The LLDB Debugger in FreeBSD

Ed Maste
The FreeBSD Foundation / The University of Cambridge
Boulder, Colorado, USA / Cambridge, England
Email: emaste@freebsd.org

February 23, 2015

Abstract

LLDB is a modern, high-performance debugger in the LLVM family of projects, and is built as a modular and reusable set of components on top of the Clang/LLVM foundation. Originally developed for Mac OS X, it now also supports FreeBSD, Linux and Windows. This paper provides an overview of the design of LLDB, compares it with the existing GNU debugger in the FreeBSD base system, and presents the path to importing LLDB as FreeBSD's debugger.

1 Introduction

BSD operating systems, including FreeBSD, have the concept of a "base system," an integrated core that is developed, maintained, tested, and released as an integrated whole. Some major components of the base system are the kernel, userland libraries, system binaries, and development tool chain – and a key component of the tool chain is the debugger.

The FreeBSD base system has long relied on the GNU debugger, GDB. The very first FreeBSD release included a copy of GDB 3.5, and every release since then has included some version of GDB. For more than a decade the project followed GDB's development, and over time a number of different contributors incorporated new versions into the FreeBSD source tree.

This work produced a growing set of changes, and each new import required additional effort to resolve conflicts where changes had been made in both the FreeBSD and the upstream (that is, the GDB project's) version. Project members attempted to get the changes incorporated into the upstream GDB project, but were unsuccessful in doing so. Eventually the growing maintenance burden discouraged effort within the FreeBSD community, and GDB imports stopped with GDB 6.1.1 in June 2004. Since that time additional bug fixes and enhancements have been made to GDB in the FreeBSD tree, but no full updates have been done. Those changes consist of 8 commits merged from upstream, 4 from Apple's GDB version, 24 changes specific to FreeBSD, and 18 "bookkeeping" changes such as tracking changes in APIs in other components or accommodating files that moved in the source hierarchy.

1.1 GPLv3

In 2007 GDB's license was updated to version 3 of the GNU General Public License (GPLv3). The GPLv3 includes additional restrictions that certain FreeBSD consumers, contributors and users find unpalatable. To date the FreeBSD project has avoided importing GPLv3 software into the base system.

The last release of GDB available under the GPLv2 is GDB 6.6. Compared to version 6.1.1 it lacks sufficient improvements to make an update worthwhile. It is therefore unlikely that a new version of GDB will be imported into the FreeBSD base system, although it will remain available through the third-party ports tree and binary packages.

1.2 A New Debugger – LLDB

As there was no path forward with GDB, it became increasingly clear that a new debugger was required for the FreeBSD base system. In 2010 Apple announced at their World-Wide Developer conference that they were working on a debugger project named LLDB, and they released it as open source later that year.

Domain	Committers	Commits
apple.com	16	4040
intel.com	7	496
gmail.com	20	445
freebsd.org	1	351
google.com	6	255
filcab.net	1	97
debian.org	1	68
mentor.com	1	48
codeplay.com	2	47
valvesoftware.com	1	39
All others	18	187

Table 1: Top ten email domains by LLDB commit count, 2012-2014

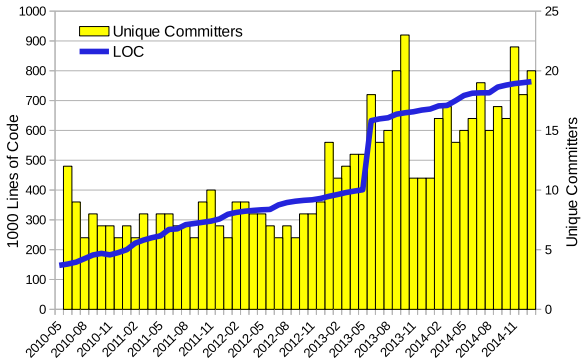

Figure 1: Committer and code size growth in LLDB

LLDB is part of the LLVM family of projects, and builds on other LLVM components. As with the rest of LLVM, it is released under the University of Illinois/NCSA license. This is a permissive BSD-like license that is an ideal match for the FreeBSD project's base system licensing philosophy.

LLDB has grown beyond an Apple project, and has major contributions from open-source groups within companies like Intel and Google, and from FreeBSD, Debian, and other independent, open-source projects. For illustrative purposes the top ten email domains by number of LLDB commits in 2012 through 2014 are shown in Table 1. LLDB's growth in code size and unique committers per month is shown in Figure 1, using data from Black Duck Software's Open Hub website[1].

2 LLDB Design

LLDB is built as a set of modular components on top of the LLVM compiler infrastructure project and the Clang front-end. Reusing Clang and LLVM components allows for a great deal of functionality with lower effort and smaller code size in comparison with other debuggers.

As an example, debuggers require an expression parser for interpreting input from the user. Some debuggers rely on an independently developed, ad-hoc parser. In contrast, LLDB includes a built-in copy of Clang and uses it as the parser. This provides extremely high fidelity in interpreting expressions: the parser for the command line is the same one that handles the source code.

2.1 Core LLDB Classes

A high level block diagram of LLDB's design is shown in Figure 2. LLDB's core functionality is implemented in a set of classes that roughly correspond to the following top-level source directories:

API LLDB provides a rich Application Programming Interface (API) for both C++ and Python consumers. LLDB vends a C++ API[2], which is converted into Python bindings[3] with SWIG, the Simplified Wrapper and Interface Generator. There is interest in supporting bindings for additional languages including C, C#, Javascript and Lua.

Breakpoint The Breakpoint classes handle resolving breakpoints by name, address, source file and line, etc., and managing lists of breakpoints.

Command Interpreter The command interpreter presents the `(lldb)` prompt to the user and interprets their commands. It also implements the `script` command which provides access to a built-in Python interpreter.

Data Formatters Data formatters present variables in a format convenient to the user, hiding the details of data structures used by the language's standard library or other runtime libraries. For example, a C++ std::string will be displayed as

```
(std::string) s = "Text"
```

rather than

```
(std::string) s = {
  _M_dataplus = (_M_p = "Text")
}
```

Expression Parser The expression parser provides an interface to the built-in copy of Clang for interpreting the user's source-level expressions, passed to the `expression` command or its alias `p`. It is possible to declare variables, use multiple statements, and use loops. For example,

```
(lldb) expr int $j = 0;
  for(int $i = 0; $i < 5; $i++) $j = $j + $i
(lldb) p $j
(int) $j = 10
```

Host LLDB currently builds for Android, FreeBSD, Linux, OS X and Windows hosts. The Host subdirectory contains the abstraction for these platforms.

Symbol Symbol classes handle obtaining debugging information, and the lookup of symbols to serve user expressions.

Target The Target subdirectory contains all of the classes that manage LLDB's view of the debuggee, also known as the inferior process. This includes the processes' memory, loaded modules, threads, and "Thread Plans" for single stepping, running to a breakpoint, calling functions and other actions.

Utility A number of register list, pseudoterminal, string and other utility classes.

2.2 LLDB Plugins

Classes for specific object file formats, system runtims, and target platforms are implemented as LLDB Plugins. These plugins are not currently runtime- loadable, but do follow consistent APIs that simplify adding support for new formats or platforms. A detailed view of LLDB's plugins appears in Figure 3.

The main task in adding FreeBSD support was the creation of a FreeBSD process plugin. This plugin manages FreeBSD target processes and threads, signals, and the ptrace interface.

The FreeBSD host class and the elf-core process plugin also required significant effort for FreeBSD support. The test suite also had to be made more portable by removing platform-specific assumptions.

3 Using LLDB

LLDB's command interpreter is designed with a consistent, structured syntax. Commands generally follow the pattern "`noun verb`" – for example, `thread list` or `breakpoint set`. The syntax is more verbose than GDB's, and longtime GDB users will take some time to adapt.

The benefit is that the command set is discoverable and regular; targeted autocompletion can provide relevant options to the user. As with GDB, commands may be abbreviated to the shortest unique prefix.

An example of starting a debug session may look like:

```
% lldb
(lldb) target create /bin/ls
Current executable set to '/bin/ls' (x86_64).
(lldb) breakpoint set -name main
Breakpoint 1: where = ls`main + 33 at
  ls.c:163, address = 0x00000000004023f1
(lldb) process launch
```

LLDB has powerful support for command aliases, and includes a built-in set of aliases for many GDB commands. Using GDB aliases, the same result as above could be achieved with:

```
% lldb /bin/ls
Current executable set to '/bin/ls' (x86_64).
(lldb) b main
Breakpoint 1: where = ls`main + 33 at
  ls.c:163, address = 0x00000000004023f1
(lldb) run
```

The built-in aliases are limited though, and some of the more esoteric overloaded functionality provided by GDB is not available through aliases. This is particularly true for breakpoints – in GDB the breakpoint command argument may be a line number, file name, function, or address, with sometimes overlapping or conflicting meaning. Migrating to LLDB's syntax and relying on the substring match to allow more concise commands is likely to be the most effective approach.

LLDB can manage multiple targets within a single debugging session. The targets may be local or remote, and may be different CPU architectures.

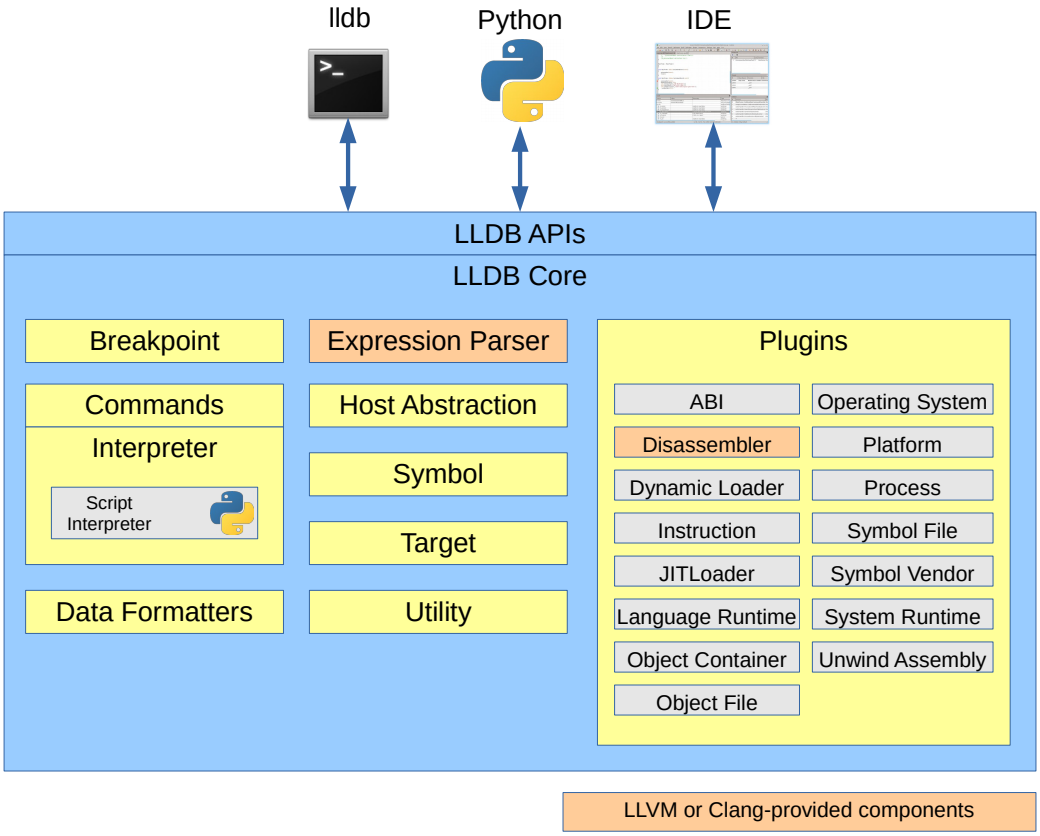

Figure 2: LLDB Block Diagram

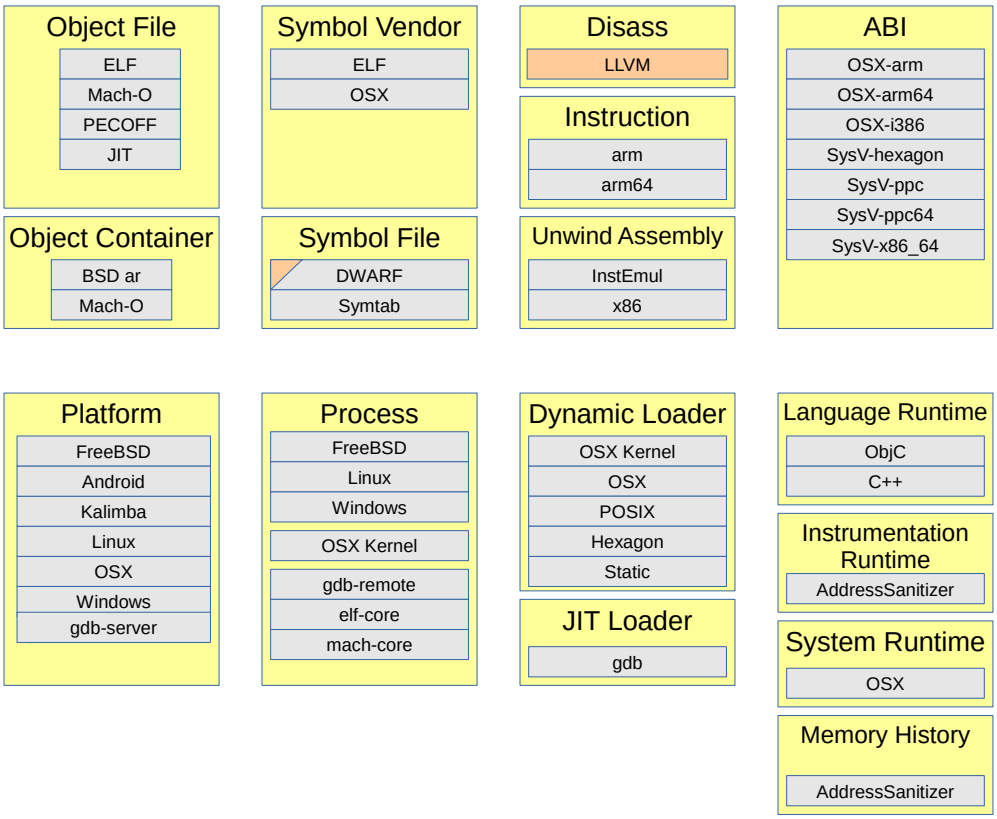

Figure 3: LLDB Plugins

3.1 Breakpoints

Breakpoints are set with the "`breakpoint set`" command. They may be set at a given address, filename and line number, function or method name, or on language-specific execptions. Breakpoints may also be restricted to a specified thread or shared library.

Breakpoints are maintained in a logical state within LLDB, and are later resolved to one or more locations. The logical breakpoint and each resolved location are given integer identifiers, joind with a period. For example, if the third breakpoint matches two locations, they will be called "`3.1`" and "`3.2`".

Breakpoints remain live throughout a debugging session, so loading a shared library with a function or method that matches an existing breakpoint specification results in new locations being added to that breakpoint. Similar, unloading a shared library may remove breakpoint locations. A breakpoint remains set, but in an unresolved state, after unloading all of its locations.

3.2 Examining Debuggee State

Whenever the debuggee process stops, LLDB prints relevant information: the thread that stopped, process location information including address, filename, and line number, the current function and its arguments, and a stop reason. The reported stop reason may be a breakpoint, watchpoint, signal, address exception, or one of a number of target- or language-specific reasons. Finally, a small portion of the source code at the current addresss is shown.

After encoutering a breakpoint or stopping for another reason, LLDB selects the most relevant thread. This will be the one that encountered a breakpoint, received a signal, performed an invalid memory access, or otherwise triggered the stop.

The "`thread list`" command lists all active threads in the debuggee, with "`thread select`" choosing the desired thread for subsequent commands.

To obtain a stack backtrace use the "`thread backtrace`" command, also available as the "`bt`" alias. By default only the current thread's backtrace is shown, but a different thread index may be provided as an argument, or "`all`" to show the stack for each thread.

While examining a backtrace the "`frame select`" command may be used to choose a specific frame. The "`up`" and "`down`" aliases provide short forms for relative frame selection. With a frame selected the "`frame variable`" command will display function arguments and local variables that are in scope.

3.3 Controlling the Debuggee

LLDB groups the single-stepping process control commands under the top-level `thread` command. "`thread step-in`" steps a single source line, continuing into function calls. "`thread step-over`" also steps a single source line, but does not stop inside of a function call. "`thread step-out`" continues until the program returns from the current function. The "`thread until <line>`" command continues until the program reaches the specified source file line, or it returns from the current function.

3.4 Scripting

LLDB provides multiple ways to interact with its scripting capability. The most basic is the "`script`" command, which invokes the embedded Python interpreter and may be used to query current program state through a set of convenience variables. New user-facing commands may also be implemented in Python.

LLDB can also invoke a script after hitting a breakpoint. That script can then control the debugee state (for example, exmaining variables and deciding to continue the process), allowing for complex breakpoint conditions.

Finally, LLDB may be used as a debugger object from a Python script, without using the standalone "`lldb`" binary. The script can "`import lldb`", create a debugger instance and a target, set breakpoints, launch, single step, and continue the target.

4 FreeBSD LLDB Roadmap

Ongoing development effort on the LLDB FreeBSD port takes place directly in the upstream LLDB repository. Earlier in the porting effort new versions of LLDB were imported into FreeBSD as snapshots from the upstream Subversion repository. Functionality was improving so quickly that waiting for a release would miss significant new features and bug fixes.

Upstream development still continues at a rapid pace, but LLDB releases have recently achieved a level of stability and maturity to allow release-based updates.

LLDB version 3.5 was imported in this manner, coincident with the corresponding Clang/LLVM import.

LLDB on FreeBSD works well on the amd64 architecture, for live and core file- based userland debugging. There is also support for the 64-bit MIPS architecture, and 32- and 64-bit PowerPC. LLDB does not yet support FreeBSD on 32- or 64-bit ARM, but support exists in LLDB for other operating systems and it is expected to be a relatively easy port.

A Google Summer of Code (GSoC) 2014 project delivered an initial proof of concept for FreeBSD kernel debugging support; additional work is required to refine this before it may be integrated into LLDB.

A key component of an overall debugging environment is a remote target stub, which allows a debugger on a host computer to attach to the stub and debug a process on a different target computer. LLDB includes a target stub named `lldb-gdbserver` for Linux that reuses the same implementation as for local debugging. Porting it to FreeBSD is a possibility.

Facebook released another debug server named `ds2`, also under a permissive license. It may be a preferable candidate for inclusion in the FreeBSD base system, as it has a smaller footprint and thus may be more suitable for embedded platforms.

An update to LLDB version 3.6 is in progress, along with Clang and LLVM, with the expectation that LLDB will be enabled by default for the amd64 platform.

Acknowledgments

The initial LLDB FreeBSD porting effort was undertaken by Kip Macy and Mark Peek, prior to the author's involvement with LLDB. After the FreeBSD port became usable a number of developers contributed bug fixes or additional CPU architecture support, including Justin Hibbits, Mike Ma, John Wolfe, and others.

Portions of this work are part of the CTSRD project that is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this paper are those of the author and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

References

[1] Black Duck Software, *Open HUB: The LLDB Debugger*, `https://www.openhub.net/p/lldb`, February 2015.

[2] *LLDB API Documentation*, `http://lldb.llvm.org/cpp_reference/html/index.html`

[3] *LLDB python API*, `http://lldb.llvm.org/python_reference/index.html`