

# LLDB Quick Start Guide

# Contents

## About LLDB and Xcode 4

### At a Glance 4

Understand LLDB Basics to Unlock Advanced Features 4

Use LLDB Equivalents for Common GDB Commands 5

Standalone LLDB Workflow 5

See Also 5

## Getting Started with LLDB 6

### LLDB Command Structure 6

Understanding Command Syntax 6

Using Command Options 7

Using Raw Commands 8

Using Command Completion in LLDB 8

Comparing LLDB with GDB 9

Scripting with Python 10

### Command Aliases and Help 11

Understanding Command Aliases 11

Using LLDB Help 12

## GDB and LLDB Command Examples 16

Execution Commands 16

Breakpoint Commands 18

Watchpoint Commands 20

Examining Variables 21

Evaluating Expressions 23

Examining Thread State 24

Executable and Shared Library Query Commands 28

Miscellaneous 30

## Using LLDB as a Standalone Debugger 31

Specifying the Program to Debug 31

Setting Breakpoints 32

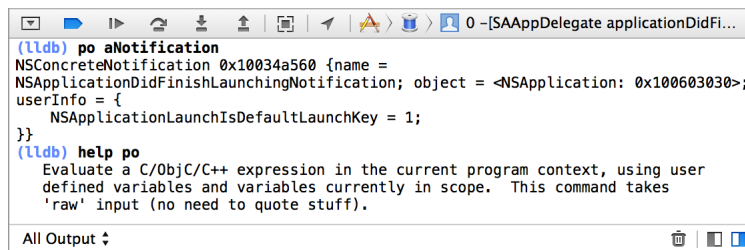
Setting Watchpoints 34

Launching the Program with LLDB 35

Controlling Your Program	36
Examining Thread State	38
Examining the Stack Frame State	39
Executing Alternative Code	41
<b>Document Revision History</b>	<b>42</b>

# About LLDB and Xcode

With the release of Xcode 5, the LLDB debugger becomes the foundation for the debugging experience on OS X.



```
(lldb) po aNotification
NSConcreteNotification 0x10034a560 {name =
NSApplicationDidFinishLaunchingNotification; object = <NSApplication: 0x100603030>;
userInfo = {
    NSApplicationLaunchIsDefaultLaunchKey = 1;
}}
(lldb) help po
Evaluate a C/Objective-C/C++ expression in the current program context, using user
defined variables and variables currently in scope. This command takes
'raw' input (no need to quote stuff).
```

LLDB is Apple’s “from the ground up” replacement for GDB, developed in close coordination with the LLVM compilers to bring you state-of-the-art debugging with extensive capabilities in flow control and data inspection. Starting with Xcode 5, all new and preexisting development projects are automatically reconfigured to use LLDB.

The standard LLDB installation provides you with an extensive set of commands designed to be compatible with familiar GDB commands. In addition to using the standard configuration, you can easily customize LLDB to suit your needs.

## At a Glance

LLDB is fully integrated with Xcode 5 for source development and the build-and-run debugging experience. You access its wealth of capabilities using the controls provided by the Xcode UI and with commands issued from the Xcode debugging console.

## Understand LLDB Basics to Unlock Advanced Features

With the LLDB command language, you can use LLDB’s advanced features. The command syntax is regular and easy to learn. Many commands are expressed by included shortcuts, saving you time and keystrokes. And you can use the LLDB help system to quickly inspect and learn the details of the existing commands, shortcuts, and command options.

You customize LLDB using the command alias capability. You can also extend LLDB through the use of Python scripts and the Python-LLDB Object Library.

---

**Relevant chapter:** [Getting Started with LLDB](#) (page 6)

---

## Use LLDB Equivalents for Common GDB Commands

LLDB, as delivered, includes many command aliases designed to be the same as GDB commands. If you are already experienced in using GDB commands, you can use the provided tables to look up GDB commands and find the LLDB equivalents, including canonical and shorthand forms.

---

**Relevant chapter:** [GDB and LLDB Command Examples](#) (page 16)

---

## Standalone LLDB Workflow

You usually experience LLDB by using the Xcode debugging features and, and you issue LLDB commands using the Xcode console pane. However, for development of open source and other non-GUI-based application debugging, you can use LLDB from a Terminal window as a traditional command-line debugger.

Knowing how LLDB works as a command-line debugger can help you understand and use the full power of LLDB in the Xcode console pane as well.

---

**Relevant Chapter:** [Using LLDB as a Standalone Debugger](#) (page 31)

---

## See Also

For a good look at how to use Xcode's debugging features, all powered by the LLDB debugging engine, see the WWDC 2013 session video for Tools #407 [WWDC 2013: Debugging with Xcode](#).

To see the latest advanced techniques to help you efficiently track down bugs with LLDB, view the WWDC 2013 session video for Tools #413 [WWDC 2013: Advanced Debugging with LLDB](#).

For more information about the use of LLDB Python scripting and other advanced capabilities, visit [The LLDB Debugger](#).

# Getting Started with LLDB

LLDB is a command-line debugging environment with functionality similar to GDB. LLDB provides the underlying debugging environment for Xcode, which includes a console pane in the debugging area for direct access to LLDB commands within the Xcode IDE environment.

This chapter briefly explains the LLDB syntax and command features, informs you on the use of the command aliasing capabilities, and introduces you to the LLDB help system.

## LLDB Command Structure

All users starting out with LLDB should be aware of the LLDB command structure and syntax in order to tap the potential of LLDB and understand how to get the most from it. In many cases, commands provided by LLDB—for instance, `list` and `b`—work just like GDB commands and make learning LLDB easier for experienced GDB users.

## Understanding Command Syntax

The LLDB command syntax is structured and regular throughout. LLDB commands are all of the form:

```
<command> [<subcommand> [<subcommand>...]] <action> [-options [option-value]]  
[argument [argument...]]
```

*Command* and *subcommand* are names of LLDB debugger objects. Commands and subcommands are arranged in a hierarchy: a particular command object creates the context for a subcommand object which follows it, which again provides context for the next subcommand, and so on. The *action* is the operation you want to perform in the context of the combined debugger objects (the string of *command subcommand..* entities preceding it). Options are *action modifiers*, which often take a value. Arguments represent a variety of different things according to the context of the command being used—for example, using the `process launch` command to start a process, the arguments in that context are what you enter on the command line as if you were invoking the process outside of a debugging session.

The LLDB command-line parsing is completed before command execution. Commands, subcommands, options, option values, and arguments are all white space separated, and double quotes are used to protect white spaces in option values and arguments. If you need to put a backslash or double quote character into an argument, you precede that character with a backslash in the argument. LLDB uses both single and double quotes equivalently, allowing doubly quoted portions of a command line to be written easily. For example:

```
(lldb) command [subcommand] -option "some \"quoted\" string"
```

can also be written:

```
(lldb) command [subcommand] -option 'some "quoted" string'
```

This command parsing design helps make LLDB command syntax regular and uniform across all commands. (For the GDB users, it also means you may have to quote some arguments in LLDB that you wouldn't have to quote in GDB.)

As example of a simple LLDB command, to set a breakpoint in file `test.c` at line 12 you enter:

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

Commands which appear to diverge from this model—for example, `print` or `b`—are typically customized command forms created by the `command alias` mechanism, which is discussed in [Command Aliases and Help](#) (page 11).

## Using Command Options

Command options in LLDB have a canonical (also referred to as “discoverable”) form and an abbreviated form. For example, here is a partial listing of the command options for the `breakpoint set` command, listing the canonical form in parentheses:

```
breakpoint set
    -M <method> ( --method <method> )
    -S <selector> ( --selector <selector> )
    -b <function-name> ( --basename <function-name> )
    -f <filename> ( --file <filename> )
    -l <linenum> ( --line <linenum> )
    -n <function-name> ( --name <function-name> )
...

```

Options can be placed in any order on the command line following the command. If arguments begin with a hyphen (–), you indicate to LLDB that you’re done with options for the current command by adding the option termination signal, a double hyphen (—). For instance, if you want to launch a process and give the process launch command the `--stop-at-entry` option, and want that same process to be launched with the arguments `--program_arg_1 value` and `--program_arg_2 value`, you enter:

```
(lldb) process launch --stop-at-entry -- --program_arg_1 value --program_arg_2 value
```

## Using Raw Commands

The LLDB command parser supports “raw” commands, in which, after command options are removed, the rest of the command string is passed uninterpreted to the command. This is convenient for commands whose arguments might be some complex expression that would be clumsy to protect with backslashes. For instance, the `expression` command is a raw command.

When you look up a command with `help`, the output for a command tells you whether the command is raw or not so that you know what to expect.

Raw commands can still have options, if your command string has dashes in it, you indicate that these are not option markers by putting an option termination (—) after the command name but before the command string.

## Using Command Completion in LLDB

LLDB supports command completion for source file names, symbol names, file names, and so forth. Completion in the Terminal window is initiated by inputting a tab character on the command line. Completions in the Xcode console work in the same way as the completions in the source code editor: The completions pop up automatically after the third character is typed, and the completions pop-up can be manually summoned via the Esc (Escape) key. In addition, completions in the Xcode console follow the Xcode Text Editing preferences specified in the Editing panel.

Individual options in a command can have different completers. For example, the `--file <path>` option in `breakpoint` completes to source files, the `--shlib <path>` option completes to currently loaded shared libraries, and so forth. The behavior can be quite specific—for example: if you specify `--shlib <path>`, and are completing on `--file <path>`, LLDB lists only source files in the shared library specified by `--shlib <path>`.



## Comparing LLDB with GDB

The command-line parsing and uniformity of the LLDB command parser differ when using LLDB compared with GDB. The LLDB command syntax sometimes forces you to be more explicit about stating your intentions, but it is more regular in use.

For example, setting a breakpoint is a common operation. In GDB, to set a breakpoint you might enter the following to break at line 12 of `foo.c`:

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

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

```
(gdb) break foo
```

More complex `break` expressions are possible in GDB. One example is `(gdb) break foo.c:foo`, which means “set the breakpoint in the function `foo` in the file `foo.c`.” But at some point the GDB syntax becomes convoluted and limits GDB functionality, especially in C++, where there may be no reliable way to specify the function you want to break on. These deficiencies happen because GDB command-line syntax is supported by a complex of special expression parsers that can be at odds with each other.

The LLDB breakpoint command, by comparison, requires only a simple, straightforward approach in its expression and provides the advantages of intelligent auto completion and the ability to set breakpoints in more complex situations. To set the same file and line breakpoint in LLDB, enter:

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

To set a breakpoint on a function named `foo` in LLDB, enter:

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

Setting breakpoints by name is even more powerful in LLDB than in GDB because you can specify that you want to set a breakpoint at a function by method name. To set a breakpoint on all C++ methods named `foo`, enter:

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

To set a breakpoint on Objective-C selectors named `alignLeftEdges:`, enter:

```
(lldb) breakpoint set --selector alignLeftEdges:
```

You can limit any breakpoints to a specific executable image by using the `--shlib <path>` expression.

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

The LLDB commands presented in this section, using the discoverable command name and canonical form of the option, may seem somewhat lengthy. However, just as in GDB, the LLDB command interpreter does a shortest-unique-string match on command names, creating an abbreviated command form. For example, the following two command-line expressions demonstrate the same command:

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

Similarly, you can combine both shortest-unique-string matching with the abbreviated option format to reduce keystrokes. Using the two together can reduce command line expressions further. For example:

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

becomes:

```
(lldb) br s -s foo.dylib -n foo
```

Using these features of LLDB offers much the same ‘shorthand’ feel and brevity as when using GDB. Look at [Breakpoint Commands](#) (page 18) and other sections in [GDB and LLDB Command Examples](#) (page 16) for more examples of how the shortest-unique-string matching and using abbreviated form options can save keystrokes.

---

**Note:** You can also use command `alias`, discussed in [Command Aliases and Help](#) (page 11), to make commonly used command-line expressions easy to remember, as well as able to be entered with only a few keystrokes.

---

## Scripting with Python

For advanced users, LLDB has a built-in Python interpreter accessible by using the `script` command. All the features of the debugger are available as classes in the Python interpreter. As a result, the more complex commands that in GDB are introduced with the `define` command can be achieved by writing Python functions

using the LLDB-Python library and then loading the scripts into your running session, accessing them with the `script` command. For more information about the LLDB-Python library, visit the [LLDB Python Reference](#) and [LLDB Python Scripting Example](#) sections of [The LLDB Debugger website](#).

## Command Aliases and Help

Now that you understand the LLDB syntax and command-line dynamics, turn your attention to two very useful features of LLDB—command aliases and the help system.

### Understanding Command Aliases

Use the command alias mechanism in LLDB to construct aliases for commonly used commands. For instance, if you repeatedly type:

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

you can construct an alias with:

```
(lldb) command alias bfl breakpoint set -f %1 -l %2
```

which lets you enter this command as:

```
(lldb) bfl foo.c 12
```

Because command aliases are useful in a wide variety of situations, you should become familiar with their construction. For a complete explanation of command alias construction, limitations, and syntax, use the LLDB help system. Type:

```
(lldb) help command alias
```

**Note:** In its default configuration, a few aliases for commonly used commands have been added to LLDB (for instance, `step`, `next`, and `continue`), but no attempt has been made to build an exhaustive set of aliases. In the experience of the LLDB development team, it is more convenient to make the basic commands unique down to a letter or two and then learn these sequences rather than to fill the namespace with lots of aliases and then have to type them out.

Users, however, are free to customize the LLDB command set in any way they like. LLDB reads the file `~/ .lldbinit` at startup. This is the file that stores the aliases created with the command `alias` when they are defined. The LLDB help system reads this initialization file and presents the aliases so you can easily remind yourself of what you've set up. To see all currently defined aliases and their definitions, use the `help -a` command and find the currently defined aliases at the end of the help output, starting with:

```
...
The following is a list of your current command abbreviations (see 'help command
alias' for more info): ...
```

---

## Using LLDB Help

Explore the LLDB help system to gain a broader understanding of what LLDB has to offer and to view details of LLDB command constructions. Becoming familiar with the `help` command gives you access to the extensive command documentation in the help system.

A simple invocation of the `help` command returns a list of all top-level LLDB commands. For example, here is a partial listing:

```
(lldb) help
The following is a list of built-in, permanent debugger commands:

_regexp-attach    -- Attach to a process id if in decimal, otherwise treat the
                   argument as a process name to attach to.
_regexp-break     -- Set a breakpoint using a regular expression to specify the
                   location, where <linenum> is in decimal and <address> is
                   in hex.
_regexp-bt        -- Show a backtrace. An optional argument is accepted; if
                   that argument is a number, it specifies the number of
                   frames to display. If that argument is 'all', full
```

```
backtraces of all threads are displayed.  
... and so forth ...
```

Invoking `help` followed by any of the commands lists the help entry for that command and any subcommands, options, and arguments. By invoking `help <command> <subcommand> [<subcommand>...]` iteratively over all the commands, you traverse the entire LLDB command hierarchy.

The `help` command display includes all current command aliases when invoked with the option `--show-aliases` (`-a` in short form).

```
(lldb) help -a
```

---

**Note:** The listing produced by `help -a` includes all the supplied command aliases and GDB command emulations as well as any command aliases you define.

---

**Important:** One alias included by popular demand is a weak emulation of the GDB `break` command. It doesn't do everything that the GDB `break` command does (for instance, it doesn't handle an expression like `break foo.c:bar`). Instead, it handles more common situations and makes the transition to using LLDB easier for experienced GDB users. The GDB `break` emulation is also aliased to `b` by default, matching the GDB convention.

To learn and exercise the LLDB command set natively, these supplied defaults put the GDB `break` emulation in the way of the rest of the LLDB breakpoint commands. In the general case with all command aliases, if you don't like one of the supplied aliases, or if it gets in the way, you can easily get rid of it. Using the GDB `break` emulator alias `b` as an example, the following command removes the supplied GDB `b` emulation alias:

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

You can also run:

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

which allows you to run the native LLDB `breakpoint` command with just `b`.

A more directed way to explore what's available in LLDB is to use the `apropos` command: It searches the LLDB help documentation for a word and dumps a summary help string for each matching command. For example:

```
(lldb) apropos file
The following commands may relate to 'file':
...
log enable                -- Enable logging for a single log channel.
memory read               -- Read from the memory of the process being
                           debugged.
memory write              -- Write to the memory of the process being
                           debugged.
platform process launch   -- Launch a new process on a remote platform.
platform select           -- Create a platform if needed and select it as
                           the current platform.
plugin load               -- Import a dylib that implements an LLDB
                           plugin.
process launch             -- Launch the executable in the debugger.
process load              -- Load a shared library into the current
                           process.
source                   -- A set of commands for accessing source file
                           information
... and so forth ...
```

Use the `help` command with command aliases to understand their construction. For instance, type the following command to see an explanation of the supplied GDB `b` emulation and its implementation:

```
(lldb) help b
...
'b' is an abbreviation for '_regexp-break'
```

Another feature of the `help` command is shown by investigating the command `break command add` (used in [Setting Breakpoints](#) (page 32)). To demonstrate it, you should execute the command and examine the help system results:

```
(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...
```

Arguments to a command that are specified in the help output in angle brackets (such as `<breakpt-id>`), indicate that the argument is a common argument type for which further help is available by querying the command system. In this case, to learn more about `<breakpt-id>`, enter:

```
(lldb) help <breakpt-id>  
  
<breakpt-id> -- Breakpoint IDs consist major and minor numbers; the major  
etc...
```

Using `help` often and exploring the LLDB help documentation is a great way to familiarize yourself with the scope of LLDB capabilities.

# GDB and LLDB Command Examples

The tables in this chapter list commonly used GDB commands and present equivalent LLDB commands and alternative forms. Also listed are the built-in GDB compatibility aliases in LLDB.

Notice that full LLDB command names can be matched by unique short forms, which can be used instead. For example, instead of `breakpoint set`, `br se` can be used.

## Execution Commands

GDB	LLDB
Launch a process with no arguments.	
(gdb) run (gdb) r	(lldb) process launch (lldb) run (lldb) r
Launch a process with arguments <args>.	
(gdb) run <args> (gdb) r <args>	(lldb) process launch -- <args> (lldb) r <args>
Launch process a.out with arguments 1 2 3 without having to supply the args every time.	
% gdb --args a.out 1 2 3 (gdb) run ... (gdb) run ...	(% lldb -- a.out 1 2 3 (lldb) run ... (lldb) run ...
Launch a process with arguments in a new terminal window (OS X only).	
—	(lldb) process launch --tty -- <args> (lldb) pro la -t -- <args>
Launch a process with arguments in an existing Terminal window, /dev/ttys006 (OS X only).	



GDB	LLDB
—	<pre>(lldb) process launch --tty=/dev/ttys006 -- &lt;args&gt; (lldb) pro la -t/dev/ttys006 -- &lt;args&gt;</pre>
Set environment variables for process before launching.	
<pre>(gdb) set env DEBUG 1</pre>	<pre>(lldb) settings set target.env-vars DEBUG=1 (lldb) set se target.env-vars DEBUG=1</pre>
Set environment variables for process and launch process in one command.	
	<pre>(lldb) process launch -v DEBUG=1</pre>
Attach to the process with process ID 123.	
<pre>(gdb) attach 123</pre>	<pre>(lldb) process attach --pid 123 (lldb) attach -p 123</pre>
Attach to a process named a.out.	
<pre>(gdb) attach a.out</pre>	<pre>(lldb) process attach --name a.out (lldb) pro at -n a.out</pre>
Wait for a process named a.out to launch and attach.	
<pre>(gdb) attach -waitfor a.out</pre>	<pre>(lldb) process attach --name a.out --waitfor (lldb) pro at -n a.out -w</pre>
Attach to a remote GDB protocol server running on the system eorgadd, port 8000.	
<pre>(gdb) target remote eorgadd:8000</pre>	<pre>(lldb) gdb-remote eorgadd:8000</pre>
Attach to a remote GDB protocol server running on the local system, port 8000.	
<pre>(gdb) target remote localhost:8000</pre>	<pre>(lldb) gdb-remote 8000</pre>
Attach to a Darwin kernel in kdp mode on the system eorgadd.	
<pre>(gdb) kdp-reattach eorgadd</pre>	<pre>(lldb) kdp-remote eorgadd</pre>
Do a source-level single step in the currently selected thread.	
<pre>(gdb) step (gdb) s</pre>	<pre>(lldb) thread step-in (lldb) step (lldb) s</pre>

GDB	LLDB
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
Step out of the currently selected frame.	
(gdb) finish	(lldb) thread step-out (lldb) finish
Backtrace and disassemble every time you stop.	
—	(lldb) target stop-hook add Enter your stop hook command(s). Type 'DONE' to end. > bt > disassemble --pc > DONE Stop hook #1 added.

## Breakpoint Commands

GDB	LLDB
Set a breakpoint at all functions named main.	

GDB	LLDB
(gdb) break main	(lldb) breakpoint set --name main (lldb) br s -n main (lldb) b main
Set a breakpoint in file <code>test.c</code> 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 basename is <code>main</code> .	
(gdb) break main (Note: This will break on any C functions named <code>main</code> .)	(lldb) breakpoint set --method main (lldb) br s -M main
Set a breakpoint at an Objective-C function: <code>-[NSString stringWithFormat:]</code> .	
(gdb) break -[NSString stringWithFormat:]	(lldb) breakpoint set --name "-[NSString stringWithFormat:]" (lldb) b -[NSString stringWithFormat:]
Set a breakpoint at all Objective-C methods whose selector is <code>count</code> .	
(gdb) break count (Note: This will break on any C or C++ functions named <code>count</code> .)	(lldb) breakpoint set --selector count (lldb) br s -S count
Set a breakpoint by a regular expression on a function name.	
(gdb) rbreak regular-expression	(lldb) breakpoint set --regex regular-expression (lldb) br s -r regular-expression
Set a breakpoint by a regular expression on a source file's 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
List all breakpoints.	

GDB	LLDB
(gdb) info break	(lldb) breakpoint list (lldb) br l
Delete a breakpoint.	
(gdb) delete 1	(lldb) breakpoint delete 1 (lldb) br del 1

## 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  Note: 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.	

GDB	LLDB
—	<pre>(lldb) watch set var global (lldb) watchpoint modify -c '(global==5)' (lldb) c ... (lldb) bt * thread #1: tid = 0x1c03, 0x00000000100000ef5 a.out`modify + 21 at main.cpp:16, stop reason = watchpoint 1 frame #0: 0x00000000100000ef5 a.out`modify + 21 at main.cpp:16 frame #1: 0x00000000100000eac a.out`main + 108 at main.cpp:25 frame #2: 0x000007fff8ac9c7e1 libdyld.dylib`start + 1 (int32_t) global = 5</pre>
List all watchpoints.	
(gdb) info break	<pre>(lldb) watchpoint list (lldb) watch l</pre>
Delete a watchpoint.	
(gdb) delete 1	<pre>(lldb) watchpoint delete 1 (lldb) watch del 1</pre>

## Examining Variables

GDB	LLDB
Show the arguments and local variables for the current frame.	
<pre>(gdb) info args and (gdb) info locals</pre>	<pre>(lldb) frame variable (lldb) fr v</pre>
Show the local variables for the current frame.	

GDB	LLDB
(gdb) info locals	(lldb) frame variable --no-args (lldb) fr v -a
Show the contents of the local variable bar.	
(gdb) p bar	(lldb) frame variable bar (lldb) fr v bar (lldb) p bar
Show the contents of the local variable bar formatted as hex.	
(gdb) p/x bar	(lldb) frame variable --format x bar (lldb) fr v -f x bar
Show the contents of the global variable baz.	
(gdb) p baz	(lldb) target variable baz (lldb) ta v baz
Show the global/static variables defined in the current source file.	
—	(lldb) target variable (lldb) ta v
Display the variables argc and argv 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 variables argc and argv only when you stop in the function named main.	
—	(lldb) target stop-hook add --name main --one-liner "frame variable argc argv" (lldb) ta st a -n main -o "fr v argc argv"
Display the variable *this only when you stop in the C class named MyClass.	
—	(lldb) target stop-hook add --classname MyClass --one-liner "frame variable *this" (lldb) ta st a -c MyClass -o "fr v *this"

## Evaluating Expressions

GDB	LLDB
Evaluate a generalized expression in the current frame.	
<pre>(gdb) print (int) printf ("Print nine: %d.", 4 + 5)</pre> <p>Or if you don't want to see void returns:</p> <pre>(gdb) call (int) printf ("Print nine: %d.", 4 + 5)</pre>	<pre>(lldb) expr (int) printf ("Print nine: %d.", 4 + 5)</pre> <p>Or use the print alias:</p> <pre>(lldb) print (int) printf ("Print nine: %d.", 4 + 5)</pre>
Create and assign a value to a convenience variable.	
<pre>(gdb) set \$foo = 5 (gdb) set variable \$foo = 5</pre> <p>Or use the print command:</p> <pre>(gdb) print \$foo = 5</pre> <p>Or use the call command:</p> <pre>(gdb) call \$foo = 5</pre> <p>To specify the type of the variable:</p> <pre>(gdb) set \$foo = (unsigned int) 5</pre>	<p>LLDB evaluates a variable declaration expression as you would write it in C:</p> <pre>(lldb) expr unsigned int \$foo = 5</pre>
Print the Objective-C description of an object.	
<pre>(gdb) po [SomeClass returnAnObject]</pre>	<pre>(lldb) expr -O -- [SomeClass returnAnObject]</pre> <p>Or use the po alias:</p> <pre>(lldb) po [SomeClass returnAnObject]</pre>
Print the dynamic type of the result of an expression.	

GDB	LLDB
<pre>(gdb) set print object 1 (gdb) p someCPPObjectPtrOrReference</pre> <p>Note: Only for C++ objects.</p>	<pre>(lldb) expr -d run-target -- [SomeClass returnAnObject] (lldb) expr -d run-target -- someCPPObjectPtrOrReference</pre> <p>Or set dynamic type printing as default:</p> <pre>(lldb) settings set target.prefer-dynamic run-target</pre>
Call a function to stop at a breakpoint in the function.	
<pre>(gdb) set unwindonsignal 0 (gdb) p function_with_a_breakpoint()</pre>	<pre>(lldb) expr -u 0 -- function_with_a_breakpoint()</pre>

## Examining Thread State

GDB	LLDB
Show the stack backtrace for the current thread.	
<pre>(gdb) bt</pre>	<pre>(lldb) thread backtrace (lldb) bt</pre>
Show the stack backtraces for all threads.	
<pre>(gdb) thread apply all bt</pre>	<pre>(lldb) thread backtrace all (lldb) bt all</pre>
Backtrace the first five frames of the current thread.	
<pre>(gdb) bt 5</pre>	<pre>(lldb) thread backtrace -c 5 (lldb) bt 5 (lldb-169 and later) (lldb) bt -c 5 (lldb-168 and earlier)</pre>
Select a different stack frame by index for the current thread.	
<pre>(gdb) frame 12</pre>	<pre>(lldb) frame select 12 (lldb) fr s 12 (lldb) f 12</pre>



GDB	LLDB
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 a 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 thread register rax.	
(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`  The LLDB command uses backticks to evaluate an expression and insert the scalar result.
Show the general-purpose registers for the current thread formatted as signed decimal.	

GDB	LLDB
—	<pre>(lldb) register read --format i</pre> <pre>(lldb) re r -f i</pre> <p>LLDB now supports the GDB shorthand format syntax, but no space is permitted after the command:</p> <pre>(lldb) register read/d</pre> <p>Note: LLDB tries to use the same format characters as <code>printf(3)</code> when possible. Type <code>help format</code> to see the full list of format specifiers.</p>
Show all registers in all register sets for the current thread.	
<pre>(gdb) info all-registers</pre>	<pre>(lldb) register read --all</pre> <pre>(lldb) re r -a</pre>
Show the values for the registers named <code>rax</code> , <code>rsp</code> and <code>rbp</code> in the current thread.	
<pre>(gdb) info all-registers rax rsp rbp</pre>	<pre>(lldb) register read rax rsp rbp</pre>
Show the values for the register named <code>rax</code> in the current thread formatted as binary.	
<pre>(gdb) p/t \$rax</pre>	<pre>(lldb) register read --format binary rax</pre> <pre>(lldb) re r -f b rax</pre> <p>LLDB now supports the GDB shorthand format syntax, but no space is permitted after the command:</p> <pre>(lldb) register read/t rax</pre> <pre>(lldb) p/t \$rax</pre>
Read memory from address <code>0xbffff3c0</code> and show four hex <code>uint32_t</code> values.	

GDB	LLDB
(gdb) x/4xw 0xbffff3c0	<pre>(lldb) memory read --size 4 --format x --count 4 0xbffff3c0  (lldb) me r -s4 -fx -c4 0xbffff3c0  (lldb) x -s4 -fx -c4 0xbffff3c0</pre> <p>LLDB now supports the GDB shorthand format syntax, but no space is permitted after the command:</p> <pre>(lldb) memory read/4xw 0xbffff3c0  (lldb) x/4xw 0xbffff3c0  (lldb) memory read --gdb-format 4xw 0xbffff3c0</pre>
Read memory starting at the expression argv[0].	
(gdb) x argv[0]	<pre>(lldb) memory read `argv[0]`</pre> <p>Note that any command can inline a scalar expression result (as long as the target is stopped) using backticks around any expression:</p> <pre>(lldb) memory read --size `sizeof(int)` `argv[0]`</pre>
Read 512 bytes of memory from address 0xbffff3c0 and save results to a local file as text.	
<pre>(gdb) set logging on (gdb) set logging file /tmp/mem.txt (gdb) x/512bx 0xbffff3c0 (gdb) set logging off</pre>	<pre>(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</pre>
Save binary memory data to a file starting at 0x1000 and ending at 0x2000.	
<pre>(gdb) dump memory /tmp/mem.bin 0x1000 0x2000</pre>	<pre>(lldb) memory read --outfile /tmp/mem.bin --binary 0x1000 0x1200  (lldb) me r -o /tmp/mem.bin -b 0x1000 0x1200</pre>
Disassemble the current function for the current frame.	
(gdb) disassemble	<pre>(lldb) disassemble --frame  (lldb) di -f</pre>

GDB	LLDB
Disassemble any functions named <code>main</code> .	
<code>(gdb) disassemble main</code>	<code>(lldb) disassemble --name main</code> <code>(lldb) di -n main</code>
Disassemble an address range.	
<code>(gdb) disassemble 0x1eb8 0x1ec3</code>	<code>(lldb) disassemble --start-address 0x1eb8</code> <code>                  --end-address 0x1ec3</code> <code>(lldb) di -s 0x1eb8 -e 0x1ec3</code>
Disassemble 20 instructions from a given address.	
<code>(gdb) x/20i 0x1eb8</code>	<code>(lldb) disassemble --start-address 0x1eb8</code> <code>                  --count 20</code> <code>(lldb) di -s 0x1eb8 -c 20</code>
Show mixed source and disassembly for the current function for the current frame.	
—	<code>(lldb) disassemble --frame --mixed</code> <code>(lldb) di -f -m</code>
Disassemble the current function for the current frame and show the opcode bytes.	
—	<code>(lldb) disassemble --frame --bytes</code> <code>(lldb) di -f -b</code>
Disassemble the current source line for the current frame.	
—	<code>(lldb) disassemble --line</code> <code>(lldb) di -l</code>

## Executable and Shared Library Query Commands

GDB	LLDB
List the main executable and all dependent shared libraries.	
<code>(gdb) info shared</code>	<code>(lldb) image list</code>
Look up information for a raw address in the executable or any shared libraries.	

GDB	LLDB
(gdb) info symbol 0x1ec4	(lldb) image lookup --address 0x1ec4 (lldb) im loo -a 0x1ec4
Look up functions matching a regular expression in a binary.	
(gdb) info function <FUNC_REGEX>	<p>This one finds debug symbols: (lldb) image lookup -r -n &lt;FUNC_REGEX&gt;</p> <p>This one finds non-debug symbols: (lldb) image lookup -r -s &lt;FUNC_REGEX&gt;</p> <p>Provide a list of binaries as arguments to limit the search.</p>
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 the a.out module.	
—	(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

## Miscellaneous

GDB	LLDB
Echo text to the screen.	
(gdb) echo Here is some text\n	(lldb) script print "Here is some text"
Remap 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  Note: If your source files are no longer located in the same location as when the program was built—maybe the program was built on a different computer—you need to tell the debugger how to find the sources at the local file path instead of the build system file path.
Supply a catchall directory to search for source files in.	
(gdb) directory /my/path	(No equivalent command yet.)

# Using LLDB as a Standalone Debugger

This chapter describes the workflow and operations in a basic Terminal debugging session. Where appropriate, LLDB operations are compared to similar GDB operations.

Most of the time, you use the LLDB debugger indirectly through the Xcode debugging features, and you issue LLDB commands using the Xcode console pane. But for development of open source and other non-GUI based application debugging, LLDB is used from a Terminal window as a conventional command line debugger. To use LLDB as a command-line debugger, you should understand how to:

- Load a process for debugging
- Attach a running process to LLDB
- Set breakpoints and watchpoints
- Control the process execution
- Navigate in the process being debugged
- Inspect variables for state and value
- Execute alternative code

The Xcode IDE automates many of these operations with its full integration of LLDB into the source editing, build, and “run for debugging” cycle with graphical controls. Knowing how these operations work from the command line also helps you understand and use the full power of the LLDB debugger in the Xcode console pane.

## Specifying the Program to Debug

First, you need to set the program to debug. As with GDB, you can start LLDB and specify the file you want to debug using the command line. Type:

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

Or you can specify the executable file to debug after it is already running using the `file` command:

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

## Setting Breakpoints

Next, you might want to set up breakpoints to begin your debugging after the process has been launched. Setting breakpoints was discussed briefly in [LLDB Command Structure](#) (page 6). To see all the options for breakpoint setting, use `help breakpoint set`. For instance, type the following to set a breakpoint on any use of a method named `alignLeftEdges::`:

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

To find out which breakpoints you've set, type the `breakpoint list` command and examine what it returns as follows:

```
(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
```

In LLDB there are two parts to a breakpoint: the logical specification of the breakpoint, which is what the user provides to the `breakpoint set` command, and the locations in the code that match that specification. For example, a 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 included inline in different places in your code.

One piece of information provided by the `breakpoint list` command output is that the logical breakpoint has an integer identifier, and its locations have identifiers within the logical breakpoint. The two are joined by a period (.)—for example, 1.1 in the example above.

Because the logical breakpoints remain live, if another shared library is loaded that includes another implementation of the `alignLeftEdges:` selector, the new location is added to breakpoint 1 (that is, a 1.2 breakpoint is set on the newly loaded selector).



The other piece of information in the breakpoint listing is whether the breakpoint location was *resolved* or not. A location is resolved when the file address it corresponds to gets loaded into the program being debugged. For instance, if you set a breakpoint in a shared library that later is unloaded, that breakpoint location remains but it is no longer resolved.

LLDB acts like GDB with the command:

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

Like GDB, LLDB always makes a breakpoint from your specification, even if it didn't find any locations that match the specification. To determine whether the expression has been resolved, check the `locations` field using `breakpoint list`. LLDB reports the breakpoint as `pending` when you set it. By looking at the breakpoints with `pending` status, you can determine whether you've made a typo in defining the breakpoint when no locations are found by examining the `breakpoint set` output. For example:

```
(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.
```

Either on all the locations generated by your logical breakpoint, or on any one of the particular locations that logical breakpoint resolved to, you can delete, disable, set conditions, and ignore counts using breakpoint-triggered commands. For instance, if you want to add a command to print a backtrace when LLDB hit the breakpoint numbered 1.1, you execute the following command:

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

By default, the `breakpoint command add` command takes LLDB command-line commands. To specify this default explicitly, pass the `--command` option (`breakpoint command add --command ...`). Use the `--script` option if you implement your breakpoint command using a Python script instead. The LLDB help system has extensive information explaining `breakpoint command add`.

## Setting Watchpoints

In addition to breakpoints, LLDB supports watchpoints to monitor variables without stopping the running process. Use `help watchpoint` to see all the commands for watchpoint manipulations. For instance, enter the following commands to watch a variable named `global` for a write operation, and to stop 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)
```

## Launching the Program with LLDB

Once you've specified what program you are going to debug and set a breakpoint to halt it at some interesting location, you need to start (or *launch*) it into a running process. To launch a program with LLDB, use the `process launch` command or one of its built-in aliases:

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

You can also attach LLDB to a process that is already running—the process running the executable program file you specified earlier—by using either the process ID or the process name. When attaching to a process by name, LLDB supports the `--waitfor` option. This option tells LLDB to wait for the next process that has that name to appear and then attach to it. For example, here are three commands to attach to the `Sketch` process, assuming that the process ID is 123:

```
(lldb) process attach --pid 123
(lldb) process attach --name Sketch
(lldb) process attach --name Sketch --waitfor
```

After you launch or attach LLDB to a process, the process might stop for some reason. For example:

```
(lldb) process attach -p 12345
Process 46915 Attaching
Process 46915 Stopped
1 of 3 threads stopped with reasons:
* thread #1: tid = 0x2c03, 0x00007fff85cac76a, where =
  libSystem.B.dylib`__getdirenties64 + 10,
  stop reason = signal = SIGSTOP, queue = com.apple.main-thread
```

Note the line that says “1 of 3 threads stopped with reasons:” and the lines that follow it. In a multithreaded environment, it is very common for more than one thread to hit your breakpoint(s) before the kernel actually returns control to the debugger. In that case, you will see all the threads that stopped for the reason listed in the stop message.

## Controlling Your Program

After launching, LLDB allows the program to continue until you hit a breakpoint. The primitive commands for process control all exist under the thread command hierarchy. Here’s one example:

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

---

**Note:** In its present version (lldb-300.2.24), LLDB can operate on only one thread at a time, but it is designed to support saying “step over the function in Thread 1, and step into the function in Thread 2, and continue Thread 3,” and so on in a future revision.

---

For convenience, all the stepping commands have easy aliases. For example, `thread continue` is invoked with just `c`, and the same goes for the other stepping program commands—which are much the same as in GDB. For example:

```
(lldb) thread step-in // The same as "step" or "s" in GDB.
(lldb) thread step-over // The same as "next" or "n" in GDB.
```

```
(lldb) thread step-out // The same as "finish" or "f" in GDB.
```

By default, LLDB has defined aliases to all common GDB process control commands (for instance, `s`, `step`, `n`, `next`, `finish`). If you find that GDB process control commands you are accustomed to using don't exist, you can add them to the `~/.lldbinit` file using `command alias`.

LLDB also supports the *step by instruction* versions:

```
(lldb) thread step-inst // The same as "stepi" / "si" in GDB.  
(lldb) thread step-over-inst // The same as "nexti" / "ni" in GDB.
```

LLDB has a *run until line* or *frame exit* stepping mode:

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

This command runs the thread until the current frame reaches line 100. If the code skips around line 100 in the course of running, execution stops when the frame is popped off the stack. This command is a close equivalent to the GDB `until` command.

LLDB, by default, shares the terminal with the process being debugged. In this mode, much like debugging with GDB, when the process is running anything you type goes to the STDIN of the process being debugged. To interrupt that process, type `CTRL+C`.

However, if you attach to a process—or launch a process—with the `--no-stdin` option, the command interpreter is always available to enter commands. Always having an `(lldb)` prompt might be a little disconcerting to GDB users at first, but it is useful. Using the `--no-stdin` option allows you to set a breakpoint, watchpoint, and so forth, without having to explicitly interrupt the program you are debugging:

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

There are many LLDB commands that won't work while the process being debugged is running: The command interpreter lets you know when a command is inappropriate most of the time. (If you find any instances where the command interpreter isn't flagging a problem case, please file a bug: [bugreport.apple.com](http://bugreport.apple.com).)

The commands that work while a process 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 subject of disabling STDIN for a process running in LLDB presents a good opportunity to show how to set debugger properties in general. For example, if you always want to run in `--no-stdin` mode, set it as a generic process property using the LLDB `settings` command. The LLDB `settings` command is equivalent to the GDB `set` command. To do this, type:

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

In LLDB, `settings` are organized hierarchically, enabling you to discover them easily. In addition, almost anywhere that you can specify a setting on a generic entity (threads, for example), you can also apply the option to a particular instance. View the current LLDB settings with the `settings list` command. You can explore how the `settings` command works in detail using the `help settings` command.

## Examining Thread State

After a process has stopped, LLDB chooses a current thread and a current frame in that thread (on stop, this is always the bottommost frame). Many of the commands for inspecting state work on this current thread or frame.

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

```
(lldb) thread list
Process 46915 state is Stopped
* thread #1: tid = 0x2c03, 0x00007fff85cac76a, where =
  libSystem.B.dylib`__getdirent64 + 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 asterisk (\*) indicates that thread #1 is the current thread. To get a backtrace for that thread, enter 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/SKMain.m:11
frame #10: 0x0000000100000f20, where = Sketch`start + 52
```

Provide a list of threads to backtrace, or use the keyword `all` to see all threads.

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

Set the selected thread, the one which will be used by default in all the commands in the next section, with the `thread select` command, where the thread index is the one shown in the `thread list` listing, using

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

## Examining the Stack Frame State

The most convenient way to inspect a frame's arguments and local variables is to use the `frame variable` command.

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

```
c = (NSInteger) 0x00000001001253b0
```

If you don't specify any variable names, all arguments and local variables are shown. If you call `frame variable`, passing in the name or names of particular local variables, only those variables are printed. For instance:

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

You can pass in a path to some subelement of one of the available locals, and that subelement is printed. For instance:

```
(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 simple operations such as `&`, `*`, `->`, `[]` (no overloaded operators). The array brackets can be used on pointers to treat pointers as arrays. For example:

```
(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 performs “object printing” operations on variables. Currently, LLDB supports only Objective-C printing, using the object's `description` method. Turn this feature on by passing the `-O` flag to `frame variable`.

```
(lldb) frame variable -O self
```



```
(SKTGraphicView *) self = 0x0000000100208b40 &lt;SKTGraphicView: 0x100208b40>
```

To select another frame to view, use the `frame select` command.

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

To move the view of the process up and down the stack, pass the `--relative` option (short form `-r`). LLDB has the built-in aliases `u` and `d`, which behave like their GDB equivalents.

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

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

## Executing Alternative Code

Expressions can also be used to call functions, as in this example:

```
(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. As a result, you don't have to quote your whole expression, or backslash protect quotes, and so forth.

The results of the expressions are stored in persistent variables (of the form `$(0-9)+`) that you can use in further expressions, such as:

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

# Document Revision History

This table describes the changes to *LLDB Quick Start Guide*.

Date	Notes
2013-09-18	New document that explains LLDB basics and provides GDB to LLDB command equivalents.



Apple Inc.  
Copyright © 2013 Apple Inc.  
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer or device for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-branded products.

Apple Inc.  
1 Infinite Loop  
Cupertino, CA 95014  
408-996-1010

Apple, the Apple logo, Carbon, Mac, Numbers, Objective-C, OS X, and Xcode are trademarks of Apple Inc., registered in the U.S. and other countries.

**APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

**IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT, ERROR OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.**

**Some jurisdictions do not allow the exclusion of implied warranties or liability, so the above exclusion may not apply to you.**