Appendix A: LLDB for GDB Users – Command Summary

LLDB is the supported engine underlying all debugging operations for use with Xcode and all officially distributed Apple development products. Some users who are new to Xcode might be more familiar with GDB commands. By default, LLDB contains a library of aliases modeled on GDB commands in order to ease getting started.

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

Note: Full LLDB command names can be matched by unique short forms. For example, instead of `breakpoint set`, you can use `br se`.

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)	

—	`(lldb) process launch --tty=/dev/ttys006 -- <args>`
	`(lldb) pro la -t/dev/ttys006 -- <args>`

Set environment 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`
Set environment variables for process and launch process in one command	
	`(lldb) process launch -v DEBUG=1`
Attach to the process with process ID 123	

`(gdb) attach 123`	`(lldb) process attach --pid 123`
	`(lldb) attach -p 123`

Attach to a process named `a.out`	

`(gdb) attach a.out`	`(lldb) process attach --name a.out`
	`(lldb) pro at -n a.out`

Wait for a process named `a.out` to launch and attach	

`(gdb) attach -waitfor a.out`	`(lldb) process attach --name a.out --waitfor`
	`(lldb) pro at -n a.out -w`

Attach to a remote GDB protocol server running on the system `eorgadd`, port 8000	
`(gdb) target remote eorgadd:8000`	`(lldb) gdb-remote eorgadd:8000`
Attach to a remote GDB protocol server running on the local system, port 8000	
`(gdb) target remote localhost:8000`	`(lldb) gdb-remote 8000`
Attach to a Darwin kernel in kdp mode on the system `eorgadd`	
`(gdb) kdp-reattach eorgadd`	`(lldb) kdp-remote eorgadd`
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`

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

Set a breakpoint in file `test.c` 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 `main`	

`(gdb) break main`
(Note: This will break on any C functions named `main`.)
	`(lldb) breakpoint set --method main`
	`(lldb) br s -M main`

Set a breakpoint at an Objective-C function: `-[NSString stringWithFormat:]`	

`(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 `count`	

`(gdb) break count`
(Note: This will break on any C or C++ functions named `count`.)
	`(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) 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	

—	`(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``
	`(int32_t) global = 5`

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`

Examining Variables

GDB	LLDB
Show the 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 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	

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

Create and assign a value to a convenience variable	

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

Print the Objective-C `description` of an object	

`(gdb) po [SomeClass returnAnObject]`	`(lldb) expr -O -- [SomeClass returnAnObject]`
	
	Or use the `po` alias:
	`(lldb) po [SomeClass returnAnObject]`

Print the dynamic type of the result of an expression	
`(gdb) set print object 1`	`(lldb) expr -d run-target -- [SomeClass`

`(gdb) p someCPPObjectPtrOrReference`
 	
Note: Only for C++ objects.
	`returnAnObject]`
	`(lldb) expr -d run-target -- someCPPObjectPtrOrReference`
	
	Or set dynamic type printing as default:
	`(lldb) settings set target.prefer-dynamic run-target`

Call a function to stop at a breakpoint in the function	

`(gdb) set unwindonsignal 0`
`(gdb) p function_with_a_breakpoint()`
	`(lldb) expr -u 0 -- function_with_a_breakpoint()`

Examining 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 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	

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

Show all registers in all register sets for the current thread	

`(gdb) info all-registers`	`(lldb) register read --all`
	`(lldb) re r -a`

Show the values for the registers named `rax`, `rsp` and `rbp` in the current thread	
`(gdb) info all-registers rax rsp rbp`	`(lldb) register read rax rsp rbp`
Show the values for the register named `rax` 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 is permitted after the command:
	`(lldb) register read/t rax`

	`(lldb) p/t $rax`
Read memory from address `0xbffff3c0` and show four 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 is permitted after the command:
	`(lldb) memory read/4xw 0xbffff3c0`
	`(lldb) x/4xw 0xbffff3c0`
	`(lldb) memory read --gdb-format 4xw 0xbffff3c0`

Read memory starting at the expression `argv[0]`	

`(gdb) x argv[0]`	``(lldb) memory read `argv[0]` ``
	
	Note that any command can inline a scalar expression result (as long as the target is stopped) using back ticks (` `` `) around any expression:
	``(lldb) memory read --size `sizeof(int)` `argv[0]` ``

Read 512 bytes of memory from address `0xbffff3c0` and save 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 to a file starting at `0x1000` and ending at `0x2000`	

`(gdb) dump memory /tmp/mem.bin 0x1000 0x2000`	`(lldb) memory read --outfile /tmp/mem.bin --binary 0x1000 0x1200`
	`(lldb) me r -o /tmp/mem.bin -b 0x1000 0x1200`

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	

—	`(lldb) disassemble --frame --mixed`
	`(lldb) di -f -m`

Disassemble the current function for the current frame and show the opcode bytes	

—	`(lldb) disassemble --frame --bytes`
	`(lldb) di -f -b`

Disassemble the current source line for the current frame	

—	`(lldb) disassemble --line`
	`(lldb) di -l`

Executable and Shared Library Query Commands

GDB	LLDB
List the main executable and all dependent shared libraries	
`(gdb) info shared`	`(lldb) image list`
Look up information for a 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 a regular expression in a binary	

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

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.)