THE LINUX FOUNDATION OPEN SOURCE SUMMIT JAPAN

AUTOMOTIVE LINUX SUMMIT

The cool features of WebAssembly Micro Runtime (WAMR) for IoT and embedded

Xin Wang

#ossummit ##autolinux xin.wang@intel.com

Agenda

- Why is WebAssembly great for IoT
- Project history and status
- Features
- Interpreters
- Ahead-of-Time compiler and loader
- Execution-in-Place (XIP)
- Debugger
- Application framework
- Quick start

Why is WebAssembly great for IoT

- Compilation target for multiple languages (C/C++/Rust/Go/AS…)
- Isolation
- Lightweight and small footprint
- High performance
- Portability
- Standardized API for host embedding runtime
- SIMD, multi-threading

- One binary for multiple architectures
- Offloading among cloud, edge and nodes
- Independent application development and deployment, accelerated innovations
- Improved system robustness and security
- Enable 3rd party apps
- Better time determinism for control automation

Project background

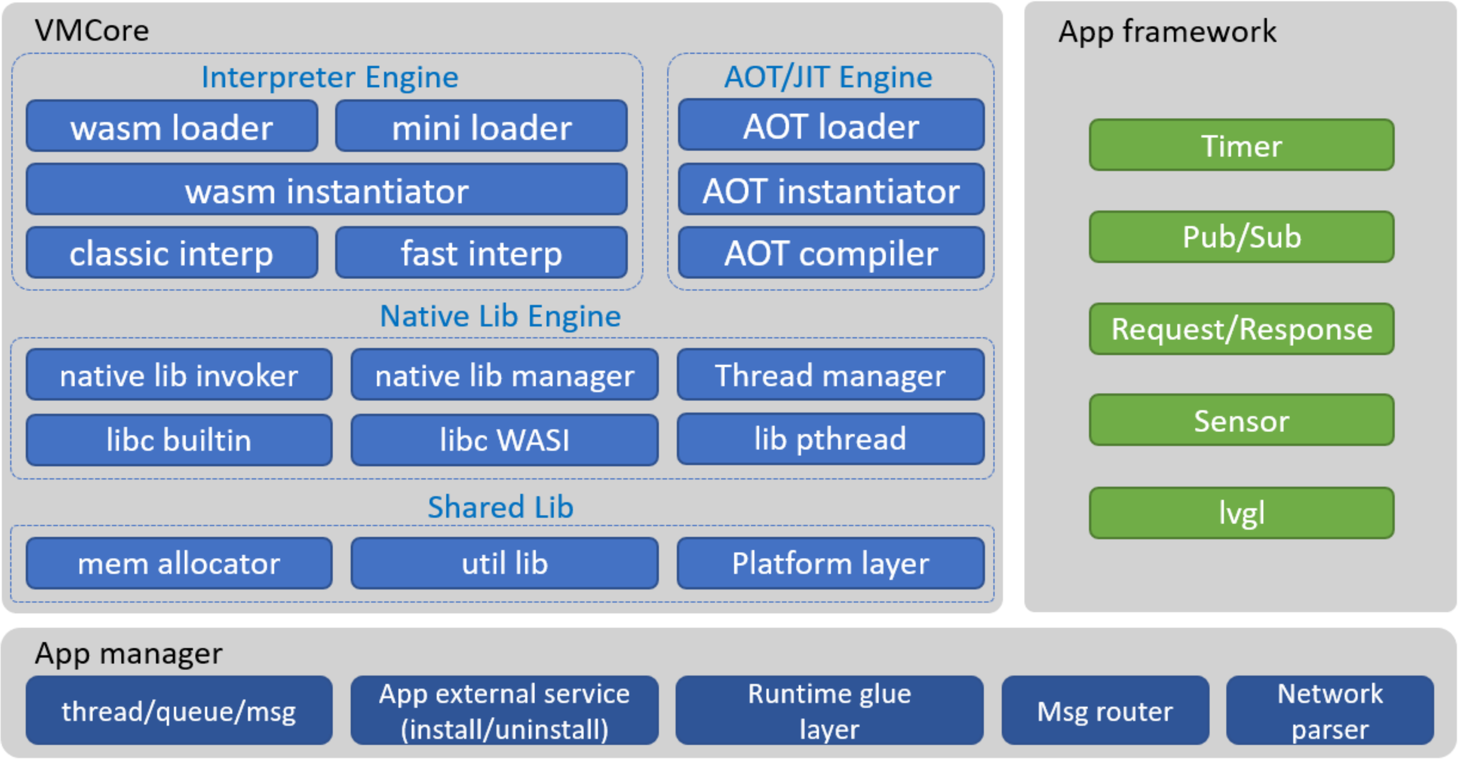
- Intel open sourced the WAMR project in May 2019
- Targeting small footprint, high performance and great adaptability from embedded to cloud
- Transferred to the Bytecode Alliance (BA) in November 2019 as one of the initiating projects
- Established open governance model and TSC in 2021
- Active community and broad adoptions by commercial and open-source projects
  - Smart contract, IoT, service mesh, trusted computing, mini-app

WAMR features

- Full WASM MVP spec support and aggressive post MVP features support
- Support Interpreter (fast and classic), Ahead of Time (AoT) and Just-in-Time (JIT) Compilation
- C based implementation, LLVM based compilation
- Small binary size and memory consumption
  - VMCore – 60K for AoT, 90K for interpreter
  - Down to 3K DRAM for running hello-world
- Near native performance with AOT/JIT, fast interpreter
- AOT module loader, Execution-in-Place (XIP)
- IoT oriented Wasm application framework and API, remote app management
- Support libc-wasi, libc-buitlin, multi-thread, multi-module, 128-bit SIMD, wasm-c-api, source debugging, app-manager, etc.

- CPU Arch support:
  - X86-64, X86-32
  - ARM, THUMB, AARCH64
  - MIPS, ARC
  - XTENSA, RISC-V
- Platform support:
  - Linux, SGX (Linux)
  - Windows, MacOS, Android
  - Zephyr, AliOS Things
  - Vxworks, RT-Thread
  - OpenRTOS

WAMR software architecture



WAMR supports fast interpreter and classic interpreter

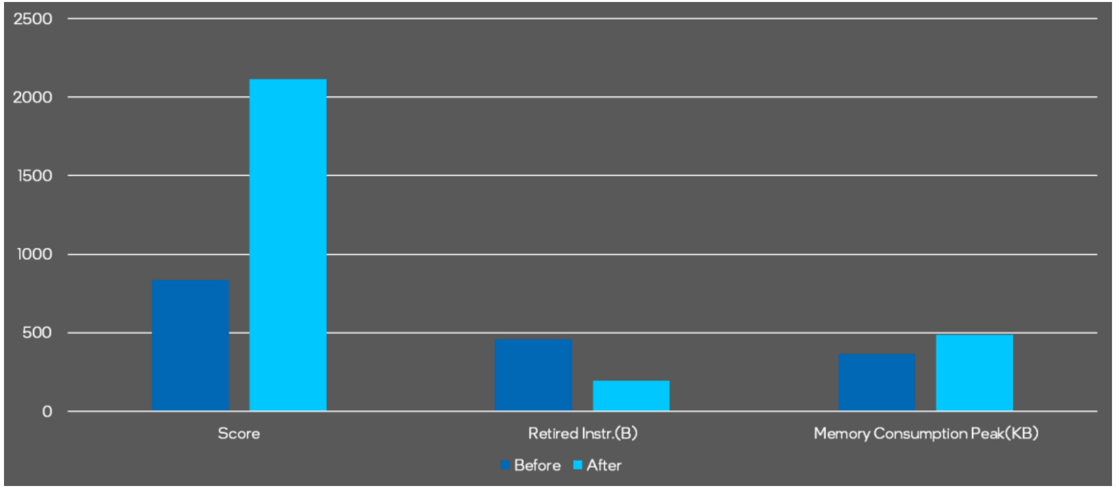
Stack based (classic)

- Smaller average bytecode
- Stack operations overhead
- Smaller memory usage

Register file based (fast)

- Extended bytecode in memory
- Pre-compilation during wasm loading
- Larger average bytecode
- Efficient execution
- More memory usage

Measurement on CoreMark



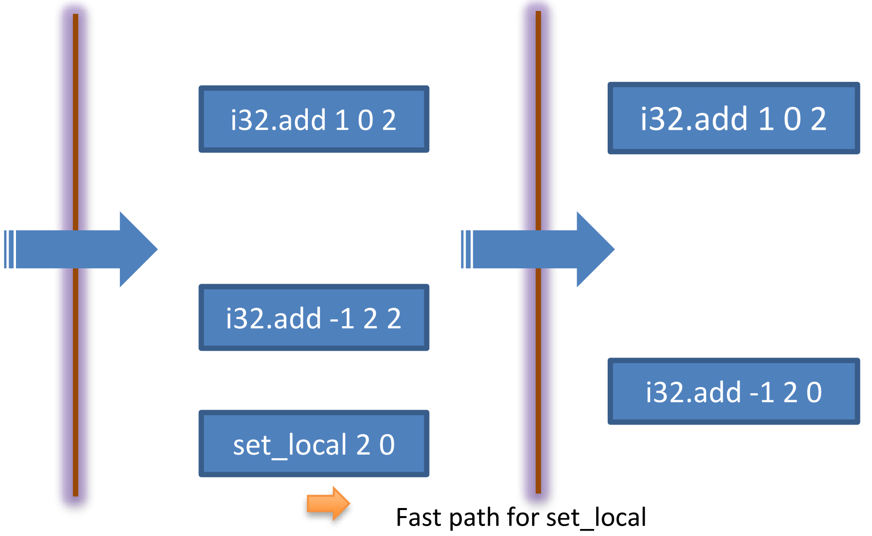
Pre-compiling Wasm to WAMR extended bytecode

- Define spaces layout

local.get 0	const:0, local:1, dynamic:0
local.get 1	const:0, local:2, dynamic:0
i32.add	const:0, local:2, dynamic:1
i32.const 1	const:1, local:2, dynamic:1
i32.add	const:1, local:2, dynamic:1
local.set 0	const:1, local:2, dynamic:1

slot index： -1 0 1 2

- Extend new bytecode



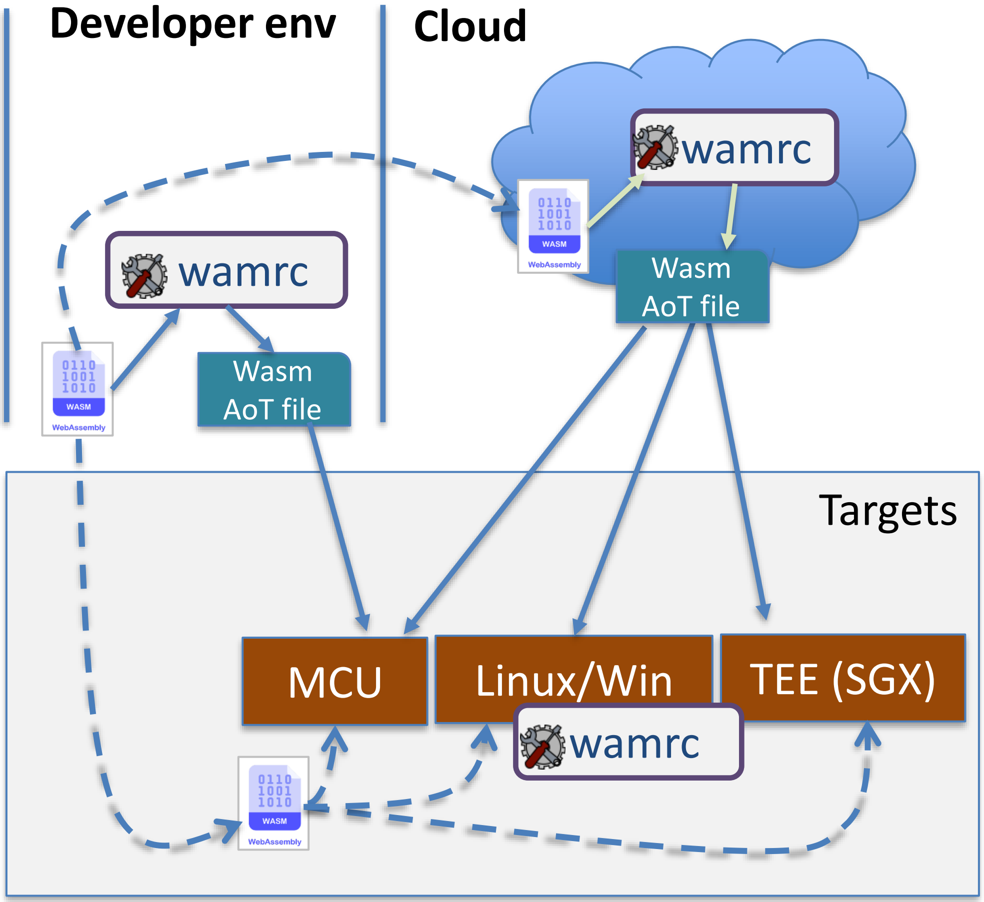
1 const space 1 local space 1 dynamic space

Ahead-of-Time Compilation and loader

WAMR supports AoT compiler “wamrc”
- Compiles Wasm module into AoT module

WAMR module loader enables AoT for various target environments
- Linux, Windows
- TEE (Intel SGX)
- MCU

Multiple paths for deploying AoT modules
- Compiling on target
- Through cloud distribution
- Through software Installation package



Execution-in-Place(XIP) for AoT module

XIP supports executing AoT compiled module from flash

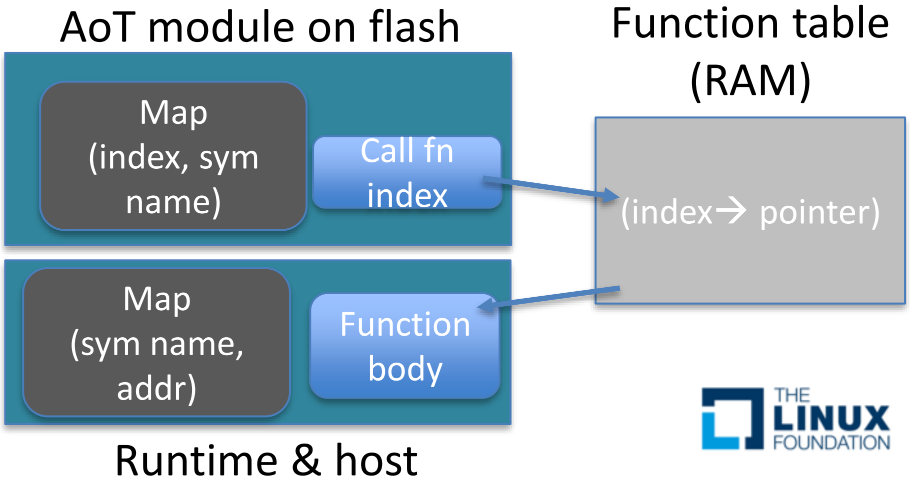
– Reduce RAM usage by loading AoT modules

wamrc supports parameters for generating XIP enabled module

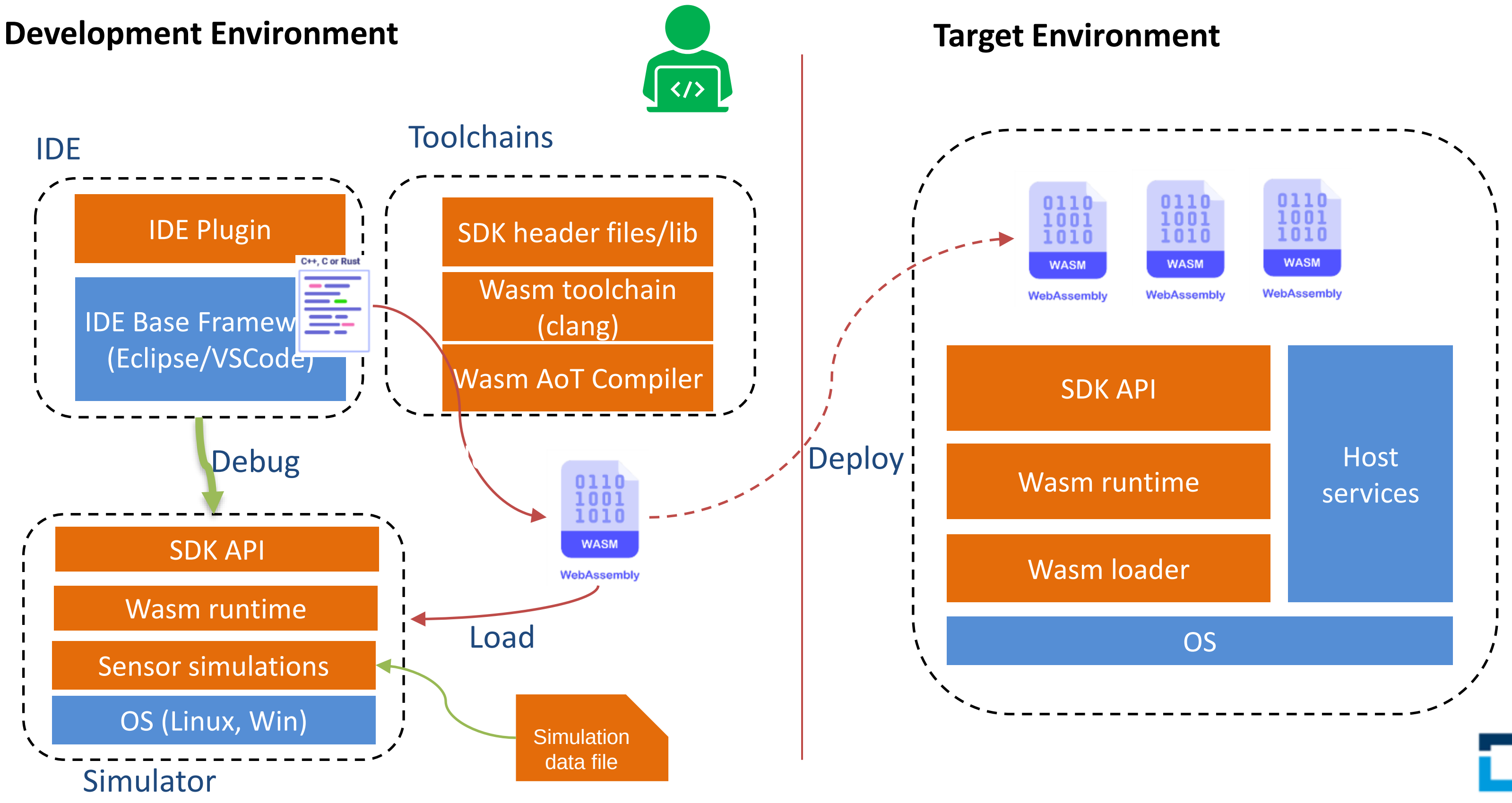
```
wamrc --enable-indirect-mode --disable-llvm-intrisics
```

Avoid patching the module for calling functions in host

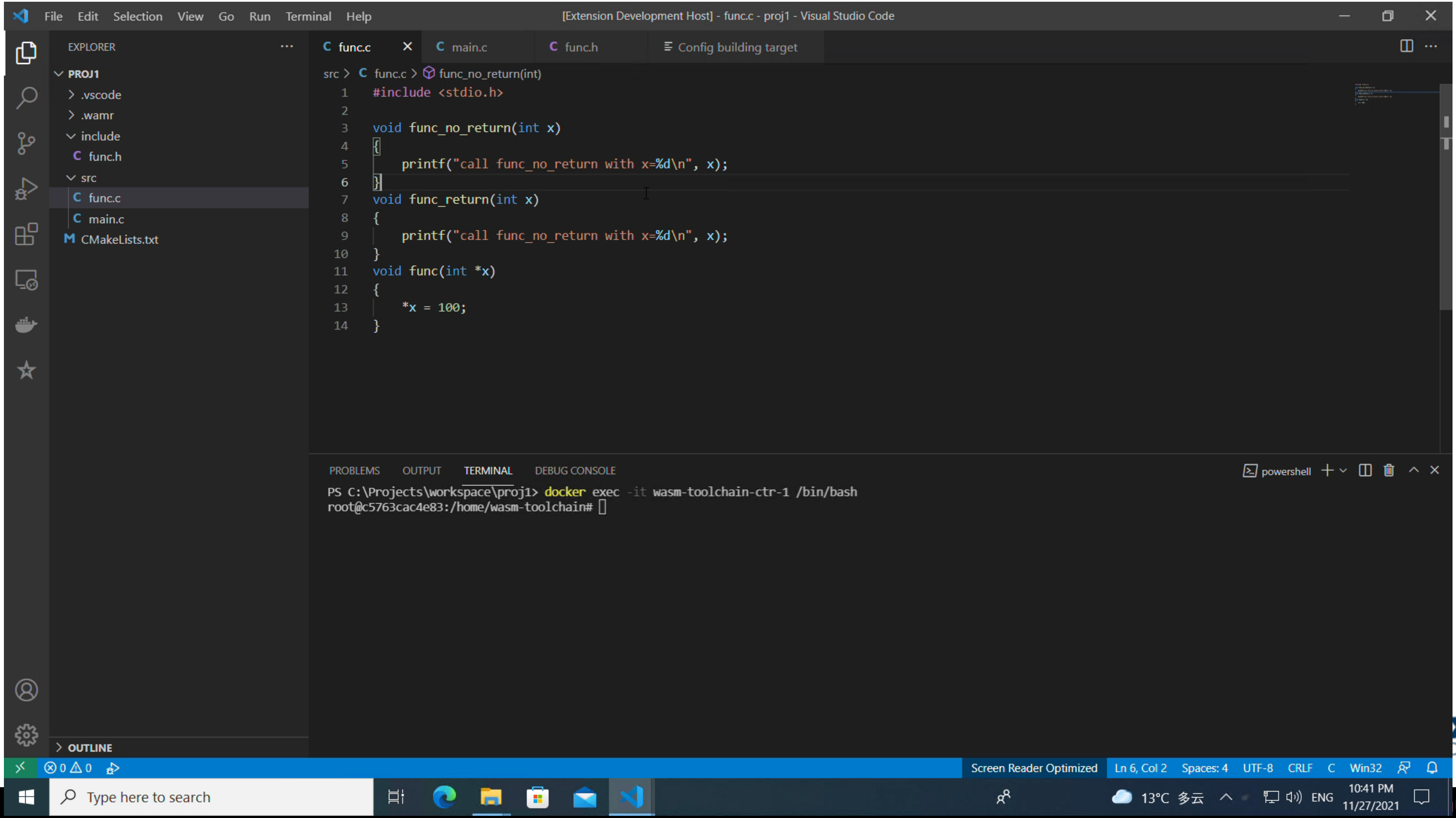
- A map (fn index, sym name) in module
- A map (sym name, fn addr) in host
- Build function table(fn index, fn addr) in RAM during loading module
- AoT module call function through index in the function table



Wasm development working flow

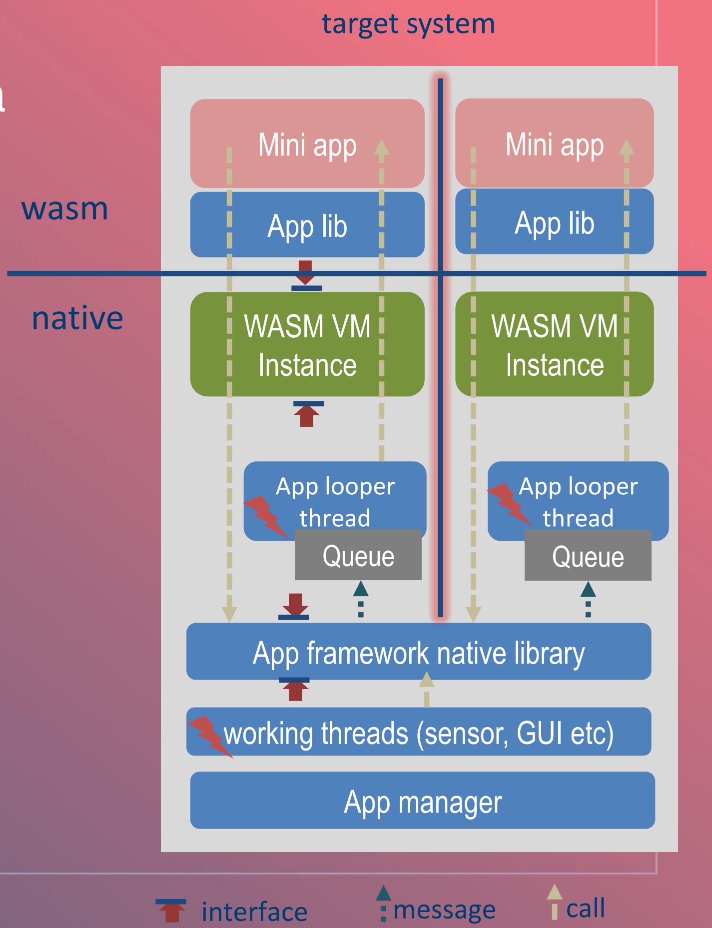


Source debugging demo



Build a Wasm app framework

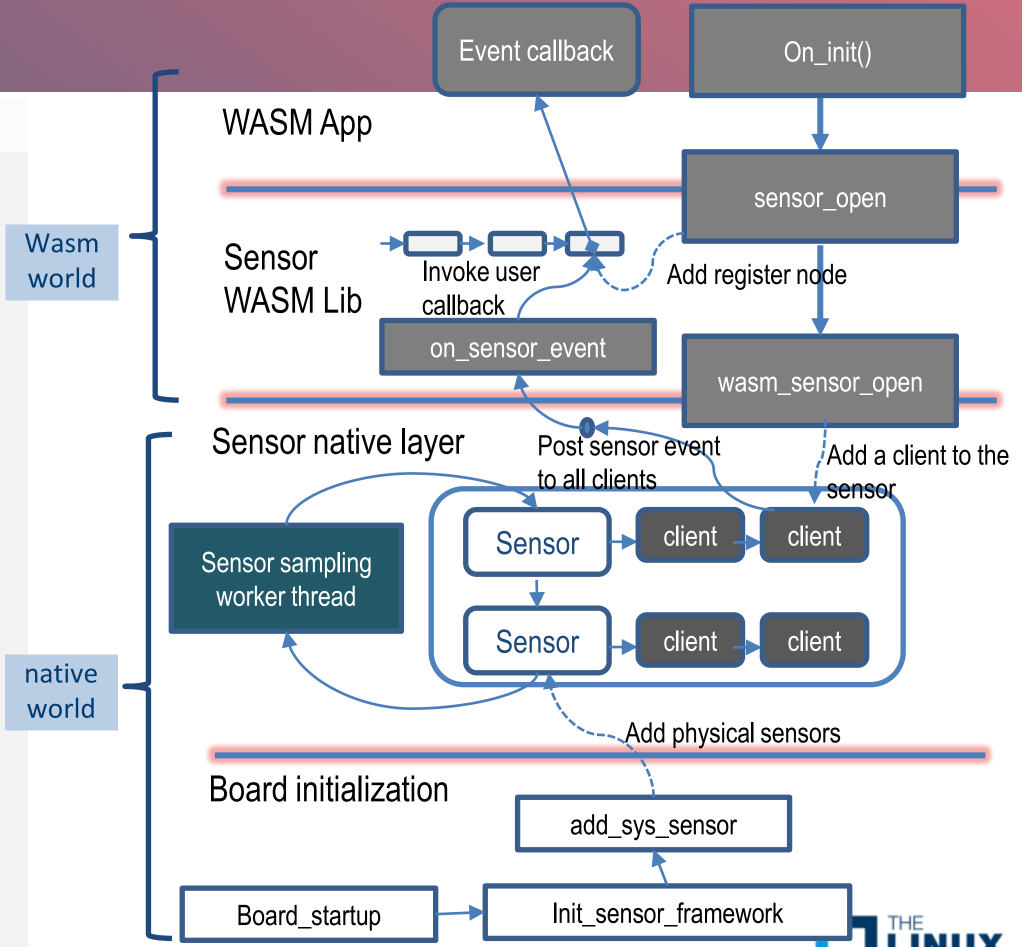
- WAMR VMCore provides API for building a customized app framework
  - Native world calls Wasm functions
  - Wasm calls native APIs
- WAMR provides an asynchronized Wasm app programming model
  - Multi apps, Queue and messaging
  - Every WASM app has its own sandbox and thread
  - Event driven programming model
  - On_Init(), On_Destory() callbacks
- Support remote application management
  - Install, start, uninstall, etc



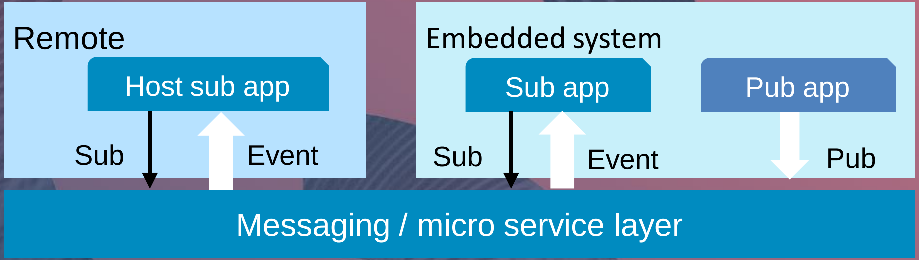
Sensor API and sample

${wamr_root}/samples/simple/wasm-apps/sensor.c

```c
#include "wasm_app.h"

/* Sensor event callback*/
Void sensor_event_handler(sensor_t sensor,
          attr_container_t *event,
          void *user_data) {
  printf("### app get sensor event\n");
  attr_container_dump(event);
}

Void on_init()
{
  sensor_t sensor;

/* open a sensor */
sensor = sensor_open("sensor_test", 0, sensor_event_handler, NULL);

  /* config the sensor */
  sensor_config(sensor, 2000, 0, 0);
}

void on_destroy() {
}
```



Simple sample – pub/sub WASM app



App as Subscriber

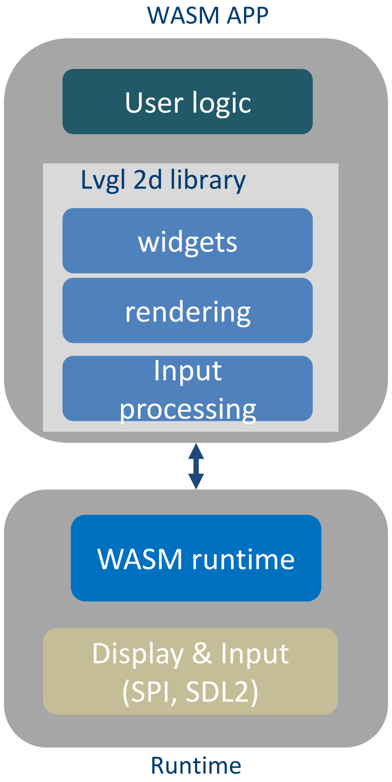
```
void on_init()
{
  api_subscribe_event (" alert/overheat", event1_handler);
}

void on_destroy()
{
}

void event1_handler(request_t *request)
{
}
```

App as Publisher

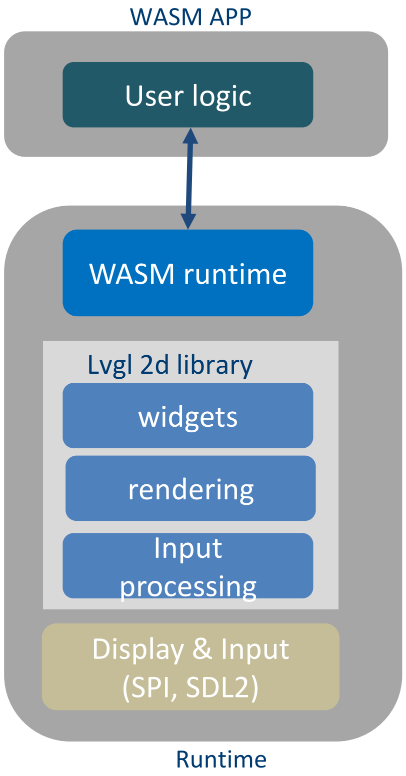
```
/* Timer callback */
void timer1_update(user_timer_t timer)
{
  attr_container_t *event;
  printf("Timer update %d\n", num++);

  event = attr_container_create("event");
  attr_container_set_string(&event,
         "warning",
         "temperature is over high");

  api_publish_event("alert/overheat",
         FMT_ATTR_CONTAINER,
         event,
         attr_container_get_serialize_length(event));

  attr_container_destroy(event);
}
void on_init()
{
   user_timer_t timer;
   timer = api_timer_create(1000, true, true, timer1_update);
}
```

Graphic User Interface on Wasm

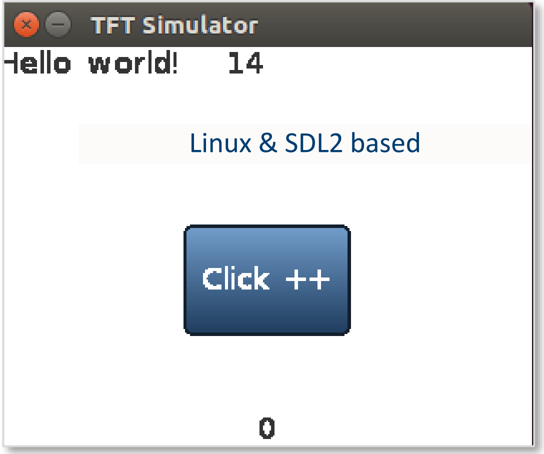
Sample littlevgl: whole lvgl 2d library is compiled to WASM



Sample gui: whole lvgl 2d library is compiled into runtime



The same WASM binary across different hardware and systems



Linux & SDL2 based

ZephyrOS & LCD based

WAMR quick start

- **1. Setup build environment**

  Install Ubuntu 18.04, and execute commands below:

  sudo apt update && sudo apt install git wget build-essential cmake -y
  wget https://github.com/WebAssembly/wasi-sdk/releases/download/wasi-sdk-12/wasi-sdk-12.0-linux.tar.gz
  tar zxvf wasi-sdk-12.0-linux.tar.gz && sudo mv wasi-sdk-12.0 /opt/wasi-sdk

- **2. Download WAMR repo**

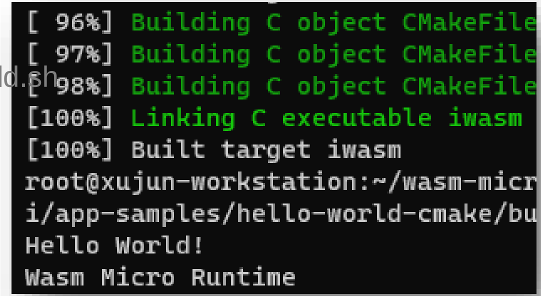
  cd ~ && git clone https://github.com/bytecodealliance/wasm-micro-runtime

- **3. Build the Linux mini product and run sample Wasm module**
  - Build runtime mini-product “iwasm”:

    cd ~/wasm-micro-runtime/product-mini/app-samples/hello-world-cmake && ./build.sh
  - Build sample wasm module:

    cd ~/wasm-micro-runtime/product-mini/platforms/linux
    mkdir build && cd build
    cmake .. && make
  - Run sample wasm module:

    ./iwasm ~/wasm-micro-runtime/product-mini/app-samples/hello-world-cmake/build/hello_world



Use AoT compiler

1. download and build llvm

cd ~/wasm-micro-runtime/wamr-compiler && ./build_llvm.sh #(Note: This may take a long time)

2. build WAMR AoT compiler (wamrc)

mdkir build && cd build
cmake .. && make

3. use wamrc to compile the wasm module

./wamrc -o test.aot ~/wasm-micro-runtime/product-mini/app-samples/hello-world-cmake/build/hello_world

4. execute the AoT module

~/wasm-micro-runtime/product-mini/platforms/linux/build/iwasm test.aot

THE LINUX FOUNDATION

Load Wasm binary and calls into Wasm functions

```
char *buffer, error_buf[128];
 wasm_module_t module;
 wasm_module_inst_t module_inst;
 wasm_function_inst_t func;
 wasm_exec_env_t exec_env;
 uint32 size, stack_size = 8092, heap_size = 8092;

 /* initialize the wasm runtime by default configurations */
 wasm_runtime_init();

 /* read WASM file into a memory buffer */
 buffer = read_wasm_binary_to_buffer(…, &size);

 /* add line below if we want to export native functions to WASM app */
 wasm_runtime_register_natives(...);

/* parse the WASM file from buffer and create a WASM module */
 module = wasm_runtime_load(buffer, size, error_buf, sizeof(error_buf));

 /* create an instance of the WASM module  */
 module_inst = wasm_runtime_instantiate(module, stack_size, heap_size,
                                        error_buf, sizeof(error_buf));
```

```
/* lookup a WASM function by its name
     The function signature can NULL here */
  func = wasm_runtime_lookup_function(module_inst, "fib",
NULL);

/* creat an execution environment to execute the WASM functions
*/
  exec_env = wasm_runtime_create_exec_env(module_inst,
stack_size);
 unit32 argv[2];

  /* arguments are always transferred in 32-bit element */
  argv[0] = 8;

  /* call the WASM function */
  if (wasm_runtime_call_wasm(exec_env, func, 1, argv) ) {
      /* the return value is stored in argv[0] */
      printf("fib function return: %d\n", argv[0]);
  }
  else {
      /* exception is thrown if call fails */
      printf("%s\n", wasm_runtime_get_exception(module_inst));
  }
```

Export native functions to Wasm module

Register the native function pointer, function signature and the symbol name in Wasm for exporting to Wasm

```
#define REG_NATIVE_FUNC(func_name, signature)  \
    { #func_name, func_name##_wrapper, signature, NULL }

static NativeSymbol native_symbols_libc_builtin[] = {
    REG_NATIVE_FUNC(puts, "($)i"),
    REG_NATIVE_FUNC(putchar, "(i)i"),
    REG_NATIVE_FUNC(memcmp, "(**~)i"),
    REG_NATIVE_FUNC(memcpy, "(**~)i"),
    REG_NATIVE_FUNC(memmove, "(**~)i"),
    REG_NATIVE_FUNC(memset, "(*ii)i"),
    REG_NATIVE_FUNC(strchr, "($i)i"),
    REG_NATIVE_FUNC(strcmp, "($$)i"),
    REG_NATIVE_FUNC(strcpy, "(*$)i"),
    ...
};

bool wasm_register_builtin_libc()
{
    int n_native_symbols = sizeof(native_symbols_libc_builtin) /
        sizeof(native_symbols_libc_builtin[0]);
    return wasm_runtime_register_natives("env",
                                         native_symbols_libc_builtin,
                                         n_native_symbols);
}
```

```
{
    "memcpy",
    memcpy_wrapper,
    "(**~)i",
    NULL
}
```

Registration element

core\iwasm\libraries\libc-builtin\libc_builtin_wrapper.c

Automatic pointer conversation between Wasm sandbox and native with signature letter "*" and "~".

```
static int32
memcmp_wrapper(wasm_exec_env_t exec_env,
               const void *s1, const void *s2, uint32 size)
{
    wasm_module_inst_t module_inst = get_module_inst(exec_env);

    /* s2 has been checked by runtime */
    if (!validate_native_addr((void*)s1, size))
        return 0;

    return memcmp(s1, s2, size);
}
```

Automatic string pointer conversation between Wasm sandbox and native with signature letter "$"

```
static int32
strcmp_wrapper(wasm_exec_env_t exec_env,
               const char *s1, const char *s2)
{
    /* s1 and s2 have been checked by runtime */
    return strcmp(s1, s2);
}
```

Reference links

- Build source code into Wasm binary
- Embed WAMR
- WAMR header file
- Build WAMR
- Export native API to Wasm
- The basic sample

Try WAMR out:

https://github.com/bytecodealliance/wasm-micro-runtime

THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
JAPAN

AUTOMOTIVE
LINUX SUMMIT

THE
LINUX
FOUNDATION