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. 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`__getdirentries64 +
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.)

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`__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 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/SKTMain.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 = (NSUInteger) 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
```