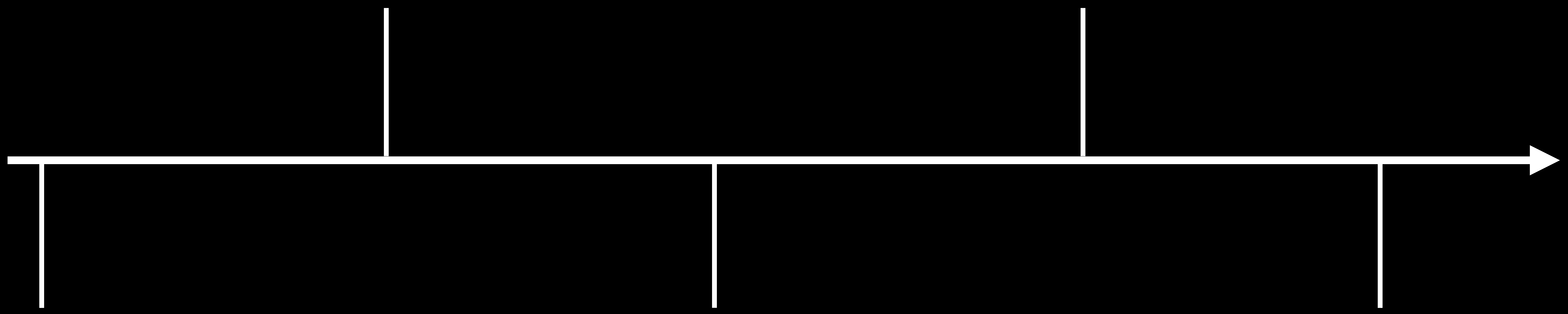
WebAssembly Debugging with LLDB

Jonas Devlieghere

FOSDEM '26 LLVM Dev Room

History



2018
Akio Yasui
creates WasmKit
(WAKit)

Max Desiatov takes
over maintainership
2022

2023
Yuta Saito joins and
WasmKit achieves 100%
SpecTest coverage

WasmKit is used
in CI for Swift 6.0
2024

2025
WebAssembly is officially
supported in Swift 6.2 and
WasmKit ships with the toolchain

https://github.com/swiftlang

https://github.com/swiftwasm

https://swiftwasm.org

Goal

- First class debugging experience for Swift compiled to WebAssembly
  - Source-level debugging (breakpoints, stepping, variables)
  - Swift language support (e.g. *Reflection Metadata*)

- How?
  - Teach WebAssembly tools about Swift
  - Teach LLDB about WebAssembly

Approaches to Wasm Debugging

Wasmtime

Code is JIT'ed in runtime
LLDB debugs the runtime

- ✓ Mature tooling can be used unmodified
- ! Mixed runtime and user code

Chrome Dev Tools

Fully browser based
Uses LLDB to parse DWARF

- ✓ Seamless experience with JavaScript
- ! Needs language support in Chrome

WAMR

Provides GDB remote stub
that LLDB can connect to

- ✓ Native LLDB experience
- ! Requires extensions in LLDB

Approaches to Wasm Debugging

Wasmtime

Code is JIT'ed in runtime
LLDB debugs the runtime

- ✓ Mature tooling can be used unmodified
- ! Mixed runtime and user code

Chrome Dev Tools

Fully browser based
Uses LLDB to parse DWARF

- ✓ Seamless experience with JavaScript
- ! Needs language support in Chrome

WAMR

Provides GDB remote stub
that LLDB can connect to

- ✓ Native LLDB experience
- ! Requires extensions in LLDB

Approaches to Wasm Debugging

Wasmtime

Code is JIT'ed in runtime
LLDB debugs the runtime

- ✓ Mature tooling can be used unmodified
- ! Mixed runtime and user code

Chrome Dev Tools

Fully browser based
Uses LLDB to parse DWARF

- ✓ Seamless experience with JavaScript
- ! Needs language support in Chrome

WAMR

Provides GDB remote stub that LLDB can connect to

- ✓ Native LLDB experience
- ! Requires extensions in LLDB

Approaches to Wasm Debugging

Wasmtime

Code is JIT'ed in runtime
LLDB debugs the runtime

- ✓ Mature tooling can be used unmodified
- ! Mixed runtime and user code

Chrome Dev Tools

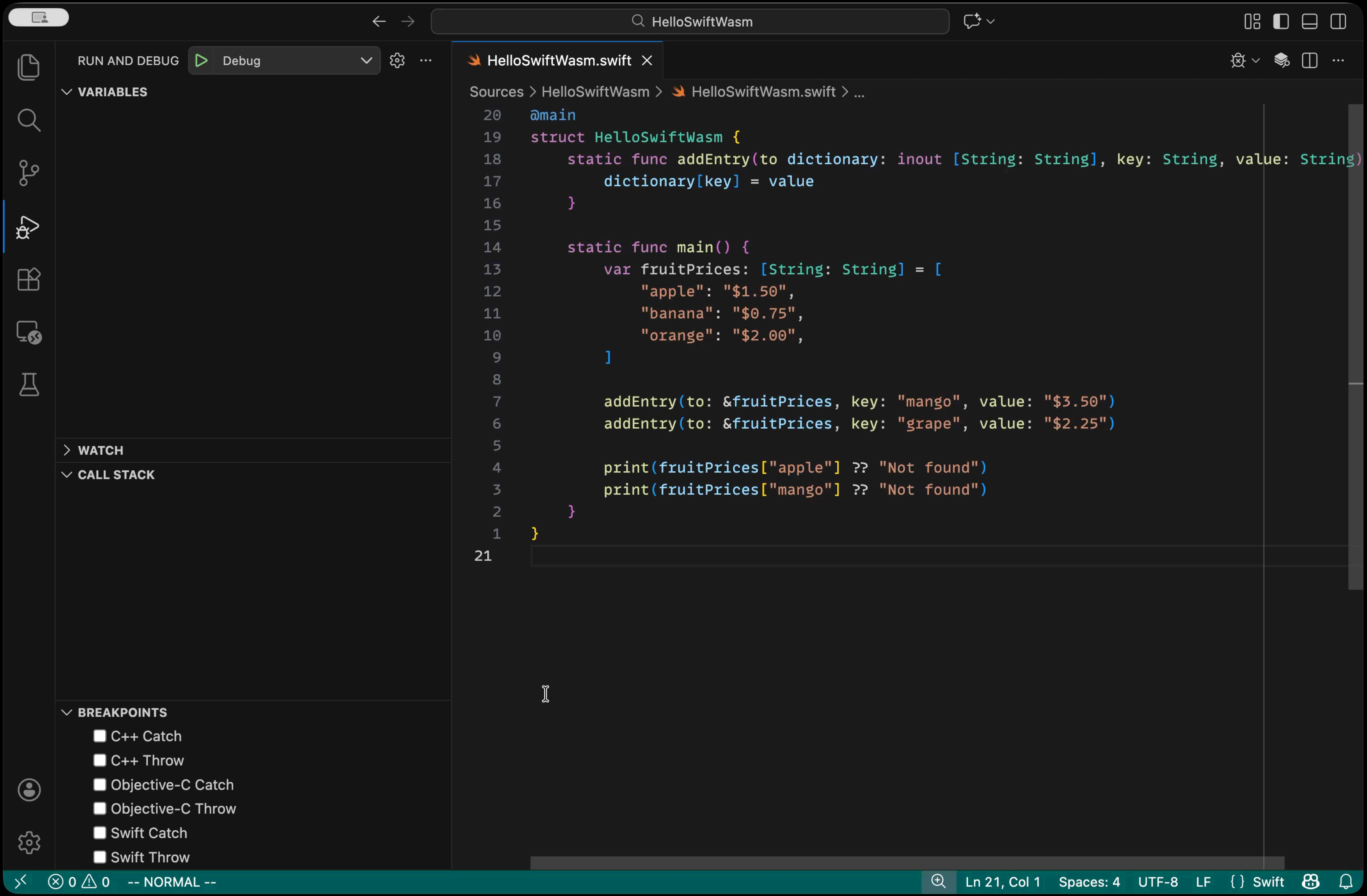
Fully browser based
Uses LLDB to parse DWARF

- ✓ Seamless experience with JavaScript
- ! Needs language support in Chrome

WAMR

Provides GDB remote stub
that LLDB can connect to

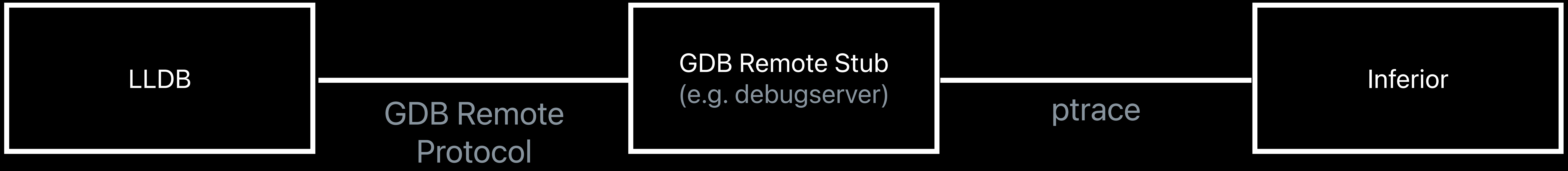
- ✓ Native LLDB experience
- ! Requires extensions in LLDB



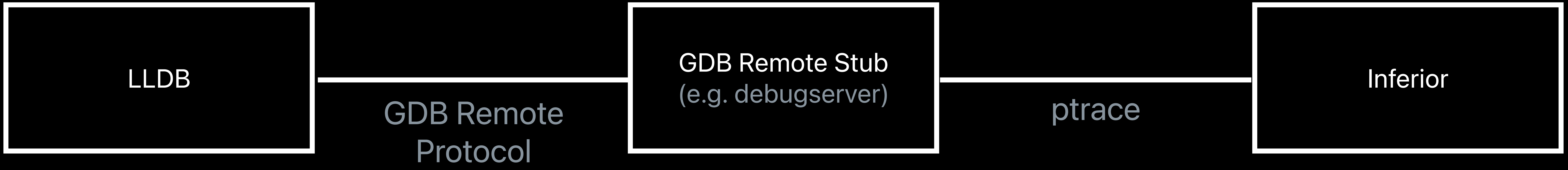
```swift
@main
struct HelloSwiftWasm {
    static func addEntry(to dictionary: inout [String: String], key: String, value: String)
        dictionary[key] = value
    }

    static func main() {
        var fruitPrices: [String: String] = [
            "apple": "$1.50",
            "banana": "$0.75",
            "orange": "$2.00",
        ]

        addEntry(to: &fruitPrices, key: "mango", value: "$3.50")
        addEntry(to: &fruitPrices, key: "grape", value: "$2.25")

        print(fruitPrices["apple"] ?? "Not found")
        print(fruitPrices["mango"] ?? "Not found")
    }
}

```

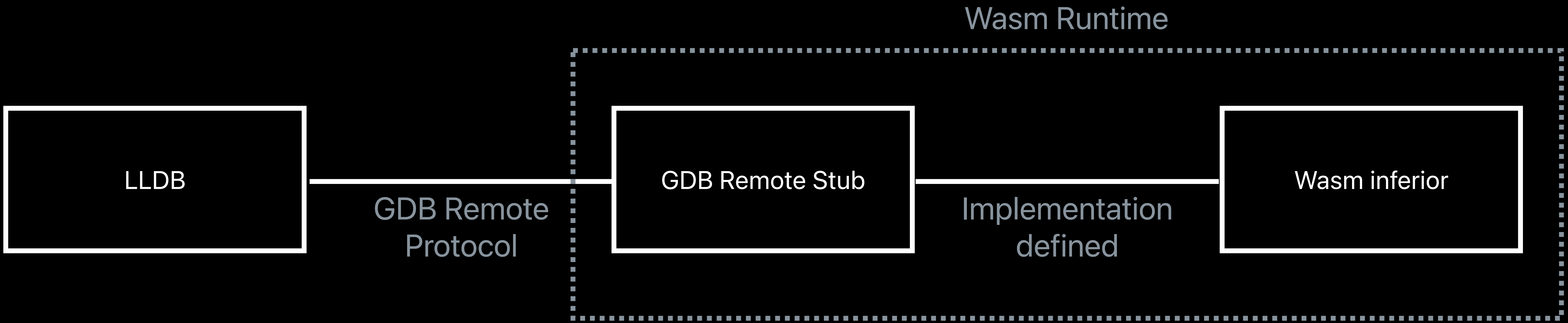
☐ C++ Catch
☐ C++ Throw
☐ Objective-C Catch
☐ Objective-C Throw
☐ Swift Catch
☐ Swift Throw

Architecture



Architecture





Existing WebAssembly Support

- Upstream
  - Loading binaries
  - Creating types from DWARF

- Downstream
  - Patches in the WAMR repository
  - Unmerged PRs from Paolo Severini

Object Files

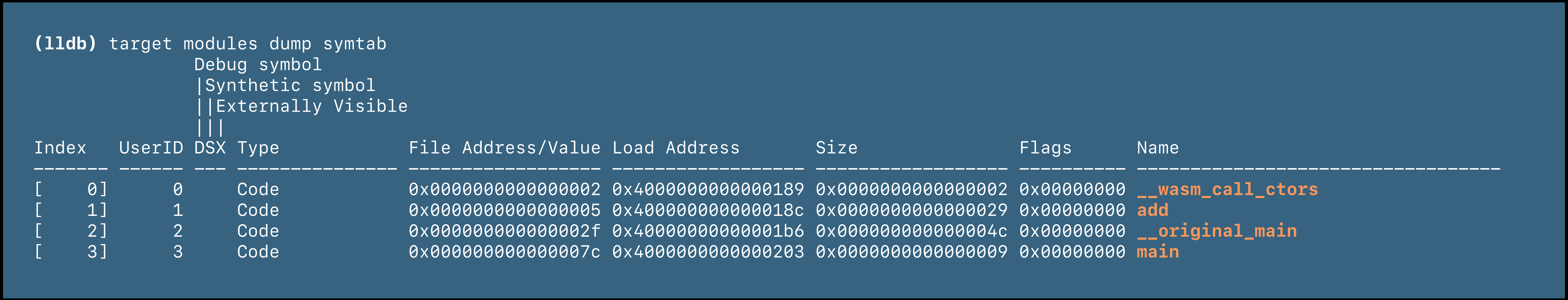
- Replace ad-hoc section parsing in `ObjectFileWasm`
  - Support standard (code, data) and custom sections (DWARF, Swift)
  - Mini Wasm interpreter for *init expression*

```
(lldb) target modules dump sections
SectID             Type          Load Address                             Perm File Off.  File Size  Flags      Section Name
------------------ ------------- ---------------------------------------  ---- ---------- ---------- ---------- ----------------------------
0x0000000000000001 code          [0x4000000000000187-0x400000000000020c)  ---  0x00000187 0x00000085 0x00000000 simple.wasm.code
0x000000000000000f dwarf-abbrev  [0x4000000000000239-0x40000000000002e2)  ---  0x00000239 0x000000a9 0x00000000 simple.wasm..debug_abbrev
0x0000000000000014 dwarf-info    [0x40000000000002f1-0x40000000000003c6)  ---  0x000002f1 0x000000d5 0x00000000 simple.wasm..debug_info
0x000000000000001b dwarf-ranges  [0x40000000000003d6-0x40000000000003ee)  ---  0x000003d6 0x00000018 0x00000000 simple.wasm..debug_ranges
0x000000000000001c dwarf-str     [0x40000000000003fc-0x40000000000004e3)  ---  0x000003fc 0x000000e7 0x00000000 simple.wasm..debug_str
0x0000000000000015 dwarf-line    [0x40000000000004f1-0x4000000000000557)  ---  0x000004f1 0x00000066 0x00000000 simple.wasm..debug_line
0x0000000000000040 wasm-name     [0x400000000000055e-0x40000000000005c5)  ---  0x0000055e 0x00000067 0x00000000 simple.wasm.name
0x0000000000000100 data          [0x4000000000000215-0x400000000000021e)  ---  0x00000215 0x00000009 0x00000000 simple.wasm..rodata
0x0000000000000200 data          [0x4000000000000224-0x4000000000000228)  ---  0x00000224 0x00000004 0x00000000 simple.wasm..data
```

https://github.com/llvm/llvm-project/pull/153634

Symbol Table

- Symbolication and breakpoints
  - Function offset and size are stored in **function section**
  - Function names encoded in the **names section**



```
(lldb) target modules dump symtab
               Debug symbol
               |Synthetic symbol
               ||Externally Visible
               |||
Index   UserID DSX Type            File Address/Value Load Address       Size               Flags      Name
------- ------ --- --------------- ------------------ ------------------ ------------------ ---------- ----------------------------------
[    0]      0     Code            0x0000000000000002 0x4000000000000189 0x0000000000000002 0x00000000 __wasm_call_ctors
[    1]      1     Code            0x0000000000000005 0x400000000000018c 0x0000000000000029 0x00000000 add
[    2]      2     Code            0x000000000000002f 0x40000000000001b6 0x000000000000004c 0x00000000 __original_main
[    3]      3     Code            0x000000000000007c 0x4000000000000203 0x0000000000000009 0x00000000 main
```

https://github.com/llvm/llvm-project/pull/153093

Backtraces

- New ProcessWasm plugin
  - No stack memory, registers or ABI (prior to Wasm EH)
  - LLDB has to rely on the runtime for unwinding
  - GDB remote extension: qWasmCallStack

```
(lldb) bt
* thread #1, name = 'nobody', stop reason = breakpoint 2.1
  * #0: 0x40000000000001a8 simple.wasm`add(a=1, b=2) + 28 at /path/to/simple.c:4
    #1: 0x40000000000001f1 simple.wasm`main + 59 at /path/to/simple.c:10
    #2: 0x400000000000020a simple.wasm`main + 7
```

https://github.com/llvm/llvm-project/pull/150143

Variables

- **Location descriptions** in DWARF
  - Empty: location unavailable
  - Implicit: location unavailable but value is known (value)
  - Memory: location in memory (address)
  - Register: location in memory (register name)

```
0x00000062:     DW_TAG_formal_parameter
                  DW_AT_location          (DW_OP_reg7)
                  DW_AT_name              ("a")      └► DWARF register 7
                  DW_AT_decl_file         ("/tmp/simple.c")
                  DW_AT_decl_line         (3)
                  DW_AT_type              (0x0000009f "int")
```

Register Locations

- Wasm uses virtual registers in DWARF
  - Globals (qWasmGlobal)
  - Locals (qWasmLocal)
  - Operand stack (qWasmStackValue)

```
0x00000062:     DW_TAG_formal_parameter
                  DW_AT_location        (DW_OP_WASM_location 0x0 0x2, DW_OP_stack_value)
                  DW_AT_name            ("a")                  │   └► argument: 2 (index)
                  DW_AT_decl_file       ("/tmp/simple.c")      └────► location: local (qWasmLocal)
                  DW_AT_decl_line       (3)
                  DW_AT_type            (0x0000009f "int")
```

https://github.com/llvm/llvm-project/pull/151010

Memory Locations

- Separate address spaces for *code* and *memory*
  - wasm32: encoded in the top 32 bits of a 64-bit address
  - wasm64: unsupported (until we have address space support)

```
struct wasm_addr_t {
  uint64_t offset    : 32;
  uint64_t module_id : 30;
  uint64_t type      : 2;

  wasm_addr_t(lldb::addr_t addr)
      : offset(addr & 0x00000000ffffffff),
        module_id((addr & 0x00ffffff00000000) >> 32), type(addr >> 62) {}
}
```

https://github.com/llvm/llvm-project/pull/150143

Swift Support

- Teach `libSwiftReflection` about Wasm
  - Reflection metadata is generated by the compiler
  - Consumed by the runtime & the debugger
  - Stored in custom section: reimplement section parsing

```
(lldb) v dictionary
([String : String]) dictionary = 4 key/value pairs {
  [0] = (key = "apple", value = "$1.50")
  [1] = (key = "banana", value = "$0.75")
  [2] = (key = "mango", value = "$3.50")
}
```

https://github.com/swiftlang/swift/pull/83923

Platform Plugin

- New `PlatformWasm`
  - Automatically selected for targets with a WebAssembly triple
  - Launches binaries under the runtime and connects to GDB stub
  - Your choice of runtime, configurable in `~/.lldbinit`

https://github.com/llvm/llvm-project/pull/171507

First Class Debugging for WebAssembly

- Any **language** supported by LLDB
  - Swift (swiftc)
  - C, C++ (clang, emscripten)

- Any **runtime** implementing the protocol
  - WebAssembly Micro Runtime (WAMR)
  - WasmKit
  - JavaScriptCore (WebKit)

https://lldb.llvm.org/resources/lldbgdbremote.html#wasm-packets

What's next

- Extend the **LLDB test suite**
  - Compile test binaries to WebAssembly
  - Run and debug them under WasmKit
  - Uncover bugs LLDB and GDB stubs
- Support more Swift language features
- Support address spaces for Wasm64