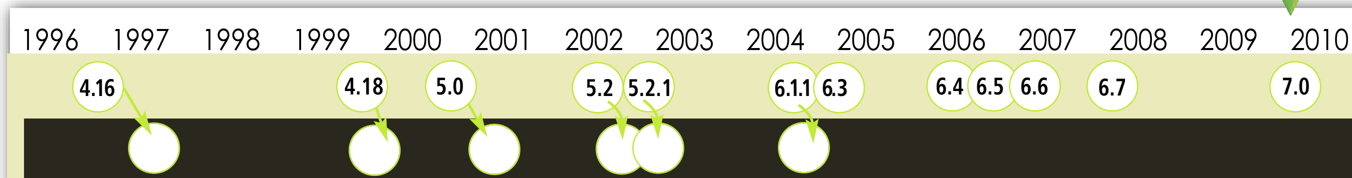
THE SEARCH FOR A NEW DEBUGGER

by Ed Maste

One attribute that distinguishes FreeBSD from other open-source operating systems is the concept of the "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 toolchain–and a key component of the toolchain is the debugger.

The GNU debugger, GDB, has long served as the base system's debugger. In 1993, the very first FreeBSD release included GDB 3.5. Every release since has included a version of GDB. The project followed GDB's development for more than a decade, and a number of different contributors incorporated new versions into the FreeBSD source tree.

This work produced a growing set of changes, and the effort increased with each new import. Project members attempted to have the changes incorporated to the upstream GDB project, but met resistance from the project's maintainers. Eventually this growing maintenance effort wore on the team, and the last import was GDB 6.1.1 in June 2004.



GPLv3

As with other GNU projects, in 2007 GDB's license changed to version 3 of the GNU Public License (GPLv3). The GPLv3 includes some restrictions that major FreeBSD contributors and consumers find unpalatable, and to date the project has avoided including any GPLv3-licensed code in the base system.

SEARCH FOR A NEW DEBUGGER

With a stale version of GDB in the base system and no clear path forward, it was clear FreeBSD needed a new debugger. Several open-source debugger projects were formed, both within and outside of the FreeBSD community. None of them reached critical mass to sustain development and produce a viable debugger we could use.

Then at their 2010 World-Wide Developer Conference, Apple announced they had their own debugger project, LLDB. It was released as open source in June of that year, and the next year it became the default debugger for Xcode, Apple's IDE. LLDB is provided under the University of Illinois/NCSA license which is a permissive, BSD-like license that is an ideal match for the FreeBSD project's licensing philosophy.

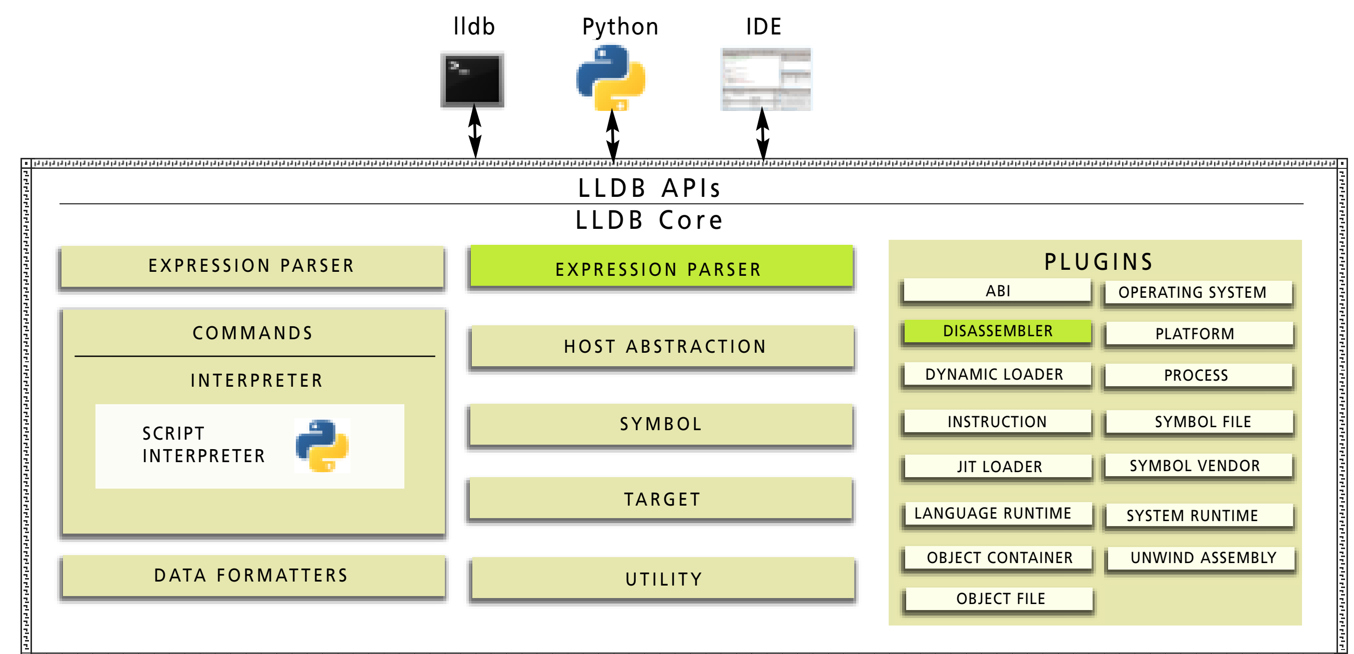
LLDB has since grown beyond an Apple project, with major contributions coming from open-source groups within companies like Intel and Google, and from FreeBSD, Debian, and other independent, open-source projects. Several people contributed to the FreeBSD port of LLDB, and our plan is that it will become the standard debugger in the FreeBSD base system.

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 compared to other debuggers. For example, LLDB has a full Clang compiler built in, used for its expression parser. If an expression is acceptable in a project's source code, it will also be handled by LLDB's expression parser, allowing a user to examine complex classes and data types in great detail and with confidence. LLVM provides LLDB's processor-specific support, including disassembly support and CPU-specific functionality. The modular design also provides the foundation for straightforward support of new processors, languages, and platforms.

A number of overall goals guide LLDB's design, and many of these derive from Clang and LLVM. To achieve high performance and reduce memory usage, LLDB attempts to parse only the debugging information required to perform an action. Threaded, high-performance classes provided by LLVM also contribute to LLDB's speed (see chart below).

LLDB aims to allow customization throughout so that a user can tailor the debugging experience. Variable and value display, type formatters, summary information, commands, prompts, and aliases can all be configured.



LLVM or CLANG-provided components

Like Clang and LLVM, LLDB inherently supports multiple CPU architectures and platforms within the same debugger binary. For example, the same debugger could be used to locally debug a native FreeBSD/amd64 application, examine a core file from a FreeBSD/MIPS system, and remotely debug an application running under a debug server on Linux. It's also possible to have multiple debug sessions running simultaneously in the same debugger.

As a debugger framework, LLDB is designed to be embedded in, or used by, other projects. IDEs and graphical front ends can easily incorporate and build on LLDB's functionality, using a C++ API. A full scripting API is also provided, currently supporting Python bindings. Python is usable from within LLDB, at the command line, for controlling execution after breakpoints, and for implementing new commands. The scripting interface is also available for external use; a Python script can create a debugger object and then use it to examine and control a debuggee's state, evaluate expressions, and so on.

LLDB USE

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 command syntax is somewhat more verbose than GDB, and adapting may take some effort for longtime GDB users. 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 a powerful support for command aliases, and includes a built-in set of aliases for many GDB commands. Using the 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 commands is not available through the 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 most effective.

Some GDB and LLDB Commands for Comparison

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 a process in a new terminal window n/a	(lldb) process launch --tty -- *args*
Attach to a process by pid (gdb) attach ***pid***	(lldb) process attach --pid ***pid*** (lldb) attach -p ***pid***
Source level single step (gdb) step (gdb) s	(lldb) thread step-in (lldb) step (lldb) s
Source level single step over function calls (gdb) next (gdb) n	(lldb) thread step-over (lldb) next (lldb) n
Source level single step out of current function (gdb) finish	(lldb) thread step-out (lldb) finish
Instruction level single step (gdb) stepi (gdb) si	(lldb) thread step-inst (lldb) si
Instruction level single step over function calls (gdb) nexti (gdb) ni	(lldb) thread step-inst-over (lldb) ni
Return immediately from current frame (gdb) return ***return expression***	(lldb) thread return ***return expression***
Set a breakpoint at all functions by name (gdb) break ***name*** (gdb) b ***name***	(lldb) breakpoint set --name ***name*** (lldb) br s -n ***name*** (lldb) b ***name***
Set a breakpoint by file and line (gdb) break ***file***:***line***	(lldb) breakpoint set --file ***file*** --line ***line*** (lldb) br s -f ***file*** -l ***line*** (lldb) b file:***line***
Set a breakpoint in all C++ methods by name (gdb) break ***name*** *assuming no C functions have the same name*	(lldb) breakpoint set --method ***name*** (lldb) br s -M ***name***
List breakpoints (gdb) info break	(lldb) breakpoint list (lldb) br l
Delete a breakpoint (gdb) delete 1	(lldb) breakpoint delete 1 (lldb) br del 1
Show arguments and local variables (gdb) info args *and* (gdb) info locals	(lldb) frame variable (lldb) fr v

(continues)

(Commands for Comparison continued)

The "target create" command (previous page chart) can create multiple targets in a debugging session with "target list" showing the active ones:

```
(lldb) target list
Current targets:
 target #0: /bin/app1 ( arch=x86-64-unknown-freebsd10.1, platform=host )
 target #1: /bin/app2 ( arch=x86-64-unknown-freebsd10.1, platform=host, pid=81166, state=exited )
```

To switch between them use "target select <number>".

BREAKPOINTS

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

Breakpoints are maintained as logical breakpoints within LLDB, which then resolve to one or more locations. The logical breakpoint and each resolved location are given integer identifiers, joined with a dot. For example, if the third breakpoint matches two function names and thus resolves to two locations, they will be called "3.1." and "3.2".

Breakpoints remain live throughout the 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. Similarly, unloading a shared library may remove breakpoint locations. A breakpoint remains set, but in an unresolved state after unloading all of its locations.

Whenever the debuggee process stops, LLDB prints relevant information: the thread that stopped, process location information including the address, filename, and line number, the current function and its arguments, and the 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 address is shown.

```
* thread #1: tid = 100641, 0x00000000004023f1 ls`main(argc=1, argv=0x00007fffffffe760)
+ 33 at ls.c:163, name = 'ls', stop reason = breakpoint 1.1
    frame #0: 0x00000000004023f1 ls`main(argc=1, argv=0x00007fffffffe760) + 33 at ls.c:163
   160  #ifdef COLORS
   161          char termcapbuf[1024];  /* termcap definition buffer */
   162          char tcapbuf[512];       /* capability buffer */
-> 163          char *bp = tcapbuf;
   164  #endif
   165
   166          (void)setlocale(LC_ALL, "");
(lldb)
```

EXAMINING DEBUGEE STATE

After encountering 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 with the "bt" alias. By default 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.

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.

The stepping commands have aliases to match GDB: "s" or "step" for thread step-in, "n" or "next" for thread step-over, and "f" or "finish" for thread step-out.

DATA FORMATTERS

LLDB has built-in support for a number of high-level data structure formats used by language runtimes and libraries. These take the internal representation of a variable and display it in a convenient user-facing format, as might be used in source code.

For FreeBSD, the C++ runtime library formatters are likely the most valuable. LLDB includes support for the two of interest in FreeBSD: libc++ and GNU libstdc++.

As an example, a std::string will show just the string contents by default, when the type formatter is enabled:

```
(lldb) expression str
(string) $1 = "This is a string."
```

Formatters may be disabled in order to see the full details of the data structure, if necessary:

```
lldb) type category disable gnu-libstdc++
(lldb) expression str
(string) $2 = {
  _M_dataplus = {_M_p   = "This is a string."}
}
```

SCRIPTING

LLDB's scripting interface may be accessed in multiple ways. The most basic is the "script" command, which invokes the embedded interpreter and may be used to query current program state through a set of convenience variables. Python may also be used to implement new LLDB commands.

LLDB can also invoke a script after hitting a breakpoint. The script can then control program state (for example, continuing the process), allowing for very complex breakpoint conditions.

Finally, LLDB can be used entirely from a Python script, without involving the stand-alone lldb binary. A script can "import lldb", create a debugger instance and then a target, set breakpoints, launch, single step, and continue the target, and examine variables or evaluate expressions.

LLDB IN FreeBSD ROADMAP

Ongoing development effort on the LLDB FreeBSD port takes place directly in the LLDB repository. It currently works well for the amd64 architecture, for both live and core file-based userland debugging. Core file debugging support also exists for the MIPS architecture. Between CPU support currently in progress by Google and work ongoing in the FreeBSD community, we expect to support 64- and

32-bit versions of the x86, ARM, MIPS, and PowerPC CPU architectures.

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

One key component under development in the LLDB project is a remote debugging stub. This allows an LLDB instance running on one computer to access and control a process running on another. This is especially important for debugging on small embedded devices, which may lack the memory or computing power to accommodate a full-featured debugger. The debugging stub is initially being implemented for Linux, but the port to FreeBSD is relatively straightforward.

Snapshots of the LLDB tree are imported into FreeBSD-Current on an occasional basis. LLDB is not yet built by default, but may be enabled by adding `WITH_LLDB=yes` to `/etc/src.conf` before running "`make buildworld`" as described in the FreeBSD Handbook.

One complication in the FreeBSD base system is that it does not include Python, so the LLDB snapshot is currently built without scripting support. It remains functional, but as a result some of the more interesting and advanced features are not available. We are evaluating different approaches to address this, likely migrating the scripting interface to a runtime rather than compile-time option. This would allow all of LLDB's Python capabilities to be enabled by simply installing the Python package or port.

We expect to update Clang and LLVM in the FreeBSD base system to version 3.5 in the near future. LLDB will be updated at the same time, and we then expect to enable building it by default. ●

Ed Maste manages project development for the FreeBSD Foundation and works in an engineering support role with the University of Cambridge Computer Laboratory. He is also a member of the elected FreeBSD Core Team. Aside from FreeBSD and LLDB, he is a contributor to a number of other open-source projects, including QEMU and Open vSwitch. He lives in Kitchener, Canada, with his wife, Anna, and sons, Pieter and Daniel.