program has a typedef for a numeric type that you know represents something that must be printed in a certain way. Again, you can add a format just to that typedef by using `type format add` with the name alias.

But things can quickly get hierarchical. Consider the following example:

```
typedef int A;
typedef A B;
typedef B C;
typedef C D;
```

Here you want to display all `A` types as hexadecimal, all `C` types as byte arrays, and leave the default types untouched (while this example might seem contrived, it is realistic in large software systems).

If you simply enter the following:

```
(lldb) type format add -f hex A
(lldb) type format add -f uint8_t[] C
```

The values of type `B` are shown as hexadecimal, and values of type `D` are shown as byte arrays:

```
(lldb) frame variable -T
(A) a = 0x00000001
(B) b = 0x00000002
(C) c = {0x03 0x00 0x00 0x00}
(D) d = {0x04 0x00 0x00 0x00}
```

This print formatting occurs because by default, LLDB cascades formats through typedef chains. You can use the `-C` option to avoid cascading, thus creating the two commands required to achieve your goal:

```
(lldb) type format add -C no -f hex A
(lldb) type format add -C no -f uint8_t[] C
```

Which provides the intended output:

```
(lldb) frame variable -T
(A) a = 0x00000001
(B) b = 2
(C) c = {0x03 0x00 0x00 0x00} (D) d = 4
```

Two additional options you should know about are `--skip-pointers` (`-p`) and `--skip-references` (`-r`). These options prevent LLDB from applying a format for type `T` to values of type `T*` and `T&` respectively:

```
(lldb) type format add -f float32[] int
(lldb) frame variable pointer *pointer -T
(int *) pointer = {1.46991e-39 1.4013e-45}
(int) *pointer = {1.53302e-42}
(lldb) type format add -f float32[] int -p
(lldb) frame variable pointer *pointer -T
(int *) pointer = 0x0000000100100180
(int) *pointer = {1.53302e-42}
```

While you can apply formats to pointers and references, they do no attempt to dereference the pointer and extract the value before applying the format. This means you are effectively formatting the address stored in the pointer itself, rather than the referenced value. For this reason, when defining formats, you might want to use the `-p` option.

To delete a custom format, type `type format delete` followed by the name of the type to which the format applies. Even if you defined the same format for multiple types on the same command, `type format delete` only removes the format for the type name specified as the argument.

To delete all formats, use `type format clear`. To see all the formats defined, use `type format list`.

However, if all you intend is to display one variable in a custom format, while leaving the others of the same type untouched, you can simply enter:

```
(lldb) frame variable counter -f hex
```

The value of counter is displayed as a hexadecimal number, and it displays this way until you either pick a different format or until you let your program run again.

6.2.1 Options

```
default
binary              | b
boolean             | B
bytes               | y
bytes with ASCII    | Y
c-string            | s
char[]
character           | c
character array     | a
complex float       | F
complex integer     | I
decimal             | i
enumeration         | E
float               | f
float32[]           | float64[]
hex                 | x
int8_t[]            | uint8_t[]
int16_t[]           | uint16_t[]
int32_t[]           | uint32_t[]
int64_t[]           | uint64_t[]
```

```
uint128_t[]
octal               | o
pointer             | p
printable character | C
unicode16           | U
unicode32
unsigned decimal    | u
```

Descriptions

default

Use the default LLDB algorithm to pick a format.

binary
b

Show the value as a sequence of bits.

boolean
B

Show the value as a TRUE/FALSE Boolean, using the customary rule that 0 is FALSE and everything else is TRUE.

bytes
y

Show the bytes one after the other. For example:

```
(int) s.x = 07 00 00 00
```

bytes with ASCII
Y

Show the bytes, but display them as ASCII characters as well. For example:

```
(int *) c.sp.x = 50 f8 bf 5f ff 7f 00 00 P.._....
```

c-string
s

Show the value as a zero-terminated C string.

char[]

Show the value as an array of characters. For example:

```
(char) *c.sp.z = {X}
```

character
c

Show the bytes as ASCII characters. For example:

```
(int *) c.sp.x = P\xf8\xbf_\xff\x7f\0\0
```

character array
a

Show the value as a character array. For example:

```
(int *) pointer = \x80\x01\x10\0\x01\0\0\0
```

complex float
F

Interpret the value as the real and imaginary part of a complex floating-point number. For example:

```
(int *) c.sp.x = 2.76658e+19 + 4.59163e-41i
```

complex integer
I

Interpret the value as the real and imaginary parts of a complex integer number. For example:

```
(int *) pointer = 1048960 + 1i
```

decimal
i

Show the value as a signed integer number (this option does not perform a cast; it simply shows the bytes as an integer with sign).

enumeration
E

Show the value as an enumeration, printing the value's name if available or the integer value otherwise. For example:

```
(enum enumType) val_type = eValue2
```

float
f

Show the value as a floating-point number (this option does not perform a cast; it only interprets the bytes as an IEEE754 floating point value).

float32[]
float64[]

Show as array of the corresponding floating-point type. For example:

```
(int *) pointer = {1.46991e-39 1.4013e-45}
```

hex
x

Show the value in hexadecimal notation (this option does not perform a cast; it simply shows the bytes as hex).

int8_t[]
uint8_t[]
int16_t[]
uint16_t[]
int32_t[]
uint32_t[]
int64_t[]
uint64_t[]
uint128_t[]

Show the values as an array of the corresponding integer types. For example:

```
(int) x = {1 0 0 0} (with uint8_t[]) (int) y = {0x00000001} (with
uint32_t[])
```

octal
o

Show the value in octal notation.

pointer
p

Show the value as a native pointer (unless the value is really a pointer, the resulting address will probably be invalid).

printable character
C

Show the bytes as printable ASCII characters. For example:

```
(int *) c.sp.x = P.._....
```

unicode16
U

Show the value as UTF-16 characters. For example:

```
(float) x = 0xd70a 0x411f
```

unicode32

Show the value as UTF-32 characters. For example:

```
(float) x = 0x411fd70a
```

unsigned decimal
u

Show the value as an unsigned integer number (this option does not perform a cast; it simply shows the bytes as an unsigned integer).

6.3 Type summaries

Type formats work by showing a different kind of display for the value of a variable. However, they only work for basic types.

To display a class or structure type in a custom format, you must use a different feature known as *type summaries*. This feature works by extracting information from classes, structures, and other aggregate types, and arranging them in a user-defined format.

For example, before adding a type summary:

```
(lldb) frame variable -T one
(i_am_cool) one = {
   (int) x = 3
   (float) y = 3.14159
   (char) z = 'E' }
```

After adding a summary:

```
(lldb) frame variable one
(i_am_cool) one = int = 3, float = 3.14159, char = 69
```

There are two ways to use type summaries:

- Bind a summary string to the type
- Write a Python script that returns the string to be used as summary

Both options are enabled by the `type summary add` command. The following obtains the output shown in the example:

```
(lldb) type summary add --summary-string "int = ${var.x}, float =
${var.y}, char = ${var.z%u}"
i_am_cool
```

6.3.1 Summary strings

Summary strings are written using a simple control language, exemplified by the snippet in the previous section. A summary string contains a sequence of tokens that are processed by LLDB to generate the summary.

Summary strings can contain plain text, control characters, and special variables that have access to information about the current object and the overall program state.

- Plain text is any sequence of characters that does not contain a `{`, `}`, `$`, or `\` character, which are the control characters (Section 4.2.2).
- Special variables begin with a `${` prefix, and end with a `}` suffix. Variables can be either a simple name, or they can refer to complex objects that themselves have sub-items. In other words, a variable looks like `${object}` or `${object.child.otherchild}`.
- A variable can also be prefixed or suffixed with other symbols meant to change the way its value is handled. An example is `${var.int_pointer[0-3]}`.

The syntax is the same as described in Chapter 4, plus additional symbols specific to summary strings. The main addition is `${var`, which is used to refer to the variable for which a summary is being created.

The simplest thing you can do is get a member variable of a class or structure by typing its *expression path*. In the previous example, the expression path for the field float `y` is simply `.y`. Thus, to use the summary string to display `y`, type `${var.y}`.

In the following code, the expression path for the `y` member of the `x` member of an object of type `B` is `.x.y`. Type `${var.x.y}` to display it in a summary string for type `B`:

```
struct A {
   int x;
   int y;
};
struct B {
   A x;
   A y;
   int z;
};
```

By default, a summary defined for type `T`, also works for types `T*` and `T&` (you can disable this behavior). For this reason, expression paths do not differentiate between `.` and `->`; the expression path above, `.x.y`, is as good as if you were displaying a `B*`, or even if the actual definition of `B` is:

```
struct B {
   A *x;
   A y;
   int *z;
};
```

This example is unlike the behavior of a frame variable which, on the contrary, enforces the distinction. As hinted above, the rationale for this choice is that waiving this distinction allows you to write a summary string once for type `T`, and use it for both `T` and `T*` instances. Because a summary string is about extracting nested members' information, a pointer to an object is as good as the object itself for the purpose.

To access the value of the integer pointed to by `B::z`, you cannot simply specify `${var.z}`, because that symbol refers to the pointer `z`. To dereference it and get the pointed value, specify `${var.z}`. The `${var` directs LLDB to first get the object that the expression paths leads to, and then dereference it. In this example, it is equivalent to `(bObject.z)` in C/C++ syntax.

Because the `.` and `->` operators can both be used, dereferences are not required in the middle of an expression path (for example, do not specify `${*(var.x).x})` to read `A::x` as contained in `*(B::x)`. To achieve that effect, simply specify `${var.x->x}`, or even `${var.x.x}`.

A summary string can contain more than one `${var` specifier, and can use `${var` and `${*var` specifiers together.

NOTE: The `*` operator only binds to the result of the whole expression path, rather than piecewise, and using parentheses will not change that behavior.

6.3.2 Formatting summary elements

An expression path can include formatting codes. Much like the type formats discussed previously, you can customize the way variables are displayed in summary strings, regardless of the format they have applied to their types. To do that, use `%/format/` inside an expression path, as in `${var.x->x%u}`, which would display the value of `x` as an unsigned integer.

You can also use other special format markers, which are not available for formats themselves, but which carry a special meaning when used in this context.

Table 6-1 Special format markers

Symbol	Description
`%S`	Object summary (the default for aggregate types)
`%V`	Object value (the default for non-aggregate types)
`%@`	Language runtime-specific description (for C++, this does nothing)
`%L`	Object location (memory address, register name, and so on)
`%#`	Number of children of this object
`%T`	Name of the object data type

Using the `--inline-children` (`-c`) option with `type summary add` directs LLDB to not search for a summary string, but instead to just print a listing of all the object's children on one line. For example, given the following type pair:

```
(lldb) frame variable --show-types a_pair
(pair) a_pair = {
    (int) first = 1;
    (int) second = 2;
}
```

If you enter the following command:

```
(lldb) type summary add --inline-children pair
```

The output becomes:

```
(lldb) frame variable a_pair
(pair) a_pair = (first=1, second=2)
```

Of course, you can obtain the same effect by entering the following:

```
(lldb) type summary add pair --summary-string "(first=${var.first},
second=${var.second})"
```

While the result is the same, using `--inline-children` can often save time. To see the values of the variables, not their names, combine the `--omit-names`, or `-O` (uppercase letter), option with `--inline-children` to get the following:

```
(lldb) frame variable a_pair
(pair) a_pair = (1, 2)
```

Which is equivalent to:

```
(lldb) type summary add pair --summary-string "(${var.first},
${var.second})"
```

6.3.3 Bit fields and array syntax

Sometimes a basic type's value represents several different values packed together in a bit field. With the classical view, there is no way to look at them. Hexadecimal display can help, but if the bits span nibble boundaries, the help is limited. Binary view would show it all without ambiguity, but is often too detailed and hard to read for real-life scenarios.

To cope with this issue, LLDB supports native bit field formatting in summary strings. If your expression paths leads to a so-called scalar type (the usual int, float, char, double, short, long, long long, double, long double and unsigned variants), you can direct LLDB to grab only some bits out of the value and display them in any format you like. If you only need one bit, you can use the `[/n/]` notation, just as in indexing an array.

To extract multiple bits, you can use a slice-like syntax: `[/n/-/m/]`. For example:

```
(lldb) frame variable float_point
(float) float_point = -3.14159

(lldb) type summary add --summary-string "Sign: ${var[31]%B} Exponent:
${var[30-23]%x}
Mantissa: ${var[0-22]%u}" float

(lldb) frame variable float_point
(float) float_point = -3.14159 Sign: true Exponent: 0x00000080 Mantissa:
4788184
```

In this example, LLDB shows the internal representation of a float variable by extracting bit fields out of a float object.

When typing a range, the extremes n and m are always included, and the order of the indices is irrelevant.

LLDB also allows you to use a similar syntax to display array members inside a summary string. For instance, you may want to display all arrays of a given type using a more compact notation than the default, and then access just the individual array members that prove interesting to your debugging task. You can direct LLDB to format arrays in special ways, independent of the way the array the members' data type is formatted. For example:

```
(lldb) frame variable sarray
(Simple [3]) sarray = {
   [0] = {
      x = 1
      y = 2
      z = '\x03'
   }
   [1] = {
      x = 4
      y = 5
      z = '\x06'
   }
   [2] = {
      x = 7
      y = 8
      z = '\t'
   }
}
```

```
(lldb) type summary add --summary-string "${var[].x}" "Simple [3]"

(lldb) frame variable sarray
(Simple [3]) sarray = [1,4,7]
```

The `[]` symbol is defined as follows: if var is an array and its size is known, apply this summary string to every element of the array. In the example above, LLDB is directed to display `.x` for every element of the array. If you find one or more of those integers anomalous, you can inspect those items in greater detail, without the array format getting in the way:

```
(lldb) frame variable sarray[1]
(Simple) sarray[1] = {
   x = 4
   y = 5
   z = '\x06'
}
```

You can also direct LLDB to print only a subset of the array range by using the same syntax used to extract bit for bit fields:

```
(lldb) type summary add --summary-string "${var[1-2].x}" "Simple [3]"

(lldb) frame variable sarray
(Simple [3]) sarray = [4,7]
```

If you are dealing with a pointer that you know is an array, you can use this syntax to display the elements contained in the pointed array instead of just the pointer value. However, because pointers have no notion of their size, the empty brackets `[]` operator does not work, and you must explicitly provide higher and lower bounds.

In general, LLDB needs the square brackets operator `[]` to handle arrays and pointers correctly, and for pointers it also needs a range. However, a few special cases are defined to make your life easier:

You can print a zero-terminated string (C-string) using the `%s` format, omitting square brackets, as in:

```
(lldb) type summary add --summary-string "${var%s}" "char *"
```

This syntax works for `char*` as well as for `char[]` because LLDB can rely on the final `\0` terminator to know when the string has ended.

LLDB has default summary strings for `char*` and `char[]` that use this special case. On debugger startup, the following are defined automatically:

```
(lldb) type summary add --summary-string "${var%s}" "char *"
(lldb) type summary add --summary-string "${var%s}" -x "char \[[0-
9]+]"
```

Any of the array formats (`int8_t[]`, `float32{}`, `...`), and the `y`, `Y` and a formats work to print an array of a non-aggregate type, even if square brackets are omitted.

```
(lldb) type summary add --summary-string "${var%int32_t[]}" "int [10]"
```

This feature, however, is not enabled for pointers because there is no way for LLDB to detect the end of the pointed data. This also does not work for other formats (such as Boolean), and you must specify the square brackets operator to get the expected output.

6.4 Python scripts

Summary strings are typically good enough for summarizing the contents of a variable. However, to do more than simply picking some values and rearranging them for display, summary strings are not effective because they lack the power to perform any kind of computation on the value of variables.

To solve this issue, you can bind some Python scripting code as a summary for your data type, and that script can both extract children variables as the summary strings do, and to perform active computation on the extracted values. For example:

```
class Rectangle
{
private:
   int height;
   int width;
public:
   Rectangle() : height(3), width(5) {}
   Rectangle(int H) : height(H), width(H*2-1) {}
   Rectangle(int H, int W) : height(H), width(W) {}
   int GetHeight() { return height; }
   int GetWidth() { return width; }
};
```

Summary strings are effective to reduce the screen real estate used by the default viewing mode, but they are not effective for displaying the area and perimeter of Rectangle objects. In this case, attach a small Python script to the Rectangle class:

```
(lldb) type summary add -P Rectangle
Enter your Python command(s). Type 'DONE' to end.
def function (valobj,internal_dict):
   height_val = valobj.GetChildMemberWithName('height')
   width_val = valobj.GetChildMemberWithName('width')
   height = height_val.GetValueAsUnsigned(0)
   width = width_val.GetValueAsUnsigned(0)
   area = height*width
   perimeter = 2*(height + width)
   return 'Area: ' + str(area) + ', Perimeter: ' + str(perimeter)
DONE
(lldb) frame variable
(Rectangle) r1 = Area: 20, Perimeter: 18
(Rectangle) r2 = Area: 72, Perimeter: 36
(Rectangle) r3 = Area: 16, Perimeter: 16
```

To write effective summary scripts, you must know the LLDB public API, which is the way Python code can access the LLDB object model.

NOTE: For further details on the API see the rest of this chapter or the LLDB API reference documentation (http://lldb.llvm.org/python_reference/index.html).

As a brief introduction, your script is encapsulated as a function that is passed two parameters, `valobj` and `internal_dict`:

- `valobj` is the object encapsulating the actual variable being displayed, and its type is `SBValue`.
- `internal_dict` is an internal support parameter used by LLDB, and you should not touch it.

Of the many possible operations on an `SBValue`, the most basic operation is to retrieve the children objects that it contains (which are the fields of the object that is wrapped by it), by calling `GetChildMemberWithName()` and passing it the child's name as a string.

If the variable has a value, you can ask for it and return it as a string using `GetValue()` or as a signed/unsigned number using `GetValueAsSigned()`, `GetValueAsUnsigned()`.

You can also retrieve an `SBData` object by calling `GetData()` and then reading the object's contents out of the `SBData`.

To traverse several levels of hierarchy, as you can do with summary strings, use the method `GetValueForExpressionPath()`, passing it an expression path similar to those used in summary strings (one of the differences is that dereferencing a pointer does not occur by prefixing the path with a *, but by calling the Dereference() method on the returned `SBValue`).

To access array slices, you cannot do that (yet) via this method call, and instead must use `GetChildAtIndex()` querying it for the array items one by one. Also, handling custom formats is something you must deal with on your own.

Other than interactively typing a Python script, there are two other ways for you to input a Python script as a summary:

- Using the `--python-script` option with type `summary add`, and entering the script code as an option argument:

```
(lldb) type summary add --python-script "height = valobj.
GetChildMemberWithName('height').GetValueAsUnsigned(0);
width = valobj.GetChildMemberWithName('width').GetValueAsUnsigned(0);
return 'Area: %d' % (height*width)" Rectangle
```

- Using the `--python-function` (`-F`) option with type `summary add`, and specifying the name of a Python function with the correct prototype.

You will probably define (or have already defined) the function in the interactive interpreter, or somehow loaded it from a file, using the command script import command. LLDB will generate a warning if it cannot find the function you passed, but it will still register the binding.

6.5 Regular expression type names

To associate the custom summary string to the array types, you must specify the array size as part of the type name. This can become tiresome when using arrays of different sizes, such as Simple [3], Simple [9], Simple [12], and so on.

If you use the `-x` option, type names are treated as regular expressions instead of type names. This would let you rephrase the above example for arrays of type Simple [3] as:

```
(lldb) type summary add --summary-string "${var[].x}" -x "Simple \[[0-
9]+\]"

(lldb) frame variable
(Simple [3]) sarray = [1,4,7]
(Simple [2]) sother = [3,6]
```

The above scenario works for Simple [3] as well as for any other array of Simple objects.

While this feature is mostly useful for arrays, you can also use regular expressions to catch other type sets grouped by name. However, as regular expression matching is slower than normal name matching, LLDB will first try to match by name in any way it can, and only when this fails will it resort to regular expression matching.

One of the ways LLDB uses this feature internally, is to match the names of STL container classes, regardless of the template arguments provided. The details for this are in file FormatManager.cpp.

The regular expression language used by LLDB is the *POSIX extended language*, as defined by the *Single UNIX Specification* (http://pubs.opengroup.org/onlinepubs/7908799/xsh/regex.h.html).

6.6 Named summaries

For a given type, there may be different meaningful summary representations. Currently, however, only one summary can be associated to a type at each moment. To temporarily override the association for a variable without changing the summary string for to its type, use named summaries.

Named summaries work by attaching a name to a summary when creating it. Then, when the summary is to be attached to a variable, the `frame variable` command supports a `--summary` option that directs LLDB to use the named summary given instead of the default summary. For example:

```
(lldb) type summary add --summary-string "x=${var.integer}" --name
NamedSummary

(lldb) frame variable one
(i_am_cool) one = int = 3, float = 3.14159, char = 69
(lldb) frame variable one --summary NamedSummary
(i_am_cool) one = x=3
```

When defining a named summary, binding it to one or more types becomes optional. Even if you bind the named summary to a type, and later change the summary string for that type, the named summary will not be changed by that. You can delete named summaries by using the type summary delete command, as if the summary name was the data type that the summary is applied to

A summary attached to a variable using the `--summary` option, has the same semantics that a custom format attached using the `-f` option has: it stays attached until you attach a new one, or until you let your program run again.

6.7 Synthetic children

Summaries work well when you can navigate through an expression path. For LLDB to do this, the appropriate debugging information must be available.

Some types are opaque, i.e. no knowledge of their internals is provided. When that is the case, expression paths do not work correctly.

In other cases, the internals are available to use in expression paths, but they do not provide a user-friendly representation of the object's value.

For instance, consider an STL vector, as implemented by the *GNU C++ Library* (http://gcc.gnu.org/onlinedocs/libstdc++/):

```
(lldb) frame variable numbers -T
(std::vector<int>) numbers = {
   (std::_Vector_base<int, std::allocator<int> >)
std::_Vector_base<int, std::allocator<int> > = {
(std::_Vector_base<int, std::allocator&tl;int> >::_Vector_impl) _M_impl =
{
          (int *) _M_start = 0x00000001001008a0
          (int *) _M_finish = 0x00000001001008a8
          (int *) _M_end_of_storage = 0x00000001001008a8
      }
   }
}
```

Here you can see how the type is implemented, and you can write a summary for that implementation. But, it will not help you infer what items are actually stored in the vector.

What you would like to see is probably something like:

```
(lldb) frame variable numbers -T
(std::vector<int>) numbers = {
   (int) [0] = 1
   (int) [1] = 12
   (int) [2] = 123
   (int) [3] = 1234
}
```

Synthetic children are a way to get that result.

The feature is based on the idea of providing a new set of children for a variable that replaces the ones available by default through the debug information. The example uses synthetic children to provide the vector items as children for the `std::vector` object.

To create synthetic children, you must provide a Python class that adheres to a given *interface*[1].

For example, consider the following footnoted example:

```
class SyntheticChildrenProvider:
   def __init__(self, valobj, internal_dict):
      /this call should initialize the Python object using valobj as the
       variable to provide synthetic children for/
   def num_children(self):
```

[1] The term is italicized because Python has no explicit notion of interface in the sense of a given set of methods that must be implemented by a Python class (http://en.wikipedia.org/wiki/Duck_typing).

```
        /this call should return the number of children that you want your
         object to have/
    def get_child_index(self,name):
       /this call should return the index of the synthetic child whose name
       is given as argument/
    def get_child_at_index(self,index):
       /this call should return a new LLDB SBValue object representing the
         child at the index given as argument/
    def update(self):
       /this call should be used to update the internal state of this
         Python object whenever the state of the variables in LLDB
         changes./[1]
    def has_children(self):
       /this call should return True if this object might have children,
         and False if this object can be guaranteed not to have
         children./[2]
```

[1] This method is optional. Also, it can optionally choose to return a value (starting with SVN rev153061/LLDB-134). If it returns a value and that value is TRUE, LLDB can cache the children and the children count that it previously obtained, and it will not return to the provider class to ask. If nothing (None) or anything other than TRUE is returned, LLDB discards the cached information and asks. Regardless, when necessary, LLDB will call update.

[2] This method is optional (starting with SVN rev166495/LLDB-175). While implementing it in terms of `num_children` is acceptable, you are encouraged to look for optimized coding alternatives when appropriate.

For examples of how synthetic children are created, look at `examples/synthetic` in the LLDB trunk (http://llvm.org/svn/llvm-project/lldb/trunk/examples/synthetic/). Be aware that the code in those files (except `bitfield/`) is legacy code and is not maintained. Look at `this example` to get a feel for this feature because it is an easy and well-commented example.

The design pattern consistently used in synthetic providers shipped with LLDB is to use the `__init__` to store the `SBValue` instance as a part of `self`. The `update` function is then used to perform the actual initialization.

After writing a synthetic children provider, you must load it into LLDB before it can be used:

- Use the LLDB script command to type Python code interactively.
- Or, use the command script import command to load Python code from a Python module (ordinary rules apply to importing modules this way).
- Or, type the code for the provider class interactively while adding it.

For example, consider a `Foo` class that makes available a synthetic children `Foo_Provider` class in a Python module contained in the `~/Foo_Tools.py` file. The following interaction sets `Foo_Provider` as a synthetic children provider in LLDB:

```
(lldb) command script import ~/Foo_Tools.py
(lldb) type synthetic add Foo --python-class Foo_Tools.Foo_Provider

(lldb) frame variable a_foo
(Foo) a_foo = {
x = 1
y = "Hello world"
}
```

LLDB has synthetic children providers for a core subset of STL classes, both in the version provided by `libstdcpp` (http://gcc.gnu.org/libstdc++/) and `libcxx` (http://libcxx.llvm.org/), as well as for several Foundation classes.

Synthetic children extend summary strings by enabling a new special variable: `${svar`. This symbol tells LLDB to refer expression paths to the synthetic children instead of the real children. For example:

```
(lldb) type summary add --expand -x "std::vector<" --summary-string
"${svar%#} items"

(lldb) frame variable numbers
(std::vector<int>) numbers = 4 items {
   (int) [0] = 1
   (int) [1] = 12
   (int) [2] = 123
   (int) [3] = 1234
}
```

In some cases, if LLDB cannot use the real object to get a child specified in an expression path, it will automatically refer to the synthetic children. While in summaries it is best to always use `${svar` to make your intentions clearer, interactive debugging can benefit from this behavior. For example:

```
(lldb) frame variable numbers[0] numbers[1]
(int) numbers[0] = 1
(int) numbers[1] = 12
```

Unlike many other visualization features, however, the access to synthetic children only works when using the `frame` variable, and it is not supported in the `expression` command:

```
(lldb) expression numbers[0]
Error [IRForTarget]: Call to a function '_ZNSt33vector<int,
std::allocator<int> >ixEm' that is not present in the target
error: Couldn't convert the expression to DWARF
```

The reason for this limitation () is that classes might have an overloaded operator `[]`, or other special provisions, and the expression command chooses to ignore synthetic children in the interest of equivalence with code you asked to be compiled from source.

As shown in this example, the `expression` command cannot parse and display some variables because the classes might have an overloaded operator `[]` or other special provision. Thus, the `expression` command ignores synthetic children in the interest of equivalence with the code you asked to be compiled from source.

6.8 Filters

Filters are a solution to the display of complex classes. At times classes have many member variables, but not all of them are necessary for you to see.

A filter solves this issue by letting you see only those member variables you care about. Of course, you can easily implement the equivalent of a filter using synthetic children, but a filter lets you do the job without writing Python code.

For example, if your class `Foobar` has member variables named A through Z, but you only need to see the variables named `B`, `H`, and `Q`, you can define the following filter:

```
(lldb) type filter add Foobar --child B --child H --child Q

(lldb) frame variable a_foobar
(Foobar) a_foobar = {
   (int) B = 1
   (char) H = 'H'
   (std::string) Q = "Hello world"
}
```

6.9 Categories

Categories are a way to group related formatters. For instance, LLDB groups the formatters for the `libstdc++` types in a category named `gnu-libstdc++`. Basically, categories serve as containers in which to store formatters for the same library or OS release.

By default, several categories are created in LLDB:

- Default

  The category where every formatter ends up, unless another category is specified.

- `gnu-libstdc++`

  Formatters for `std::string`, `std::vector`, `std::list`, and `std::map` as implemented by `libstdcpp` .

- `libcxx`

  Formatters for `std::string`, `std::vector`, `std::list`, and `std::map` as implemented by `libcxx`.

- `system`

  Truly basic types for which a formatter is required.

- `VectorTypes`

  Compact display for several vector types.

The `type` command has several options that also include an `add` option:

```
type filter add
type format add
type summary add
type synthetic add
```

To specify a custom category in which to add the formatter, use a `type` *`<option>`* `add` command with the `--category` (`-w`) option. For example, the following command automatically creates a (disabled) category named `newcategory`:

```
(lldb) type summary add Foobar --summary-string "a foobar" --category
newcategory
```

To delete the formatter, you must specify the correct category.

Categories can be in one of two states: enabled or disabled. A category is initially disabled and can be enabled using the `type category enable` command. To disable an enabled category, use the **`type category disable`** command.

The order in which categories are enabled or disabled is significant, because LLDB uses that order when looking for formatters. Therefore, when you enable a category, it becomes the second one to be searched (after default, which always stays on top of the list). The default categories are enabled in such a way to use the following search order:

1. default
2. objc
3. gnu-libstdc++
4. libcxx

5. VectorTypes
6. system

As noted, `gnu-libstdc++` and `libcxx` contain formatters for C++ STL data types. system contains formatters for `char*` and `char[]`, which reflect the behavior of older versions of LLDB which had built-in formatters for these types. Because these are now formatters, you can even replace them with your own versions.

No special command exists to create a category. When you place a formatter in a category, if that category does not exist, it is automatically created. For example:

```
(lldb) type summary add Foobar --summary-string "a foobar" --category
newcategory
```

This command automatically creates a (disabled) category named `newcategory`.

Another way to create a new (empty) category is to enable it, as in:

```
(lldb) type category enable newcategory
```

However, in this case LLDB warns you that enabling an empty category has no effect. If you add formatters to the category after enabling it, they will be honored. But an empty category itself does not change the way any type is displayed. The reason the debugger warns you is because enabling an empty category might be a typo, and you effectively wanted to enable a similarly-named but not-empty category.

6.10 Finding formatters 101

Given a variable, the process of searching for a formatter (including formats, starting in SVN rev r192217[1]) involves a rather intricate set of rules. LLDB starts looking in each enabled category, according to the order in which they were enabled (latest enabled first). In each category, LLDB does the following:

1. If there is a formatter for the type of the variable, use it.
2. If this object is a pointer, and there is a formatter for the pointee type that does not skip pointers, use it.
3. If this object is a reference, and there is a formatter for the referred type that does not skip references, use it.
4. If this object type is a typedef, go through the typedef hierarchy. (LLDB might not be able to do this if the compiler has not emitted enough information.) If the required information to traverse typedef hierarchies is missing, type cascading will not work. (The LLVM Clang compiler emits the correct debugging information for LLDB to cascade.) If at any level of the hierarchy there is a valid formatter that can cascade, use it.
5. If everything has failed, repeat the above search, looking for regular expressions instead of exact matches.
6. If any of those attempts returned a valid formatter to be used, that one is used, and the search is terminated (without going to look in other categories). If nothing was found in the current category, the next enabled category is scanned according to the same algorithm. If there are no more enabled categories, the search has failed.

NOTE: Previous versions of LLDB defined cascading to mean not only going through typedef chains, but also through inheritance chains. This feature has been removed since it significantly degrades performance. You must set up your formatters for every type in inheritance chains to which you want the formatter to apply.

[1] http://llvm.org/viewvc/llvm-project?view=revision&revision=192217

7 Python scripting

LLDB supports controlling the debugger from a Python script.

7.1 LLDB API

The entire LLDB API is available as Python functions through a script bridging interface. This means you can use the LLDB API directly from Python, either interactively or to build Python apps that provide debugger features.

Additionally, you can use Python as a programmatic interface in the LLDB command interpreter (for brevity, this interpreter is referred to as the *embedded interpreter*). In this context it has full access to the LLDB API.

The LLDB API is contained in a Python module named `lldb`. When writing Python extensions, a useful resource is the *LLDB Python Classes Reference Guide* (http://lldb.llvm.org/python_reference/index.html).

The documentation is also accessible in an interactive debugger session with the following command:

```
(lldb) script help(lldb)
```

You can also get help using a module class name. The full API that is exposed for that class will be displayed in a man page style window. To get help on the `lldb.SBFrame` class:

```
(lldb) script help(lldb.SBFrame)
```

Or you can get help using any Python object. In the following example, the `lldb.process` object is used, which is a global variable in the `lldb` module that represents the currently-selected process:

```
(lldb) script help(lldb.process)
```

7.2 Embedded Python interpreter

You can access the embedded Python interpreter in a variety of ways from within LLDB. The easiest way is to use the LLDB `script` command with no arguments at the LLDB command prompt:

```
(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()' or
Ctrl-D.
>>> 2+3
5
>>> hex(12345)
'0x3039'
>>>
```

This script drops you into the embedded Python interpreter. When running under the script command, LLDB sets some convenience variables which give you quick access to the currently-selected entities that characterize the program and debugger state. In each case, if there is no currently-selected entity of the appropriate type, the variable's `IsValid` method will return FALSE.

Table 7-1 Python convenience variables

Variable	Type	Description
`lldb.debugger`	`lldb.SBDebugger`	Contains the debugger object whose script command was invoked. The lldb.SBDebugger object owns the command interpreter and all the targets in your debug session. There will always be a Debugger in the embedded interpreter.
`lldb.target`	`lldb.SBTarget`	Contains the currently-selected target - for instance the one made with the file or selected by the target select <target-index> command. The lldb.SBTarget manages one running process, and all the executable and debug files for the process.
`lldb.process`	`lldb.SBProcess`	Contains the process of the currently-selected target. The lldb.SBProcess object manages the threads and allows access to memory for the process.
`lldb.thread`	`lldb.SBThread`	Contains the currently-selected thread. The lldb.SBThread object manages the stack frames in that thread. A thread is always selected in the command interpreter when a target stops. Use the thread select <thread-index> command to change the currently selected thread. As long as you have a stopped process, there will be some selected thread.

Table 7-1 Python convenience variables (cont.)

Variable	Type	Description
`lldb.frame`	`lldb.SBFrame`	Contains the currently-selected stack frame. The lldb.SBFrame object manages the stack locals and the register set for that stack. A stack frame is always selected in the command interpreter when a target stops. Use the frame select <frame-index> command to change the currently selected frame. As long as you have a stopped process, there will be some selected frame.

While extremely convenient, these variables have few restrictions on their use.

First, they only hold the values of the selected objects on entry to the embedded interpreter. Their values are not updated as you use the LLDB APIs to change, for example, the currently-selected stack frame or thread.

Second, they are only defined and meaningful while in the interactive Python interpreter. There is no guarantee on their value in any other situation, thus you should not use them when defining Python formatters, breakpoint scripts, or commands (or any other Python extension point that LLDB provides).

NOTE: As a rationale for such behavior, consider that LLDB can run in a multi-threaded environment, and another thread might call the script command, changing the value out from under you.

To assist you in getting started with objects and LLDB scripting, be aware that most of the LLDB Python objects can briefly describe themselves when they are passed to the Python print function:

```
(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()' or
Ctrl-D.
>>> print lldb.debugger
Debugger (instance: "debugger_1", id: 1)
>>> print lldb.target a.out
>>> print lldb.process
SBProcess: pid = 59289, state = stopped, threads = 1, executable = a.out
>>> print lldb.thread
SBThread: tid = 0x1f03
>>> print lldb.frame
frame #0: 0x0000000100000bb6 a.out main + 54 at main.c:16
```

7.3 Run a script when a breakpoint is hit

One powerful use of the LLDB Python API is to have a Python script run when a breakpoint is hit. Adding Python scripts to breakpoints provides a way to create complex breakpoint conditions, and they allow for smart logging and data gathering.

When your process hits a breakpoint to which you have attached some Python code, the code is executed as the body of a function which takes three arguments:

```
def breakpoint_function_wrapper(frame, bp_loc, dict):
  # Your code goes here
```

Table 7-2 Python breakpoint function arguments

Argument	Type	Description
`frame`	`lldb.SBFrame`	The current stack frame where the breakpoint got hit. The object will always be valid. This frame argument might match the currently selected stack frame found in the LLDB module global variable, `lldb.frame`.
`bp_loc`	`lldb.SBBreakpointLocation`	The breakpoint location that was just hit. Breakpoints are represented by `lldb.SBBreakpoint` objects, which can have one or more locations. These locations are represented by lldb.SBBreakpointLocation objects.
`dict`	`dict`	The Python session dictionary as a standard Python dictionary object.

Optionally, a Python breakpoint command can return a value. Returning False informs LLDB that you do not want to stop at the breakpoint. Any other return value (including None or leaving out the return statement altogether) is equivalent to directing LLDB to stop at the breakpoint. This command is useful in situations where a breakpoint is required to only stop the process when certain conditions are met, and you do not want to inspect the program state manually at every stop and then continue.

Writing some Python code and attaching it to a breakpoint is a fairly simple task. The following example shows how to track the order in which the functions in a given shared library are first executed during one run of your program.

```
(lldb) breakpoint set --func-regex=. --shlib=libfoo.dylib
Breakpoint created: 1: regex = '.', module = libfoo.dylib, locations =
223
(lldb) script counter = 0
(lldb) breakpoint command add --script-type python 1
Enter your Python command(s) and type 'DONE' to end.
> # Increment counter.  Since this is in a function, it must be a
global Python variable
> global counter
> counter += 1
> # Get the name of the function
> name = frame.GetFunctionName() >
# Print the order and the function name
```

```
> print '[%i] %s' % (counter, name)
> # Disable current breakpoint location so it doesn't get hit again >
bp_loc.SetEnabled(False)
> # No need to stop here
> return False
> DONE
```

This simple method gathers an order file that you can use to optimize function placement within a binary for execution locality. This is done by setting a regular expression breakpoint that will match every function in the shared library. The regular expression, `.`, will match any string that has at least one character in it, so it is used. This will result in one `lldb.SBBreakpoint` object that contains an `lldb.SBBreakpointLocation` object for each function.

When the breakpoint is hit, a counter is used to track the order in which the function at this breakpoint location was hit. Because the code is past the location that was hit, you can get the name of the function from the location, disable the location so you will not count this function again, and then log some information and continue the process.

NOTE: The counter must be initialized, which is done with the simple one-line version of the script command.

The `breakpoint` command add command in this example attaches a Python script to breakpoint 1. To remove the breakpoint command:

```
(lldb) breakpoint command delete 1
```

7.4 Create new command using a Python function

You can use Python functions to create new LLDB command interpreter commands, which will work like all the natively-defined LLDB commands. This provides a very flexible and easy way to extend LLDB to meet your debugging requirements.

To write a Python function that implements a new LLDB command, define the function to take four arguments as follows:

```
def command_function(debugger, command, result, internal_dict):
# Your code goes here
```

Optionally, you can also provide a Python `docstring`, and LLDB will use it when providing help for your command. For example:

```
def command_function(debugger, command, result, internal_dict):
"""This command takes a lot of options and does many fancy things"""
   # Your code goes here
```

Table 7-3 Python command function arguments

Argument	Type	Description
`debugger`	`lldb.SBDebugger`	The current debugger object.
`command`	`Python string`	A Python string containing all arguments for your command. To chop up the arguments, try using the shlex module's `shlex.split(command)` to properly extract the arguments.
`result`	`lldb.SBCommandReturnObject`	A return object that encapsulates success or failure information for the command and output text that is to be printed as a result of the command. The plain Python `print` command also works, but text will not go into the result by default (this command is useful as a temporary logging facility).
`internal_dict`	`Python dict object`	The dictionary for the current embedded script session that contains all variables and functions.

For convenience, you can treat the result object as a Python file object:

```
print >>result, "my command does lots of cool stuff"
```

`SBCommandReturnObject` and `SBStream` both support this file-like behavior by providing `write()` and `flush()` calls at the Python layer.

Another convenience when defining LLDB command-line commands is the command, `command script import`. It imports a module specified by the file path so you are not required to change your `PYTHONPATH` for temporary scripts. This command also offers another convenience whereby if your new script module has a function of the form:

```
def __lldb_init_module(debugger, internal_dict):
# Command Initialization code goes here
```

Where `debugger` and `internal_dict` are defined as shown above. In this case, the function is run when the module is loaded, allowing you to add any commands into the current debugger.

NOTE: This function only runs when you use the LLDB command, `command script import`. It does not run if anyone imports your module from another module.

To always run code when your module is loaded from LLDB, or when it is loaded with an import statement in Python code, you can test the `lldb.debugger` object because you imported the module at the top of the Python `ls.py` module. This test must be in code that is not contained inside any function or class, just like the standard test for `__main__` that all Python modules typically do. The sample code would look as follows:

```
if __name__ == '__main__':
    # Create a new debugger instance in your module if your module
    # can be run from the command line. When running a script from
    # the command line, there will not be any debugger object in
    # lldb.debugger, so you can just create it if needed
    lldb.debugger = lldb.SBDebugger.Create()
elif lldb.debugger:
    # Module is being run inside the LLDB interpreter
    lldb.debugger.HandleCommand('command script add -f ls.ls ls')
    print 'The "ls" python command has been installed and is ready for
use.'
```

Next, create a module named `ls.py` in the `~/ls.py` file, which implements a function that the LLDB's Python command code can use:

```
#!/usr/bin/python

import lldb
import commands
import optparse
import shlex

def ls(debugger, command, result, internal_dict):
    print >>result, (commands.getoutput('/bin/ls %s' % command))

# And the initialization code to add your commands
def __lldb_init_module(debugger, internal_dict):
    debugger.HandleCommand('command script add -f ls.ls ls')
    print 'The "ls" python command has been installed and is ready for
use.'
```

Finally, load the module into LLDB and use it:

```
% lldb
(lldb) command script import ~/ls.py
The "ls" python command has been installed and is ready for use.
(lldb) ls -l /tmp/
total 365848
-rw-r--r--@  1 someuser  wheel         6148 Jan 19 17:27 .DS_Store
-rw-------   1 someuser  wheel         7331 Jan 19 15:37 crash.log
```

NOTE: A more interesting template, `cmdtemplate.py`, is available in the source repository (http://llvm.org/svn/llvm-project/lldb/trunk/examples/python/cmdtemplate.py). It can help you create LLDB commands quickly.

A commonly-required facility is the ability to create a command that performs some token substitution and then runs a different debugger command (typically, it performs a `po` on the result of an expression evaluated on its argument). For example, given the following program:

```
#import <Foundation/Foundation.h>
NSString*
ModifyString(NSString* src)
{
    return [src stringByAppendingString:@"foobar"];
}

int main()
{
    NSString* aString = @"Hello world";
    NSString* anotherString = @"Let's be friends";
    return 1;
}
```

You might want a `pofoo X` command that equates `po [ModifyString(X) capitalizedString]`. The following debugger interaction shows how to achieve that goal:

```
(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()' or
Ctrl-D.
>>> def pofoo_funct(debugger, command, result, internal_dict):
...   cmd = "po [ModifyString(" + command + ") capitalizedString]"
...   lldb.debugger.HandleCommand(cmd)
...
>>> ^D
(lldb) command script add pofoo -f pofoo_funct
(lldb) pofoo aString
$1 = 0x000000010010aa00 Hello Worldfoobar
(lldb) pofoo anotherString
$2 = 0x000000010010aba0 Let's Be Friendsfoobar
```

8 Script example

You can script LLDB in two ways:

- A Unix Python session can initiate and run a debug session non-interactively using LLDB.
- Within the LLDB debugger tool, you can use Python scripts to perform many tasks, including inspecting program data, iterating over containers, and determining if a breakpoint should stop execution or continue.

The following sections show how to use Python scripting to find a bug in a program that searches for text in a large binary tree.

8.1 The test program and input

The test program is a simple C program (`dictionary.c`) that reads a text file and stores all the words from the file in a binary search tree, sorted alphabetically. It then enters a loop prompting yo for a word, searching for the word in the tree (using a binary search), and reporting whether it found the word in the tree.

The input text file used to test the program contains the text for William Shakespeare's famous tragedy, *Romeo and Juliet*.

8.2 The bug

When you try running the program, you will find there is a problem. While it successfully finds some of the words it is expected to find, such as *love* or *sun*, it fails to find the word *Romeo*, which must be in the input text file:

```
% ./dictionary Romeo-and-Juliet.txt
Dictionary loaded.
Enter search word: love
Yes!
Enter search word: sun
Yes!
Enter search word: Romeo
No!
Enter search word: ^D
%
```

8.3 Check for the word in the tree: use DFS

The first task is to determine whether the word, *Romeo*, was inserted into the tree. Because *Romeo and Juliet* has thousands of words, trying to examine the binary search tree by hand is completely impractical. Therefore, a Python script is written to search the tree.

The script is written as a recursive depth-first search (DFS) function that traverses the entire tree, searching for a word while maintaining information about the path from the root of the tree to the current node. If the function finds the word in the tree, it returns the path from the root to the node containing the word.

Following is the DFS function in Python, with line numbers added for easy reference in the subsequent explanations of the code:

```
 1: def DFS (root, word, cur_path):
 2:     root_word_ptr = root.GetChildMemberWithName ("word")
 3:     left_child_ptr = root.GetChildMemberWithName ("left")
 4:     right_child_ptr = root.GetChildMemberWithName ("right")
 5:     root_word = root_word_ptr.GetSummary()
 6:     end = len (root_word) - 1
 7:     if root_word[0] == '"' and root_word[end] == '"':
 8:         root_word = root_word[1:end]
 9:     end = len (root_word) - 1
10:     if root_word[0] == '\'' and root_word[end] == '\'':
11:        root_word = root_word[1:end]
12:     if root_word == word:
13:         return cur_path
14:     elif word < root_word:
15:         if left_child_ptr.GetValue() == None:
16:             return ""
17:         else:
18:             cur_path = cur_path + "L"
19:             return DFS (left_child_ptr, word, cur_path)
20:     else:
21:         if right_child_ptr.GetValue() == None:
22:             return ""
23:         else:
24:             cur_path = cur_path + "R"
25:             return DFS (right_child_ptr, word, cur_path)
```

8.4 Work with program variables in Python

Before calling a Python function on any of the program's variables, put the variable into a form that Python can access. To do this, examine the parameters for the DFS function:

- The first parameter is a node in a binary search tree, put into a Python variable.
- The second parameter is the word being searched for (a string). The third parameter is a string representing the path from the root of the tree to the current node.

The first parameter is the most interesting: it is the Python variable that must contain a node in the search tree. How can a variable be taken out of your program and stored in a Python variable? What kind of Python variable will it be?

The answers are to use the LLDB API functions, which are provided as part of the LLDB Python module. When running Python from inside LLDB, LLDB automatically provides the current frame object as a Python variable, `lldb.frame`. This variable is of type `SBFrame` (see the LLDB API for more information on SBFrame objects).

A frame object can be asked to find and return its local variable. The `FindVariable()` function is called on the `lldb.frame` object to provide the dictionary variable as a Python variable:

```
root = lldb.frame.FindVariable ("dictionary")
```

This line is executed in the Python script interpreter in LLDB, and it requests the current frame to find the variable named dictionary and return it. The returned value is stored in the Python variable named `root`. This answers the question of how to get the variable, but it still does not explain what is put into `root`.

In the LLDB API, observe that the `SBFrame` method `FindVariable()` returns an object of type `SBValue`. `SBValue` objects are used, among other things, to wrap up program variables and values. There are many useful methods defined in the `SBValue` class that allow you to get information or children values out of `SBValues`. The `SBValue` methods used in the DFS function are `GetChildMemberWithName()`, `GetSummary()`, and `GetValue()`.

NOTE: For more information on `SBValues`, see the `SBValue.h` header file.

8.5 Explanation of the DFS script

Before describing the example code in detail, following is a high-level overview of what the code does.

The nodes in the binary search tree are of type `tree_node *`, which is defined as follows:

```
typedef struct tree_node
{
  const char *word;
  struct tree_node *left;
  struct tree_node *right;
} tree_node;
```

- Lines 2 to 11 of the DFS function involve getting data out of the current tree node and getting ready to do the actual search.
  - Lines 2 to 4 of the function get the word, left, and right fields out of the current node and store them in Python variables.
  - Because `root_word_ptr` is a pointer to the word, and the word itself is required, line 5 calls `GetSummary()` to get a string containing the value out of the pointer.
  - Because `GetSummary()` adds quotes around its result, lines 6 to 11 strip the surrounding quotes off the word.
- Lines 12 to 25 are the actual depth-first search.
  - Line 12 checks to see if the word in the current node is the one being searched for. If so, the search is completed.
  - Line 13 returns the current path.
  - Otherwise, line 14 checks to see if the search should go left (that is, the search word comes before the current word).
  - If the search goes left, line 15 checks to see if the left pointer child is NULL (None is the Python equivalent of NULL).
  - If the left pointer is NULL, then the word is not in this tree and the function returns an empty path (line 16).
  - Otherwise, it adds an `L` to the end of the current path string to indicate that the search is going left (line 18), and then recursively searches on the left child (line 19).
  - Lines 20-25 are the same as lines 14-19, except that the search goes right instead of left.

It can be difficult to type all the DFS function code directly into the interpreter, because if you make a single typing mistake, you must completely start over. Therefore, we recommend you write longer, more complicated script functions in a separate file (in this case, `tree_utils.py`), and then import the file into the LLDB Python interpreter.

8.6 Use the DFS script

Start by running the dictionary program in the Hexagon simulator, and assign it a unique port number:

```
hexagon-sim -G 54321  dictionary -- Romeo_and_Juliet.txt
hexagon-sim INFO: The rev_id used in the simulation is 0x00008d68
(v68n_1024)
hexagon-sim INFO: Setting up debug server on port 54321
```

Port 54321 is used in this example, but you can use any available, valid port number.

Next, launch LLDB in a separate terminal, and use `gdb-remote` to attach to the dictionary program running on the simulator:

```
>hexagon-lldb dictionary

Hexagon utilities (pagetable, tlb, pv) loaded
(lldb) target create "dictionary"
Current executable set to '/local/mnt/workspace/dictionary' (hexagon).
(lldb) gdb-remote 54321
Process 1 stopped
* thread #1, name = 'T1', stop reason = signal SIGTRAP
    frame #0: 0x00000000 dictionary`_start
dictionary`_start:
->  0x0 <+0>:    {     jump 0x98 }
    0x4 <+4>:    {     jump 0x80 }
    0x8 <+8>:    {     jump 0x8c }
    0xc:      <unknown>
(lldb) breakpoint set -n find_word
Breakpoint 1: where = dictionary`find_word + 12 at dictionary.c:108:6,
address = 0x0000644c
(lldb) c
Process 1 resuming
```

Back in the simulator's terminal, wait for the message, `Dictionary loaded. Enter search word:`, to appear, and then enter `Romeo<CR>`. In the LLDB terminal, you will see the breakpoint at `find_word` and the program is stopped.

At this point, the DFS function is ready to determine whether the word, *Romeo*, is stored in the tree. To use the function in LLDB on the dictionary program, do something like the following:

```
Process 1 stopped
* thread #1, name = 'T1', stop reason = breakpoint 1.1
    frame #0: 0x0000644c dictionary`find_word(dictionary=0x000141c0,
word="romeo") at dictionary.c:108:6
   105         in the binary search tree.  */
   106
   107      int find_word(tree_node *dictionary, char *word) {
-> 108      if (!word || !dictionary)
   109         return 0;
   110
   111 int compare_value = strcmp(word, dictionary->word);
(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()' or
Ctrl-D.
>>> import tree_utils
>>> root = lldb.frame.FindVariable ("dictionary")
```

```
>>> current_path = ""
>>> path = tree_utils.DFS (root, "Romeo", current_path)
>>> print path
LLRRL
>>> exit
(lldb)
```

The first bit of code shows starting LLDB, attaching to the dictionary program, and getting to the `find_word` function in the LLDB. In this example, the interesting part begins when the script command drops the program into the embedded interactive Python interpreter. This code is described line-by-line.

- The first line, `import tree_utils`:

  Imports the file containing the DFS function (`tree_utils.py`) into Python.

  When importing the file, the `.py` extension is omitted. You can call any function in the imported file after assigning it the prefix `tree_utils` so Python knows where to look for the function.

- The line, `root = lldb.frame.FindVariable ("dictionary")`:

  Gets the program variable dictionary, which contains the binary search tree and stores it in the Python variable root. (For details on how this works, see Section 8.4.)

- The line, `current_path = ""`:

  Initializes `current_path` from the root of the tree to the current node. Because the search starts at the root of the tree, the current path starts as an empty string. As searching goes right and left through the tree, the DFS function will append an `R` or an `L` to the current path, as appropriate.

- The line, `path = tree_utils.DFS (root, "Romeo", current_path)`:

  Calls the DFS function (prefixing it with the module name so Python can find it). The function is passed three parameters: the binary tree stored in the variable root, the word being searched for, and the current path. The path that the DFS function returns is assigned to the Python variable path.

- The last line, `print path`:

  Indicates whether the word was found in the tree, and if found, the path from the tree root to the word.

In this code example, the word, `Romeo`, was found in the tree; the search path through the tree is left-left-right-right-left.

8.7 Use breakpoint command scripts

At this point, you are halfway through the scripting example. The word is confirmed to be in the binary tree, and its location in the tree is known. The next task is to determine why the binary search algorithm is not finding the word. Do this with breakpoint command scripts.

The binary search algorithm has two main decision points: the decision to follow the right branch, and the decision to follow the left branch. Set a breakpoint at each of these decision points, and attach a Python breakpoint command script to each breakpoint. The breakpoint commands use the global path Python variable that you got from the DFS function.

Each time one of these decision breakpoints is hit, the script compares the actual decision with the decision the front of the path variable indicates should be made (that is, the first character in the path). If the actual decision and the path agree, the front character is stripped off the path, and execution is resumed. In this case, you never see the breakpoint being hit.

But if the decision differs from what the path says it should be, the script prints a message and does not resume execution, leaving you at the first point in the program where a wrong decision is being made.

8.8 Python breakpoint command scripts

Python breakpoint command scripts are not what they seem. When you enter a Python breakpoint command in LLDB, it appears that you are entering one or more plain lines of Python. But LLDB then takes what you entered and wraps it into a Python function (just like using the def Python command). It automatically gives the function an obscure, unique, hard-to-stumble-across function name, and gives it two parameters: `frame` and `bp_loc`.

When the breakpoint is hit, LLDB wraps up the frame object where the breakpoint was hit, and the breakpoint location object for the breakpoint that was hit, and puts them into Python variables for you. It then calls the Python function that was created for the breakpoint command, and passes in the frame and breakpoint location objects.

What does this mean for you when writing the Python breakpoint commands? It means that there are two things to keep in mind:

- To access any Python variables created outside your script, you must declare such variables to be global. If you do not declare them as global, then the Python function will treat them as local variables, and you will get unexpected behavior.
- All Python breakpoint command scripts automatically have a `frame` and `bp_loc` variable. LLDB preloads the variables with the correct context for the breakpoint. You are not required to use these variables, but they are available.

8.9 Decision point breakpoint commands

Following is what the Python breakpoint command script looks like for the decision to go right:

```
global path
if path[0] == 'R':
    path = path[1:]
    thread = frame.GetThread()
    process = thread.GetProcess()
    process.Continue()
else:
    print "Here is the problem; going right, should go left!"
```

As a reminder, LLDB takes this script and wraps it in a function, like this example:

```
def some_unique_and_obscure_function_name (frame, bp_loc):
    global path
    if path[0] == 'R':
        path = path[1:]
        thread = frame.GetThread()
        process = thread.GetProcess()
        process.Continue()
    else:
        print "Here is the problem; going right, should go left!"
```

LLDB calls the function, passing in the correct frame and breakpoint location whenever the breakpoint is hit.

Notes about this function:

- First, it is accessing and updating a piece of state (the `path` variable), and conditioning its behavior based upon this variable. Because the variable is defined outside of the script (and therefore outside of the corresponding function), Python must know that it is accessing a global variable. That is what the first line of the script does.
- Next, the function checks where the path says the search should go, and compares it to the actual decision (recall that the program is at the breakpoint for the decision to go right). If the path agrees with the decision, the first character is stripped off of the path.
- Because the decision matched the path, the program should resume execution by making use of the frame parameter that LLDB guarantees will be available. The LLDB API functions are used to get the current thread from the current frame, and then to get the process from the thread. Once the process is gotten, use it to resume the execution (using the `Continue()` function).
- If the decision to go right does not agree with the path, execution is not resumed. The breakpoint is allowed to remain stopped (by doing nothing), and a message is printed noting that the problem has been found and explaining what the problem is.

8.10 Use breakpoint commands

Here is what happens when using the breakpoint commands in the program. The command source list `-n find_word` displays the function containing the two decision points. In the code below, the breakpoints should be set on lines 113 and 115:

```
(lldb) source list -n find_word
File: /Volumes/Data/HD2/carolinetice/Desktop/LLDB-Web-
Examples/dictionary.c.
101
102 int
103 find_word (tree_node *dictionary, char *word)
104 {
105   if (!word || !dictionary)
106     return 0;
107
108   int compare_value = strcmp (word, dictionary->word);
109
110   if (compare_value == 0)
111     return 1;
112   else if (compare_value < 0)
113     return find_word (dictionary->left, word);
114   else
115     return find_word (dictionary->right, word);
116 }
117
```

Set the breakpoints, enter the breakpoint command scripts, then see what happens:

```
(lldb) breakpoint set -l 113
Breakpoint created: 2: file ='dictionary.c', line = 113, locations = 1,
resolved = 1
(lldb) breakpoint set -l 115
Breakpoint created: 3: file ='dictionary.c', line = 115, locations = 1,
resolved = 1
(lldb) breakpoint command add -s python 2
Enter your Python command(s). Type 'DONE' to end.
> global path
> if (path[0] == 'L'):
>     path = path[1:]
>     thread = frame.GetThread()
>     process = thread.GetProcess()
>     process.Continue()
> else:
>     print "Here is the problem. Going left, should go right!"
> DONE
(lldb) breakpoint command add -s python 3
Enter your Python command(s). Type 'DONE' to end.
> global path
> if (path[0] == 'R'):
>     path = path[1:]
>     thread = frame.GetThread()
>     process = thread.GetProcess()
>     process.Continue()
> else:
>     print "Here is the problem. Going right, should go left!"
> DONE
```

```
(lldb) continue
Process 696 resuming
Here is the problem. Going right, should go left!
Process 696 stopped
* thread #1: tid = 0x2d03, 0x000000010000189f dictionary`find_word + 127
at dictionary.c:115, stop reason = breakpoint 3.1
  frame #0: 0x000000010000189f dictionary`find_word + 127 at
dictionary.c:115
    112   else if (compare_value < 0)
    113     return find_word (dictionary->left, word);
    114   else
 -> 115     return find_word (dictionary->right, word);
    116 }
    117
    118 void
(lldb)
```

After setting the breakpoints, adding the breakpoint commands, and continuing, the program runs and then hits one of the breakpoints, printing out the error message from the breakpoint command. At this point in the tree, the search algorithm decided to go right, but the path says the node should go to the left. Examining the word at the node where the program stopped, and the search word, reveals the following:

```
(lldb) expr dictionary->word
(const char *) $1 = 0x0000000100100080 "dramatis"
(lldb) expr word
(char *) $2 = 0x00007fff5fbff108 "romeo"
```

In this example, the word at the current node is `dramatis`, while the word being searched for is `romeo`, which comes after `dramatis`, alphabetically. Thus, going right seems to be the correct decision. Use Python to see what it thinks the path from the current node to the word is:

```
(lldb) script print path LLRRL
```

According to Python the search needed to go left-left-right-right-left from the current node to find the word being searched for. Double-check the tree to see what word it has at that node:

```
(lldb) expr dictionary->left->left->right->right->left->word (const char
*) $4 = 0x0000000100100880 "Romeo"
```

In this example, the word being searched for is `romeo`, while the word at the DFS location is `Romeo`. Here is the source of the problem: one word is uppercase, while the other word is lowercase. There appears to be a case conversion problem somewhere in the program.

This is the end of the example showing how you can use Python scripting in LLDB to help find bugs in your program.

8.11 Source files for the example

The Python script used in this example is written for Python 2.7.

The code used in this example—including the DFS function, Dictionary program (with case-conversion bug), and other Python script examples (`tree_utils.py`)—is available in the following files from https://lldb.llvm.org/use/python.html#source-files-for-the-example:

- `tree_utils.py`

  Example Python functions using LLDB API (including DFS)

- `dictionary.c`

  Sample dictionary program, with a bug

You can get the text for *Romeo and Juliet* from the Gutenberg Project (http://www.gutenberg.org).

9 Debug with the booter image

To use LLDB with a booter image, such as bootimg.pbn or runelf.pbn, define the LLDB_HEXAGON_BOOTER_PATH environment variable with the complete path to the booter executable.

Load the application to debug in LLDB. When you run the program, LLDB launches hexagon-sim with the correct flags to run your application under the booter.

For details on how to debug the application, see the booter image documentation.

10 Troubleshooting

This chapter describes potential problems with using LLDB, and how to resolve them.

10.1 File and line breakpoints not getting hit

If your file and line breakpoints are not being hit, first verify that your source files were compiled with debug information. Typically, this means passing `-g` to the compiler when compiling your source file.

When setting breakpoints in implementation source files (.c, .cpp, .cxx, .m, .mm, and so on), by default, LLDB searches only for compile units whose file names match. If your code does tricky things such as using `#include` to include source files, breakpoints in `bar.c` file are inlined into the compile unit for the `foo.c` file:

```
% cat foo.c
#include "bar.c"
#include "baz.c"
...
```

If your code does this, or if your build system combines multiple files such that breakpoints from one implementation file are compiled into another implementation file, you must instruct LLDB to always search for inlined breakpoint locations by adding the following line to your `~/.lldbinit` file:

```
% *echo "settings set target.inline-breakpoint-strategy always"
>> ~/.lldbinit*
```

This line directs LLDB to always look in all compile units and search for breakpoint locations by file and line, even if the implementation file does not match. Setting breakpoints in header files always searches all compile units, because inline functions are commonly defined in header files, and they often cause multiple breakpoints to have source line information that matches many header file paths.

If you set a file and line breakpoint using a full path to the source file, this path must match the full paths in the debug information. If the paths do not match (possibly due to passing in a resolved source file path that does not match an unresolved path in the debug information), this can cause breakpoints to not be resolved. Try setting breakpoints using the file base name only.

If you are using an IDE and move your project in your file system and then build again, sometimes doing a clean-then-build will solve the issue. This will fix the issue if some `.o` files did not get rebuilt after the move, as the `.o` files in the build folder might still contain stale debug information with the old source locations.

LLDB supports debugging multiple ELF files, but only as separate targets. It supports debugging a single file that is postprocessed to contain multiple ELF files (a stitched ELF file) if the symbol tables are combined into one table. If a postprocessed file contains more than one symbol table, LLDB does not supported debugging this file. The ELF standard states:

> "SHT_SYMTAB and SHT_DYNSYM. These sections hold a symbol table. Currently, an object file may have only one section of each type, ..."
> (https://refspecs.linuxfoundation.org/elf/gabi41.pdf)

You cannot load a stitched file without a symbol table and try to use more than one ELF file as multiple symbol tables. You also cannot load a file and try to use a second file as an additional symbol table. Only one symbol table per executable is allowed, except for shared libraries loaded by the loader in the standard way.

10.2 Check for debug symbols

To determine if a module contains debug information, check whether that module has any compile units (source files). For example:

```
(lldb) file /tmp/a.out
(lldb) image list
[  0] 71E5A649-8FEF-3887-9CED-D3EF8FC2FD6E 0x0000000100000000 /tmp/a.out
      /tmp/a.out.dSYM/Contents/Resources/DWARF/a.out
[  1] 6900F2BA-DB48-3B78-B668-58FC0CF6BCB8 0x00007fff5fc00000
/usr/lib/dyld
...
(lldb) script lldb.target.module['/tmp/a.out'].GetNumCompileUnits()
1
(lldb) script lldb.target.module['/usr/lib/dyld'].GetNumCompileUnits()
0
```

In this example, the `/tmp/a.out` file includes a compile unit, while the `/usr/lib/dyld` file does not.

11 Architecture

For information on the LLDB architecture, see the LLDB website (http://lldb.llvm.org/architecture/index.html).

A Acknowledgments

This document was derived from the LLDB Project documentation under the terms of the LLDB Release License. Following are corresponding license statements.

- LLDB Release License
- Copyrights and Licenses for Third Party Software Distributed with LLDB
- Block Implementation Specification

A.1 LLDB Release License

Portions of this LLDB Debugger are subject to the below list of conditions and disclaimers:

University of Illinois/NCSA
Open Source License

Developed by:

LLDB Team

http://lldb.llvm.org/

A.2 Copyrights and Licenses for Third Party Software Distributed with LLVM

The LLVM software contains code written by third parties. Such software will have its own individual LICENSE.TXT file in the directory in which it appears. This file will describe the copyrights, license, and restrictions which apply to that code.

The disclaimer of warranty in the University of Illinois Open Source License applies to all code in the LLVM Distribution, and nothing in any of the other licenses gives permission to use the names of the LLVM Team or the University of Illinois to endorse or promote products derived from this Software.

The following pieces of software have additional or alternate copyrights, licenses, and/or restrictions:

Program	Directory
Autoconf	`llvm/autoconf`
	`llvm/projects/ModuleMaker/autoconf`
	`llvm/projects/sample/autoconf`
CellSPU backend	`llvm/lib/Target/CellSPU/README.txt`
Google Test	`llvm/utils/unittest/googletest`
OpenBSD regex	`llvm/lib/Support/{reg*, COPYRIGHT.regex}`
pyyaml tests	`llvm/test/YAMLParser/{*.data, LICENSE.TXT}`

A.3 Block Implementation Specification

Block Implementation Specification