LLDB Tutorial: Adding debugger support for your target

LLVM 2016 tutorial

Deepak Panickal
Andrzej Warzyński

Codeplay Software
@codeplaysoft

March 18, 2016
Outline

• LLDB architecture crash course
  ▶ Overview of LLDB
  ▶ User scenarios such as breakpoints, stack-walking etc.
  ▶ Debugging tips

• Both generic and specialized architectures are covered
  ▶ MSP430 lldb debugging, which we have implemented for this tutorial
    ▶ github.com/codeplaysoftware/lldb-msp430
  ▶ ARM architecture is also referred to for the generic cases

• Focusing on debugging ELF executables on Linux
Overview

Part 1: The Basics

Part 2: ELF And Architecture Support

Part 3: Registers

Part 4: Memory and Breakpoints

Part 5: Other Key Features

Part 6: Debugging Tips

Part 7: MSP430 Quick Recap
The Basics
LLDB - Architecture

LLDB offers multiple options:

- **user drivers**: command line, lldb-mi, Python
- **debug API**: ptrace/simulator/runtime/actual drivers
lldb/lldb-server

**lldb**

- Runs on host
- Interacts with the **user**
- Understands symbols, DWARF information, data formats, etc.
- Plugin architecture
  - ProcessGDBRemote, DynamicLoaderPOSIXDYLD, ABISysV_msp430 are some...

**lldb-server**

- Runs on both remote and host, communicates to lldb via RSP over whichever medium is available
- Interacts with the **hardware/simulator**
- Deals with binary data and memory addresses
- Plugin architecture
  - ObjectFileELF, ProcessLinux, are some...
GDB Remote Serial Protocol

- Simple, ASCII message based protocol
- Designed for debugging remote targets
- Originally developed for gdb<->gdbserver communication
- Extended for LLDB, see lldb-gdb-remote.txt

Packet structure:

```plaintext
$ ... # hh
```

checksum

packet data
GDB Remote Serial Protocol

Sample session:

< 1> send packet: +
< 19> send packet: $QStartNoAckMode#b0
< 1> read packet: +
< 6> read packet: $OK#9a
< 1> send packet: +
< 41> send packet: $qSupported:xmlRegisters=i386,arm,mips#12
< 110> read packet: $PacketSize=20000;QStartNoAckMode+;QThreadSuffixSupported+;QListThreadsInStopReply+;.....#9f
< 26> send packet: $QThreadSuffixSupported#e4
< 6> read packet: $OK#9a
< 27> send packet: $QListThreadsInStopReply#21
< 6> read packet: $OK#9a
< 13> send packet: $qHostInfo#9b
< 325> read packet: $triple:61726d2d2d6c696e75782d616e64726f6964;ptrsize:4;watchpoint_exceptions_received:before; endian:little;os_version:3.4.67;...;hostname:6c6f63616c686f7374;default_packet_timeout:20;#7a
< 10> send packet: $vCont?#49
< 17> read packet: $vCont;c;C;s;S#62
< 27> send packet: $qVAttachOrWaitSupported#38
< 4> read packet: $#00
< 16> send packet: $qProcessInfo#dc
< 162> read packet: $pid:596;parent-pid:595;real-uid:0;real-gid:0;effective-uid:0;effective-gid:0;...#ae

- **QThreadSuffixSupported**: Adding a thread-id to packets
- **61726d2d2d6c696e75782d616e64726f6964**: arm--linux-android
Architectures for this talk

- MSP430 from Texas Instruments
  - "Low-power mixed-signal processors (...) for a wide range of industrial and consumer applications." (ti.com)
  - A 16-bit RISC architecture not yet supported by LLDB
  - A lot of tools available, including a gdb-server (mspdebug)
  - There's an LLVM backend, however, according to README.txt:
    
    Disclaimer: This backend should be considered as highly experimental. I never seen nor worked with this MCU...

- ARM
  - Very popular 32/64-bit RISC architecture
  - Already supported by LLDB (lldb-server for Linux/Android)
MSP430

- Registers
  - 16 registers in total
  - r0 - PC
  - r1 - SP
  - r2 - status register
  - r3 - zero register

- Also relevant
  - 2 byte memory addressing
  - Has 27 instructions

- mspdebug instead of lldebug-server
  - mspdebug implements gdb-server
  - We do not modify mspdebug

Figure: The MSP-EXP430G2 dev board (ti.com)
MSP430 - LLDB Architecture

- **lldb**
  - **user driver**
  - **TCP Socket**
  - **GDB RSP**

- **mspdebug**
  - **sim**
  - **gdb-server**
  - **JTAG/USB**

- **hardware**
MSP430 - Required tools

- Required tools (assuming Debian-based linux)

```bash
sudo apt-get install binutils-msp430 gcc-msp430 msp430-libc mspdebug
```

- The code

```c
#include <msp430g2553.h>

int main(void)
{
    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
    P1DIR |= 0x01; // Set P1.0 to output direction

    for (;;)
    {
        volatile unsigned long i;
        P1OUT ^= 0x01; // Toggle P1.0 using exclusive-OR
        i = 99999; // Delay
        do
        {
            i--;
        } while (i != 0);
    }
}
```

- Building the executable

```bash
msp430-gcc -mmcu=msp430g2553 -O0 -g led.c
```
ELF And Architecture Support
Loading the binary

We need to load binary sections/debug info

- LLDB supports ELF/Mach-O/PECOFF out of the box
- For debugging info, DWARF is supported
- MSP430 compiler provides ELF with relatively good DWARF
- As of now

```
(lldb) file ~/led.elf
error: '~/led.elf' doesn't contain any 'host' platform architectures: x86_64, i386
```

- First step towards LLDB understanding MSP430 would be adding the triple
Adding the triple

• Architecture and core

```c
static const CoreDefinition g_core_definitions[] = {
    { eByteOrderLittle, 4, 2, 4, llvm::Triple::arm, ArchSpec::eCore_arm_generic, "arm"},
    // ...

    // MSP430
    { eByteOrderLittle, 2, 2, 2, llvm::Triple::msp430, ArchSpec::eCore_msp430, "msp430" }
};

static const ArchDefinitionEntry g_elf_arch_entries[] = {
    // ...
    { ArchSpec::eCore_arm_generic, llvm::ELF::EM_ARM, LLDB_INVALID_CPUTYPE, 0xFFFFFFFFu, 0xFFFFFFFFu },
    // ...
    { ArchSpec::eCore_msp430, llvm::ELF::EM_MSP430, LLDB_INVALID_CPUTYPE, 0xFFFFFFFFu, 0xFFFFFFFFu }
};
```

• From llldb:

```
(lldb) file led.elf
Current executable set to 'led.elf' (msp430).
(lldb) target list
Current targets:
* target #0: /home/user/examples/led.elf ( arch=msp430-**-, platform=remote-linux )
```
Architecture Support

• Adding the OS (not really required)

```cpp
bool ArchSpec::SetArchitecture (ArchitectureType arch_type, uint32_t cpu, uint32_t sub, uint32_t os)
{
    // ...
    switch (os)
    {
        case llvm::ELF::ELFOSABI_GNU: m_triple.setOS (llvm::Triple::OSType::Linux); break;
        // MSP
        case llvm::ELF::ELFOSABI_STANDALONE: m_triple.setOS (llvm::Triple::OSType::Standalone); break;
    }
    // ...
}
```

• Can see the OS now

```bash
(lldb) target list
Current targets:
  * target #0: /home/user/examples/led.elf ( arch=msp430-**-standalone, platform=remote-linux )
```

• LLDB gets the OS from EI_OSABI in ELF header
Is this really enough?

- To add custom ELF sections modify `ObjectFileELF.cpp`
- Inspecting the ELF sections for MSP430:

```
(lldb) image dump sections led.elf
Sections for '/media/andrzej/Build/msp430/led.elf' (msp430):

<table>
<thead>
<tr>
<th>SectID</th>
<th>Type</th>
<th>File Address</th>
<th>File Off.</th>
<th>File Size</th>
<th>Flags</th>
<th>Section Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000001</td>
<td>regular</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>led.elf.</td>
<td></td>
</tr>
<tr>
<td>0x00000002</td>
<td>code</td>
<td>[0x0000f800-0x0000f89a]</td>
<td>0x00000094</td>
<td>0x0000009a</td>
<td>0x00000006</td>
<td>led.elf..text</td>
</tr>
<tr>
<td>0x00000003</td>
<td>regular</td>
<td>[0x0000200-0x0000202)</td>
<td>0x0000011e</td>
<td>0x00000000</td>
<td>0x00000003</td>
<td>led.elf..noinit</td>
</tr>
<tr>
<td>0x00000004</td>
<td>regular</td>
<td>[0x000ffe0-0x00100000)</td>
<td>0x0000011e</td>
<td>0x00000020</td>
<td>0x00000006</td>
<td>led.elf..vectors</td>
</tr>
<tr>
<td>0x00000005</td>
<td>dwarf-aranges</td>
<td>0x0000000500</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>led.elf..debug_aranges</td>
<td></td>
</tr>
<tr>
<td>0x00000006</td>
<td>dwarf-info</td>
<td>0x000001dc</td>
<td>0x00000150</td>
<td>0x00000000</td>
<td>led.elf..debug_info</td>
<td></td>
</tr>
<tr>
<td>0x00000007</td>
<td>dwarf-abbrev</td>
<td>0x000000096</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>led.elf..debug_abbrev</td>
<td></td>
</tr>
<tr>
<td>0x00000008</td>
<td>dwarf-line</td>
<td>0x0000007cf</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>led.elf..debug_line</td>
<td></td>
</tr>
<tr>
<td>0x00000009</td>
<td>dwarf-frame</td>
<td>0x000000b72</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>led.elf..debug_frame</td>
<td></td>
</tr>
<tr>
<td>0x0000000a</td>
<td>dwarf-str</td>
<td>0x000000b96</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>led.elf..debug_str</td>
<td></td>
</tr>
<tr>
<td>0x0000000b</td>
<td>dwarf-loc</td>
<td>0x000000c2b</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>led.elf..debug_loc</td>
<td></td>
</tr>
<tr>
<td>0x0000000c</td>
<td>dwarf-ranges</td>
<td>0x000000cc47</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>led.elf..debug_ranges</td>
<td></td>
</tr>
<tr>
<td>0x0000000d</td>
<td>regular</td>
<td>0x000000c4f</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>led.elf..shstrtab</td>
<td></td>
</tr>
<tr>
<td>0x0000000e</td>
<td>elf-symbol-table</td>
<td>0x0000000f40</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>led.elf..symtab</td>
<td></td>
</tr>
<tr>
<td>0x0000000f</td>
<td>regular</td>
<td>0x00001660</td>
<td>0x0000004a</td>
<td>0x00000000</td>
<td>led.elf..strtab</td>
<td></td>
</tr>
</tbody>
</table>
```
No debug info?

- Inspect the line table

```
(lldb) image dump line-table led.c
warning: No source filename matched 'led.c'.
error: no source filenames matched any command arguments
```

- Inspect symbols

```
(lldb) image lookup -F main
```

- Though according to msp430-readelf, there seems to be DWARF info

```
$ msp430-readelf --debug-dump=line ~/led.elf
The File Name Table (offset 0x131):
Entry  Dir  Time  Size  Name
 1      0      0      0  led.c
 2      1      0      0  msp430x20x3.h
Line Number Statements:
[0x000000014c] Extended opcode 2: set Address to 0xc148
[0x0000000151] Special opcode 9: advance Address by 0 to 0xc148 and Line by 4 to 5
```
Fixing DWARF

C'mon LLDB, addresses can be two bytes-wide:

```cpp
bool DWARFCompileUnit::Extract(const DWARFDataExtractor &debug_info, lldb::offset_t *offset_ptr)
{
    // ...
    bool addr_size_OK = ((m_addr_size == 4) || (m_addr_size == 8));
    // ...
}
DWARFCompileUnit.cpp
```

```cpp
bool DWARFDebugArrangeSet::Extract(const DWARFDataExtractor &data, lldb::offset_t *offset_ptr)
{
    if ((m_header.version >= 2 && m_header.version <= 5) &&
        (m_header.addr_size == 4 || m_header.addr_size == 8) &&
        (m_header.length > 0))
}
DWARFDebugArrangeSet.cpp
```

```cpp
bool DWARFDebugLine::ParseStatementTable(...)
{
    // ...
    case DW_LNE_set_address:
        if (arg_size == 4)
            state.address = debug_line_data.GetU32(offset_ptr);
}
DWARFDebugLine.cpp
```
Fixing DWARF

C'mon LLDB, **addresses can be two bytes-wide:**

```cpp
bool DWARFCompileUnit::Extract(const DWARFDataExtractor &debug_info, lldb::offset_t *offset_ptr)
{
    // ...
    bool addr_size_OK = ((m_addr_size == 2) || (m_addr_size == 4) || (m_addr_size == 8));
    // ...
}
```

**DWARFCompileUnit.cpp**

```cpp
bool DWARFDebugArrangeSet::Extract(const DWARFDataExtractor &data, lldb::offset_t *offset_ptr)
{
    if ((m_header.version >= 2 && m_header.version <= 5) &&
        (m_header.addr_size == 2 || m_header.addr_size == 4 || m_header.addr_size == 8) &&
        (m_header.length > 0))
}
```

**DWARFDebugArrangeSet.cpp**

```cpp
bool DWARFDebugLine::ParseStatementTable(...)
{
    // ...
    case DW_LNE_set_address:
        if (arg_size == 2)
            state.address = debug_line_data.GetU16(offset_ptr);
}
```

**DWARFDebugLine.cpp**
Finally, we can read the line table:

```
(lldb) image dump line-table led.c
Line table for /media/andrzej/Build/msp430/led.c in `led.elf
0xf83e: /media/andrzej/Build/msp430/led.c:4
0xf844: /media/andrzej/Build/msp430/led.c:5
0xf84a: /media/andrzej/Build/msp430/led.c:6
0xf854: /media/andrzej/Build/msp430/led.c:12
0xf85e: /media/andrzej/Build/msp430/led.c:13
0xf868: /media/andrzej/Build/msp430/led.c:16
0xf87c: /media/andrzej/Build/msp430/led.c:17
```

... and lookup symbols:

```
(lldb) image lookup -F main
1 match found in led.elf:
  Address: led.elf[0xf83e] (led.elf..text + 62)
  Summary: led.elf`main at led.c:4
```
Part 3

Registers
Registers - llvm-server

How does llvm-server know what are the available registers?

- From the Register Context
- Used internally by llvm-server
- Contains information on all supported registers
- Based on information specified in `g_register_infos_<arch>[][]`

For ARM,

```c
static RegisterInfo g_register_infos_arm[] = {
    // NAME     ALT      SZ OFFSET ENCODING FORMAT      EH_FRAME          DWARF
    // ======= ==> ===== === == ========= == = == = == = == == == == == == == == == == = == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == == } RegisterInfos_arm.h
```
Registers - Key data structures

```c
struct RegisterInfo {
    const char *name; // r0
    const char *alt_name; // can be NULL
    uint32_t byte_size; // 4
    uint32_t byte_offset;
    lldb::Encoding encoding; // eEncodingUint, eEncodingIEEE754
    lldb::Format format; // eFormatHext, eFormatFloat
    uint32_t kinds[lldb::kNumRegisterKinds]; // see below
    uint32_t *value_regs; // nullptr
    uint32_t *invalidate_regs; // nullptr
};
```

```c
enum RegisterKind {
    eRegisterKindEHFrame = 0, // the register numbers seen in eh_frame
    eRegisterKindDWARF, // the register numbers seen DWARF
    eRegisterKindGeneric, // e.g. PC, SP, FP
    eRegisterKindProcessPlugin, // e.g. LLDB_INVALID_REGNUM
    eRegisterKindLLDB, // lldb's internal register numbers
    kNumRegisterKinds
};
```

Illb-private-types.h

Illb-enumerations.h

Final steps: mark all the 'special' registers

 ⇒ **What:** eRegisterKindGeneric (field in `struct RegisterInfo`)
 ⇒ **Where:** `g_register_infos_arm[]` (array in `RegisterInfos_arm.h`)

This is required for LLDB to know where to look for PC, SP, FP etc.

**Possible values:**

```c
// Generic Register Numbers
#define LLDB_REGNUM_GENERIC_PC 0 // Program Counter
#define LLDB_REGNUM_GENERIC_SP 1 // Stack Pointer
#define LLDB_REGNUM_GENERIC_FP 2 // Frame Pointer
#define LLDB_REGNUM_GENERIC_RA 3 // Return Address
... llpdb defines.h
```
Registers - llDb

How does llDb know what are the available registers?

- llDb-server reads RegisterContext
- llDb creates a RegisterContext from the info provided by llDb-server

For ARM, llDb → $qRegisterInfo → llDb-server

llDb-server → register info → llDb

```
< 18> send packet: $qRegisterInfo0#72
< 118> read packet: $name:r0;bitsize:32;offset:0;encoding:uint;format:hex;
      set:General Purpose Registers;ehframe:0;dwarf:0;generic:arg1;charset:
< 18> send packet: $qRegisterInfo1#73
```

Similar situation when later reading registers:

```
< 20> send packet: $p5e;thread:4cd6;#63
< 20> read packet: $00000000#00
```
Registers - ARM read/write

- Carried out by lldb-server
- For ARM, this normally done via `ptrace`:

```c
Error
NativeRegisterContextLinux::DoReadRegisterValue(uint32_t offset,
       const char* reg_name,
       uint32_t size,
       RegisterValue &value)
{
   // ...
   long data;
   Error error = NativeProcessLinux::PtraceWrapper(PTRACE_PEEKUSER,
         m_thread.GetID(),
         reinterpret_cast<void*>(offset),
         nullptr,
         0,
         &data);
   // ...
}  
```  

- For `register write`, replace PTRACE_PEEKUSER with PTRACE_POKEUSER
Registers - MSP430

For MSP430, there is no lldb-server

- We use mspdebug instead, which has gdb-server
- We cannot modify gdb-server
- LLDB provides an option to define registers in Python

For MSP430,

```python
msp_register_infos = [
    { 'name': 'r0', 'set': 0, 'bitsize': 16, 'encoding': eEncodingUint, 'format': eFormatAddressInfo, 'alt-name': 'pc' },
    { 'name': 'r1', 'set': 0, 'bitsize': 16, 'encoding': eEncodingUint, 'format': eFormatAddressInfo, 'alt-name': 'sp' },
    { 'name': 'r2', 'set': 0, 'bitsize': 16, 'encoding': eEncodingUint, 'format': eFormatAddressInfo },
    { 'name': 'r3', 'set': 0, 'bitsize': 16, 'encoding': eEncodingUint, 'format': eFormatAddressInfo },
    ...
]
```

We load registers using this command

```
(lldb) settings set plugin.process.gdb-remote.target-definition-file <filename>
```
Registers - MSP430 read/write

- For MSP430, mspdebug does all the magic
- Packet exchange for MSP430:

```plaintext
(lldb) register read
< 7> send packet: $Hg1#e0
< 1> read packet: +
< 4> read packet: $#00
< 1> send packet: +
< 5> send packet: $g#67
< 1> read packet: +
< 68> read packet: $78f87c02030000008202ff5a0000000000000...0000000#3a
< 1> send packet: +

General Purpose Registers:
  r0 = 0xf878  led.elf\main + 58
  r1 = 0x027c
  r2 = 0x0003
  r3 = 0x0000
  r4 = 0x0282
  r5 = 0x5aff
  ...
```
Memory and Breakpoints
Memory

First, user requests to read memory and **lldb sends an m-packet**

(lldb) memory read 0x800
< 12> send packet: $m800,200#c3
< 1> read packet: +
<1028> read packet: $ffffffffffffffffffffffff...fffffffffffffffff#00
< 1> send packet: +
0x00000800: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff

Next, **lldb-server/mspdebug receives the packet** and ... (for ARM):

```cpp
Error
NativeProcessLinux::ReadMemory (...)
{
    Error error = NativeProcessLinux::PtraceWrapper(PTRACE_PEEKDATA,
        &data);
    ...

NativeProcessLinux.cpp
```
Breakpoints - Overview

**Exceptional Control Flow - Traps**

- On Linux, break/trap instruction raises **SIGTRAP**
- SIGTRAP normally indicates a **breakpoint**
- ptrace **intercepts all signals** (apart from SIGKILL) sent to its tracee
Breakpoints - Setting a breakpoint with ptrace

```cpp
Error
SoftwareBreakpoint::CreateSoftwareBreakpoint (...) {
    // ...
    Error error = process.GetSoftwareBreakpointTrapOpcode (size_hint, bp_opcode_size, bp_opcode_bytes);

    // ...
    // Enable the breakpoint.
    uint8_t saved_opcode_bytes [MAX_TRAP_OPCODE_SIZE];
    error = EnableSoftwareBreakpoint (process, addr, bp_opcode_size, bp_opcode_bytes, saved_opcode_bytes);
    // ...
}

Error
SoftwareBreakpoint::EnableSoftwareBreakpoint (...) {
    // ...
    // Save the original opcodes by reading them so we can restore later.
    size_t bytes_read = 0;

    Error error = process.ReadMemory (addr, saved_opcode_bytes, bp_opcode_size, bytes_read);

    // ...
    // Write a software breakpoint in place of the original opcode.
    size_t bytes_written = 0;
    error = process.WriteMemory (addr, bp_opcode_bytes, bp_opcode_size, bytes_written);
    // ...
}
```

SoftwareBreakpoint.cpp
Breakpoints - opcodes

- Most architectures offer a special break instruction
- LLDB needs the corresponding op-code

```c
Error
NativeProcessLinux::GetSoftwareBreakpointTrapOpcode (size_t trap_opcode_size_hint,
    size_t &actual_opcode_size,
    const uint8_t *&trap_opcode_bytes)
{
    // ...
    static const uint8_t g_aarch64_opcode[] = { 0x00, 0x00, 0x20, 0xd4 };  
    static const uint8_t g_arm_breakpoint_opcode[] = { 0xf0, 0x01, 0xf0, 0xe7 };   
    static const uint8_t g_i386_opcode [] = { 0xcc };  
    static const uint8_t g_mips64_opcode[] = { 0x00, 0x00, 0x00, 0x0d }; 
    static const uint8_t g_mips64el_opcode[] = { 0xd0, 0x00, 0x00, 0x00 };  
    static const uint8_t g_thumb_breakpoint_opcode[] = { 0x01, 0xde };
    //...
    NativeProcessLinux.cpp
```

- For MSP430, this is not required (not using llldb-server)
Adding 'op-code' for MSP430:

```cpp
size_t PlatformLinux::GetSoftwareBreakpointTrapOpcode (Target &target,
    BreakpointSite *bp_site)
{
    ArchSpec arch = target.GetArchitecture();
    const uint8_t *trap_opcode = NULL;
    size_t trap_opcode_size = 0;

    switch (arch.GetMachine())
    {
    default:
        assert(false && "CPU type not supported!");
        break;
    
    case llvm::Triple::msp430:
    {
        static const uint8_t g_msp430_opcode[] = { 0x43, 0x43 };
        trap_opcode = g_msp430_opcode;
        trap_opcode_size = sizeof(g_msp430_opcode);
    }
    break;
    } // ...
```

PlatformLinux.cpp
Breakpoints - Step-by-step

▶ Setting the breakpoint (lldb):

(lldb) breakpoint set -n main
Breakpoint 1: where = led.elf`main + 6 at led.c:5, address = 0xf844

▶ Resolving the breakpoint and sending the corresponding packet (lldb):

< 13> send packet: $Z0,f844,2#48
< 1> read packet: +
< 6> read packet: $OK#9a

▶ Overwritting the opcode (lldb-server/mspdebug)

▶ Hitting the breakpoint (lldb-server/mspdebug → lldb):

< 135> read packet: $T0500:44f8;01:7c02;02:0100;03:0000;04:8202;05:ff5a;06:0000;07:0000;08:0000;09:0000;
0a:0000;0b:0000;0c:0000;0d:0000;0e:0000;0f:0000;#9c
...  
Process 1 stopped
* thread #1: tid = 0x0001, 0xf844 led.elf`main + 6 at led.c:5, stop reason = breakpoint 1.1
  frame #0: 0xf844 led.elf`main + 6 at led.c:5
Breakpoints - Summary

Finally (provided that we have debug information):

```
(lldb) breakpoint set --file led.c --line 16
(lldb) continue
Process 1 resuming
Process 1 stopped
* thread #1: tid = 0x0001, 0xf868 led.elf`main + 42 at led.c:16, stop reason =
breakpoint 1.1
  frame #0: 0xf868 led.elf`main + 42 at led.c:16
  13     i = 99999; // Delay
  14     do
  15      {
  ->  16       i--;
  17     } while (i != 0);
  18     }
  19  }
(lldb) next
Process 1 stopped
* thread #1: tid = 0x0001, 0xf87c led.elf`main + 62 at led.c:17, stop reason = step over
    frame #0: 0xf87c led.elf`main + 62 at led.c:17
  14     do
  15      {
  16       i--;
->  17     } while (i != 0);
  18      }
  19  }
```
Other Key Features
ABI - Overview

Implemented as part of the LLDB ABI plugin

- Based on the calling convention for the architecture
- ABI plugin also tells LLDB about callee-saved and caller-saved registers
- Much easier when a stack and frame pointer, or CFI is available

What's CFI?

- Call Frame Information
- Generated by the compiler and available in DWARF
- LLDB uses info from .eh_frame

For MSP430

- Don't have frame pointer or CFI
- The return address is pushed on to the stack
ABI - MSP430

We implemented an experimental MSP430 ABI, \texttt{ABISysV_msp430}

- Example:

```
(lldb) bt
* thread #1: tid = 0x0001, 0xc15e a.out`calc2(y=22) + 6 at led.c:11,
  stop reason = signal SIGTRAP
  * frame #0: 0xc15e a.out`calc2(y=22) + 6 at led.c:11
  frame #1: 0xc154 a.out`calc1(x=22) + 12 at led.c:6
```

- The two frames listed are correct
- However, the variable values are incorrect
- Unwinding to the main function is not being done yet

Our temporary solution for now

- Set the PC to be SP + 2
- This would \textbf{only work} for functions with one argument!
- Have to investigate this further
For ARM,

- Start by implementing CreateDefaultUnwindPlan(), specifying offsets to the PC, FP, based on the CFA

```c++
bool ABISysV_arm::CreateDefaultUnwindPlan (UnwindPlan &unwind_plan)
{
    uint32_t fp_reg_num = dwarf_r11;
    uint32_t pc_reg_num = dwarf_pc;
    // ...
    row->GetCFAValue().SetIsRegisterPlusOffset (fp_reg_num, 2 * ptr_size);
    row->SetRegisterLocationToAtCFAPlusOffset(fp_reg_num, ptr_size * -2, true);
    row->SetRegisterLocationToAtCFAPlusOffset(pc_reg_num, ptr_size * -1, true);
    unwind_plan.AppendRow (row);
    unwind_plan.SetSourceName ("arm default unwind plan");
    // ...
}
```

- This info is used to locate the values to be stored into the respective registers, and for unwinding the stack
Expression Evaluation

• Without implementing any code, we already have simple expression evaluation for MSP430

```
(lldb) expr y
(int) $4 = 2200
(lldb) expr &y
(int *) $5 = 0x03e6
(lldb) expr y = y * 10
(int) $6 = 22000
(lldb) memory read 0x3e6 -f d -s 2
0x000003e6: 22000
```

• LLDB uses Clang for parsing, and performs IR interpretation

• For expression evaluation to work properly, please use the latest MSP430 compiler from TI
  
  ▶ www.ti.com/tool/msp430-gcc-opensource

• Future work for MSP430 would be to support calling functions existing in the binary from IR, by implementing `PrepareTrivialCall()` in the ABI
Expression Evaluation - Behind the scenes

```c
void __lldb_expr(void *__lldb_arg)
{
    y = y * 10;
}
```

target datalayout = "e-m:e-p:16:16-i32:16:32-a:16-n8:16"
target triple = "msp430--"

define void @_Z12$__lldb_exprPv"(i8* %"$__lldb_arg") #0 {
entry:
%0 = getelementptr i8, i8* %"$__lldb_arg", i32 8
%1 = bitcast i8* %0 to i16**
%2 = getelementptr i8, i8* %"$__lldb_arg", i32 0
%3 = bitcast i8* %2 to i16**
%"$__lldb_arg.addr" = alloca i8*, align 2, !clang.decl.ptr !5
store i8* %"$__lldb_arg", i8** %"$__lldb_arg.addr", align 2
%guard.uninitialized = icmp eq i8 0, 0
br i1 %guard.uninitialized, label %init.check, label %init.end

init.check:
    ; preds = %entry
    %4 = load i16*, i16** %1, align 2
    %5 = load i16, i16* %4, align 2
    %mul = mul i16 %5, 10
    %6 = load i16*, i16** %1, align 2
    store i16 %mul, i16* %6, align 2
    store i16* %6, i16** %3, align 2
    br label %init.end
// ...}
```
Disassembly

• LLDB leverages the **disassembler from LLVM**

• MSP430 doesn't have a disassembler yet implemented in LLVM

```
(llldb) dis
error: Unable to find Disassembler plug-in for the 'msp430' architecture.
```

• For ARM,

```cpp
DisassemblerLLVMC::DisassemblerLLVMC (...) {
    // ...
    const char *triple = arch.GetTriple().getTriple().c_str();
    ArchSpec thumb_arch(arch);
    if (arch.GetTriple().getArch() == llvm::Triple::arm){
        std::string thumb_arch_name (thumb_arch.GetTriple().getArchName().str());
        // ...
        thumb_arch_name.insert(0, "thumb");
    }
    thumb_arch.GetTriple().setArchName(llvm::StringRef(thumb_arch_name.c_str()));
    // ...
    m_disasm_ap.reset (new LLVMCDisassembler(triple, cpu, features_str.c_str()));
    // ...
    m_alternate_disasm_ap.reset(new LLVMCDisassembler(thumb_triple.c_str()));
}
```

DisassemblerLLVMC.cpp
Single-stepping

- For MSP430, LLDB sends the `s RSP packet` to request mspdebug to step

  Process 1 stopped
  * thread #1: tid = 0x0001, 0xc164 a.out`calc2(y=22000) + 12 at led.c:12,
  stop reason = step in
  frame #0: 0xc164 a.out`calc2(y=22000) + 12 at led.c:12
       9  int calc2(int y)
       10  11    return 10 + y ;-> 12
(lldb) s
< 5> send packet: $s#73
< 1> read packet: +
< 135> read packet: $T0500:66c1;01:ec03;02:0000;03:0000;04:04c0;05:04c0;06:0200;07:0000;08:ffff;09:9886;0a:0100;0b:0302;0c:6e00;0d:9886;0e:0100;0f:0400;#e2
< 1> send packet: +

- `PTRACE_SINGLESTEP` tells the kernel to stop at every instruction

  NativeProcessLinux::SingleStep(lldb::tid_t tid, uint32_t signo){
      //...
      // If hardware single-stepping is not supported, we just do a continue. The breakpoint on the
      // next instruction has been setup in NativeProcessLinux::Resume.
      return PtraceWrapper(SupportHardwareSingleStepping() ? PTRACE_SINGLESTEP : PTRACE_CONT,
                           tid, nullptr, (void*)data);
  }

NativeProcessLinux.cpp
Debugging Tips
Debugging Tips

LLDB logs provide detailed information

- `log enable <log-channel><log-category>`
- There are many channels and categories available

```
(lldb) log list
Logging categories for 'gdb-remote':
  packets - log gdb remote packets
  memory - log memory reads and writes
Logging categories for 'lldb':
  break - log breakpoints
  dyld - log shared library related activities
  expr - log expressions
```

- `log enable gdb-remote packets`
  - Shows all RSP communication, very useful for debugging
  - Can separate lldb and lldb-server

- `log enable lldb unwind`
  - useful for tracing unwinding problems
Debugging Tips

Sending a single RSP packet

- **process plugin packet send**
- Very useful to isolate lldb and lldb-server

```plaintext
(lldb) process plugin packet send g
packet: g
response: 54c1ee030000000004c004c0010000000ffffff9886010003026e00988601000400

(lldb) process plugin packet send ?
packet: ?
response: T0500:54c1;01:ee03;02:0000;03:0000;04:04c0;05:04c0;06:0100;07:0000;08:ffff;09:9886;0a:0100;0b:0302;0c:6e00;0d:9886;0e:0100;0f:0400;
```

Enabling lldb-server logs

```plaintext
$ lldb-server g --log-file /dev/tty --log-channels "lldb all:gdb-remote all"
localhost:1234 ~/arm_binary.out
```
Debugging Tips

• Make sure both LLDB and executable are built in **debug mode**
• Use dwarfdump to **verify DWARF** generated for the executable is proper
  ▶ For MSP430, due to LLDB parsing DWARF wrongly, couldn't see source file/line numbers
• Use **image dump sections** command to make sure sections have been loaded properly
• Use **objdump/readelf** to
  ▶ Check disassembly
  ▶ Verify ELF header
• Use **strace** for debugging signals
• Use **GDB** to compare debugging sessions
  ▶ For MSP430, we used **msp430-gdb** for reference
MSP430 Quick Recap
MSP430 Quick Recap

Steps for implementing support for MSP430 in LLDB:

▷ msp430 triple to LLDB

▷ Describe msp430 registers in a python file

▷ Fix DWARF parsing errors in LLDB

▷ Added msp430 breakpoint opcode to LLDB

▷ Implemented msp430 ABI

Very useful links:

• binutils - www.ti.com/tool/msp430-gcc-opensource

• mspdebug - dlbeer.co.nz/mspdebug/
MSP430 Quick Recap

$ ./mspdebug sim
MSPDebug version 0.23 - debugging tool for MSP430 MCUs
(mspdebug) $ prog ~/led.elf
Erasing...
Programming...
Writing 2 bytes at fffe [section: __reset_vector]...
Writing 16 bytes at c000 [section: .rodata]...
Writing 804 bytes at c010 [section: .text]...
Writing 4 bytes at c334 [section: .data]...
Done, 826 bytes total
(mspdebug) $ gdb
Bound to port 2000. Now waiting for connection...

(lldb) file /media/andrzej/Build/msp430/led.elf
Current executable set to '/media/andrzej/Build/msp430/led.elf' (msp430).
(lldb) settings set plugin.process.gdb-remote.target-definition-file msp430_target_definition.py
(lldb) b main
Breakpoint 1: where = led.elf`main + 6 at led.c:5, address = 0xf844
(lldb) gdb-remote 2000
Process 1 stopped
* thread #1: tid = 0x0001, 0xf844 led.elf`main + 6 at led.c:5, stop reason = breakpoint 1.1
    frame #0: 0xf844 led.elf`main + 6 at led.c:5
        int main(void)
        {
            WDTCTL = WDTPW + WDTOLD;
Homework

- Investigating the ABI and disassembly
- Upstreaming
- Reverse engineering locks with MSP430 mcus
  - microcorruption.com
- Write a debugger for AVR
  - Yet another simple RISC architecture, used in Arduino
    - avr-eclipse.sourceforge.net/wiki/index.php/Debugging
- Check out our code from GitHub:
  - github.com/codeplaysoftware/lldb-msp430
Final hint

Wondering how to write a debugger for your architecture?

Get in touch with Codeplay!

⇒ deepak@codeplay.com

⇒ andrzej@codeplay.com

⇒ www.codeplay.com