

Why should I use LLDB?

Deepak Panickal
Ewan Crawford



- Introductions
- Motivation
- LLDB Command-line and MI Interface
- Leveraging LLVM Libraries
- LLDB C++ and Python API
- Future Developments
- Summary
- Q/A



Introductions



- Hi!
- We're debugger engineers at Codeplay
- Working on LLDB for the past three years on customer projects
- In the past year, have been adapting LLDB for Qualcomm's Hexagon DSP
- Upstreaming relevant patches to the community



- Codeplay!
- Heterogeneous compiler experts
- 35 talented engineers
- Based out of Edinburgh, Scotland



Edinburgh Castle

- We work on
 - R&D (both self and externally funded)
 - Work for hire contracting, compiler/debugger tech
 - Standards via bodies such as Khronos



Motivation



- LLDB has a clean, **maintainable** architecture
- Works on all major platforms (Windows being added) and many architectures
- Inviting more people to start using it as their primary debugger
- We aim to showcase LLDB's features from the **user** point of view
- Examples



Command-line Interface



- LLDB's command-line interface is **verbose**
- Commands have long and **short** forms
- Structured well
- **Tab completion** of commands (not on Windows yet)

- Easy jump from **GDB** to **LLDB** as a lot of commands are similar
- <http://lldb.llvm.org/lldb-gdb.html>



- **help** is very detailed

```
(lldb) help
Debugger commands:

apropos      -- Find a list of debugger commands related to a particular word/subject.
breakpoint   -- A set of commands for operating on breakpoints. Also see _regexp-break.
expression   -- Evaluate a C/ObjC/C++ expression in the current program context, using user defined
                variables and variables currently in scope.
frame        -- A set of commands for operating on the current thread's frames.
...
print        -- ('expression --') Evaluate a C/ObjC/C++ expression in the current program context,
                using user defined variables and variables currently in scope.
q            -- ('quit') Quit out of the LLDB debugger.
r            -- ('process launch -c /bin/sh --') Launch the executable in the debugger.
s            -- ('thread step-in') Source level single step in specified thread (current thread,
                if none specified).
step         -- ('thread step-in') Source level single step in specified thread (current thread,
                if none specified).
t            -- ('thread select') Select a thread as the currently active thread.
x            -- ('memory read') Read from the memory of the process being debugged.
```

For more information on any command, type 'help <command-name>'.



- help <command-name>

```
(lldb) help breakpoint
The following subcommands are supported:
  clear   -- Clears a breakpoint or set of breakpoints in the executable.
  delete   -- Delete the specified breakpoint(s). If no breakpoints are specified, delete them all.
  enable   -- Enable the specified disabled breakpoint(s). If no breakpoints are specified, enable all of
them.
  list-- List some or all breakpoints at configurable levels of detail.
  set -- Sets a breakpoint or set of breakpoints in the executable.
```

- help <command-name> <subcommand-name>

```
(lldb) help breakpoint set
      Sets a breakpoint or set of breakpoints in the executable.
Syntax: breakpoint set <cmd-options>

  -c <expr> ( --condition <expr> )
      The breakpoint stops only if this condition expression evaluates to true.

  -f <filename> ( --file <filename> )
```



- Any unique **short** form of a command can be used

```
(lldb) watchpoint set variable count  
(lldb) w s v count
```

- **Settings** are detailed and easily accessible

```
(lldb) settings set target.process.stop-on-sharedlibrary-events on  
  
(lldb) settings set target.output-path stdout.txt
```



- **apropos** for searching commands related to a word

```
(lldb) apropos disassem

The following built-in commands may relate to 'disassem':
  disassemble -- Disassemble bytes in the current function, or elsewhere in the executable program as specified by the user.

The following settings variables may relate to 'disassem':

  disassembly-format -- The default disassembly format string to use when disassembling instruction sequences.
  stop-disassembly-count -- The number of disassembly lines to show when displaying a stopped context.
  stop-disassembly-display -- Control when to display disassembly when displaying a stopped context.
  target.x86-disassembly-flavor -- The default disassembly flavor to use for x86 or x86-64 targets.
  target.use-hex-immediates -- Show immediates in disassembly as hexadecimal.
  target.hex-immediate-style -- Which style to use for printing hexadecimal disassembly values.
```



- Rich debugging with detailed process state information

```
(lldb) file a.out
Current executable set to 'a.out' (x86_64).

(lldb) breakpoint set --name main --file example.c
Breakpoint 1: where = a.out`main + 8 at example.c:22, address = 0x00000000004005d7

(lldb) b factorial
Breakpoint 2: where = a.out`factorial + 7 at example.c:5, address = 0x00000000004005a4

(lldb) r
Process 2210 launched: '/home/ewan/Desktop/Scratch/talk/c_example/a.out' (x86_64)
Process 2210 stopped
* thread #1: tid = 2210, 0x00000000004005d7 a.out`main + 8 at example.c:22, name = 'a.out', stop reason = breakpoint 1.1
  frame #0: 0x00000000004005d7 a.out`main + 8 at example.c:22
19  {
20
21      int number;
-> 22      printf("Enter a number to calculate factorial of: ");
23      scanf("%d",&number);
24
25      int fact = factorial(number);
(lldb)
```



LLDB-MI Interface



- Machine Interface(MI) is a **protocol** allowing a front end to communicate with the debugger in text form
- Many existing IDEs currently use GDB for their debugging through **GDB-MI**
- LLDB-MI understands the GDB-MI protocol and runs as a separate executable using the LLDB C++ API
- More details about the design of LLDB-MI
<http://www.codeplay.com/portal/lldb-mi-driver---part-1-introduction>



LLDB-MI Interface

Debug - C:\Users\Ewan\Scratch\C\fatorial.c - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help

Quick Access Java Debug

Debug Breakpoints Expressions Modules

Debug_factorial [C/C++ Remote Application]

- factorial [4889]
 - Thread [1] (Suspended : Breakpoint)
 - _Z9fatorial() at factorial.c:23 0xb6f964fc
 - main() at factorial.c:23 0xb6f96558
 - _libc_init() at ?:0 0xb6f19506
 - _start() at ?:0 0xb6f96498

LLDB (via lldbmi)

Disassembly factorial.c

```
#include <stdio.h>

int factorial(int n)
{
    int result = 1;
    int i;

    for (i = 1; i <= n; i++)
    {
        result *= i;
    }

    return result;
}
```

Variables

Name	Type	Value
(*)_ n	int	6
(*)_ n	int	6
(*)_ result	int	1
(*)_ i	int	1

Registers

Name	Value	Description
General Registers		General Purpose and FPU Register Group
r0	0x00000006	
r1	0xbe846a14	
r2	0x00000001	
r3	0x00000006	
r4	0xbe846a14	
r5	0xbe846a1c	
r6	0x00000001	
r7	0xb6f9653c	
r8	0x00000000	
r9	0x00000000	
r10	0x00000000	
r11	0xbe8469cc	
r12	0x1CCCCCCC	



LLDB-MI Interface



Debug - Eclipse

File Edit Navigate Search Project Run Window Help

Quick Access Java Debug

Disassembly factorial.c

b6f964d4: .long 0xffffffff58
4 {
b6f964d8: str r11, [sp, #-4]!
b6f964dc: add r11, sp, #0
b6f964e0: sub sp, sp, #20
b6f964e4: str r0, [r11, #-16]
5 int result = 1;
b6f964e8: mov r3, #1
b6f964ec: str r3, [r11, #-8]
8 for (i = 1; i <= n;i++)
b6f964f0: mov r3, #1
b6f964f4: str r3, [r11, #-12]
b6f964f8: b 0xb6f96518
.10 result *= i;
♦ b6f96fc: ldr r3, [r11, #-8]
b6f96500: ldr r2, [r11, #-12]

Variables

Name	Type	Value
(*)_n	int	6
(*)_n	int	6
(*)_result	int	1
(*)_i	int	1

Registers

Memory

Monitors

&(result) : 0xBE8469C4 <Hex>

Address	0 - 3	4 - 7	8 - B	C - F
BE8469C0	01000000	01000000	1C6A84BE	DC6984BE
BE8469D0	3C65F986	06000000	0C6A84BE	0795F1B6
BE8469E0	F86984BE	00000000	00000000	00000000
BE8469F0	00000000	9864F986	A48EF986	948EF986
BE846A00	8C8EF986	9C8EF986	00000000	FF96F8B6
BE846A10	01000000	296B84BE	00000000	336B84BE
BE846A20	486B84BE	856B84BE	9E6B84BE	B36B84BE
BE846A30	C86B84BE	E36B84BE	EE6B84BE	0DC6C84BE
BE846A40	206C84BE	316C84BE	446C84BE	206E84BE
BE846A50	4C6E84BE	626E84BE	8C6E84BE	A56E84BE
BE846A60	B96E84BE	E36E84BE	096F84BE	156F84BE
BE846A70	2F6F84BE	526F84BE	5C6F84BE	676F84BE
BE846A80	00000000	10000000	D7B00F00	06000000
BE846A90	00100000	11000000	64000000	03000000



Leveraging LLVM Libraries



- Get all of the following for free
- Up to date **language support** with Clang compiler infrastructure
- Disassembling instructions for many architectures.
- LLDB has both IR interpreter for simple expressions and JIT-ing for complex **multi-line expressions** on supported architectures leveraging the power of Clang and LLVM



- **expression** command for expression evaluation

```
(lldb) help expr

Evaluate a C/ObjC/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).

Syntax: expression <cmd-options> -- <expr>

-D <count> ( --depth <count> )
    Set the max recurse depth when dumping aggregate types (default is infinity).

-F ( --flat )
    Display results in a flat format that uses expression paths for each variable or member.
...
.

Examples:

expr my_struct->a = my_array[3]
expr char c[] = "foo"; c[0]
```



- Multi-line expression evaluation

```
(lldb) expr
Enter expressions, then terminate with an empty line to evaluate:
1: int i = 0;
2: for (;i<10;++i){
3:   printf("%d\n",factorial(i));
4: }
1
1
2
6
24
120
720
5040
40320
362880
```

- Can declare **local** variables



- C++ expressions in a C program

```
(lldb) expr  
  
Enter expressions, then terminate with an empty line to evaluate:  
1: auto square_lambda = [] (int i) { return (i*i);};  
2: int $squared = square_lambda(16);  
  
(lldb) print $squared  
(int) $squared = 256
```

- Variable types are deduced

```
(lldb) expr -T -- structVar  
(complexStruct) $4 = {  
    (unsigned int) firstInt = 2  
    (long) secondInt = -1  
    (char [3]) firstString = "abc"  
    (char *) secondString = 0x00000000004005f4 "abc"  
}
```



- Using the disassembler

```
(lldb) dis -n square(int) -m

a.out`square(int) at main.c:5
4     int square(int n)
5     {
6
a.out`square(int):
    0x40052d <+0>: pushq  %rbp
    0x40052e <+1>: movq   %rsp, %rbp
    0x400531 <+4>: movl   %edi, -0x4(%rbp)
a.out`square(int) + 7 at main.c:7
6
-> 7     return n * n;
8 }
-> 0x400534 <+7>: int3
    0x400535 <+8>: cld
    0x400537 <+10>: imull  -0x4(%rbp), %eax
a.out`square(int) + 14 at main.c:8
7     return n * n;
8 }
9
```



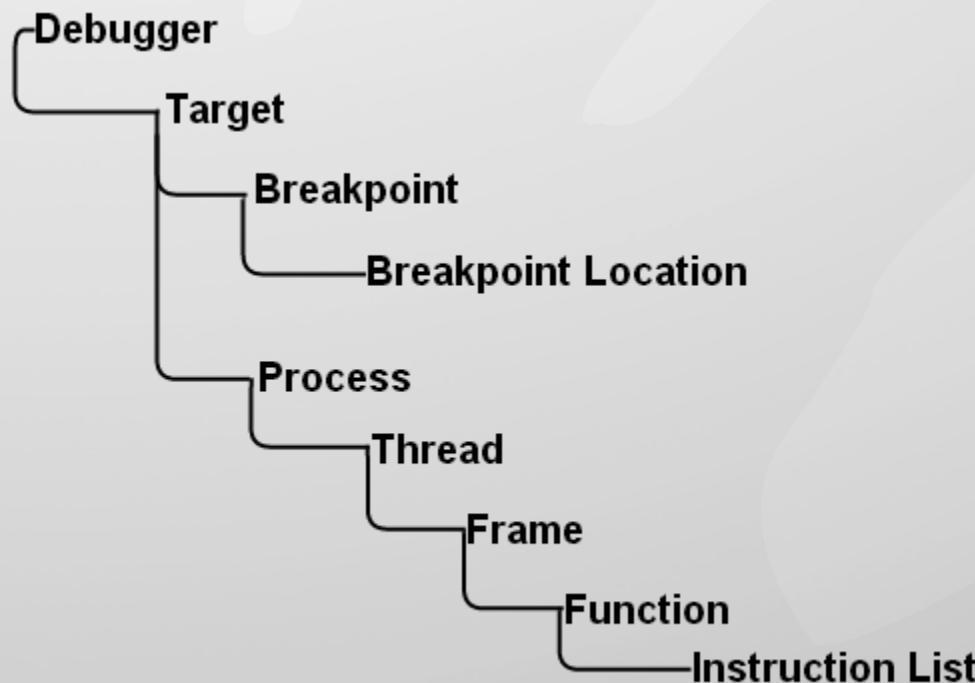
LLDB C++ API



- LLDB provides a **public C++ API** which allows features like disassembly and object file inspection to be integrated into user created C++ applications
- The API consists of a light **wrapper** around internal LLDB objects. e.g. Target and Frame
- API can be used through the LLDB shared library
- C++ API documentation
http://lldb.llvm.org/cpp_reference/html/index.html



- API Object Organisation



LLDB Python API



- The LLDB API is available through Python script bindings
- Allows any Python script loading the LLDB module to **automate** repetitive debugging tasks
- API can be used inside LLDB's **embedded python** interpreter
- Python API documentation
http://lldb.llvm.org/python_reference/index.html



- Python bindings are created using SWIG
 - SWIG (Simplified Wrapper and Interface Generator) provides a **wrapper** around C/C++ code, allowing it to be called by a different programming language
 - **Interface files** are used by SWIG to define how functions are made visible in the target language
 - SWIG is run at build time to create a `lldb.py` module
 - Developers can define SWIG interface files for **any supported language** they want to use LLDB functionality in. e.g. Java, C#, Lua



```
import lldb

# Create a new debugger instance
debugger = lldb.SBDebugger.Create()
debugger.SetAsync (False)
target = debugger.CreateTargetWithFileAndArch ("./a.out", lldb.LLDB_ARCH_DEFAULT)

# Set breakpoint on function defined by command line argument. WARNING: No error checking.
main_bp = target.BreakpointCreateByName (sys.argv[1], target.GetExecutable().GetFilename());

process = target.LaunchSimple (None, None, os.getcwd()) # launch process
if process.GetState() == lldb.eStateStopped:
    thread = process.GetThreadAtIndex (0)                  # Get the first thread
    frame = thread.GetFrameAtIndex (0)                    # Get the first frame
    allVars = frame.get_all_variables()
    print("all variables: ")
    for var in allVars:                                  # print all variables
        print str(var)
```

```
$ python printVars.py factorial
all variables:
(int) n = 6
(int) result = 0
(int) i = 0
```



- Using the embedded Python interpreter
 - Full featured interpreter
 - Can import external modules
 - (lldb) script ‘command’, will run without dropping into interpreter
 - (lldb) command script import ‘script file’, loads external scripts

```
(lldb) help script
    Pass an expression to the script interpreter for evaluation and return the results. Drop into the
interactive interpreter if no expression is given. This command takes 'raw' input (no need to quote stuff).

Syntax: script [<script-expression-for-evaluation>]

(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()' or Ctrl-D.

>>> import math

>>> math.ceil(6.7)
7.0
```



- The API can be used inside the interpreter through convenience variables
 - llDb.debugger
 - llDb.frame
 - llDb.target
 - llDb.process
 - llDb.thread

```
(lldb) script
>>> print llDb.frame
frame #0: 0x00000000004004f1 a.out`main + 4 at loop.c:4

>>> print llDb.frame.GetSP()
140737488346352

>>> print hex(llDb.frame.GetSP())
0x7fffffffdfcf0L
```



- New LLDB commands can also be created using Python functions
 - Work like native lldb commands
 - Clean way to extend LLDB functionality
 - Example from LLDB Python Reference
<http://lldb.llvm.org/python-reference.html>

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

```
(lldb) command script import ~/ls.py

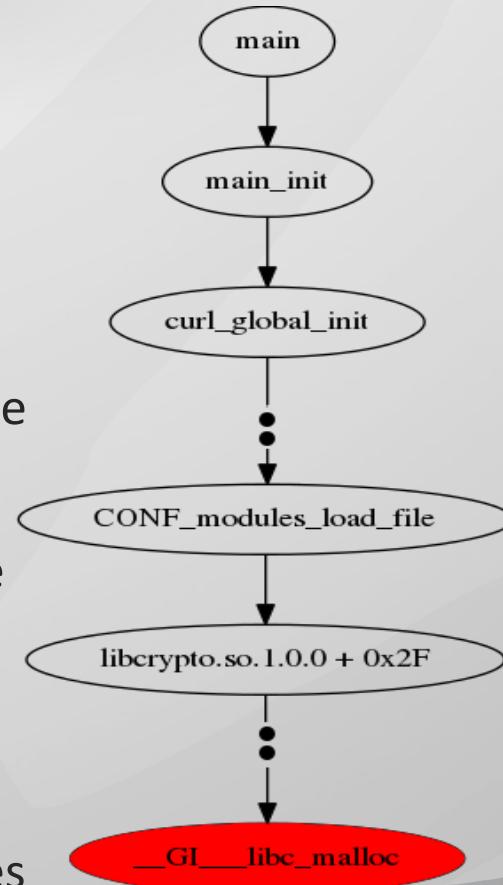
(lldb) ls -l ~/LLVM/llvm/tools/lldb
total 88
drwxrwxr-x  4 ewan ewan 4096 Mar 30 16:32 cmake
-rw-rw-r--  1 ewan ewan 1205 Mar 30 16:32 CMakeLists.txt
```



- One of the most powerful features the Python API enables is **running scripts** when breakpoints are hit
 - Python function definition:
`breakpoint_function_wrapper(frame, bp_loc, dict)`
 - Function returns False to prevent LLDB from stopping when breakpoint is hit
 - Not stopping allows **profiling data** to be silently gathered every time a breakpoint is hit



- Breakpoint Script Example
 - Walks up the call stack on breakpoint hit and adds functions to call graph
 - Graph printed as image using pydot module
 - Graph generated from LLDB debugging the popular **curl** application
 - Script was attached to a conditional breakpoint on all calls to **malloc()** of 3 bytes



```
# Full code available from https://github.com/EwanC/BreakpointScript
callGraph = CallGraph();                                # User defined class
root = callGraph.addNode("Root",-1);

def bpStack (frame, bp_loc, internal_dict): # Run when breakpoint is hit

    thread = frame.GetThread()
    numFrames = thread.GetNumFrames()

    lastnode = root                                # Parent function
    for f in reversed(range(0, numFrames)): # Walk the stack
        name = thread.GetFrameAtIndex(f).GetFunctionName()

        # Debug info not available
        if name == "???" or name == None:
            # Use location in module for name
            # Omitted here for brevity

        # Update call graph with function
        node = callGraph.update(name,lastnode,f)
        lastnode = node

    return False # LLDB doesn't stop when breakpoint is hit

def draw(): # Print graph to png image
    callGraph.graph.write_png('BPCallStack.png')
```



Future Developments



- Improved Windows support including addition of **native Windows debugging** and better command-line features
- Improved LLDB-MI support for more IDEs
- Support for more **architectures** to the many LLDB already supports(X86, ARM, X86_64, ARM64, ...)
- **C# API** interface
- Lots of devs working to make it awesome!



Summary



- Why use LLDB?
 - Reuse LLVM components, so ideal as part of LLVM based toolchain
 - Supported on all major **platforms** and supports many architectures
 - **Embedded Python** interpreter with API
 - Clean, maintainable and high-performance debugger
 - Powerful **multi-line** expression evaluation
 - Info on building LLDB is available at <http://lldb.llvm.org/build.html>
- Contact
 - Deepak Panickal, deepak@codeplay.com
 - Ewan Crawford, ewan@codeplay.com
 - For LLDB dev issues, the mailing list is lldb-dev@cs.uiuc.edu

