SSS Agenda for WAMR TSC

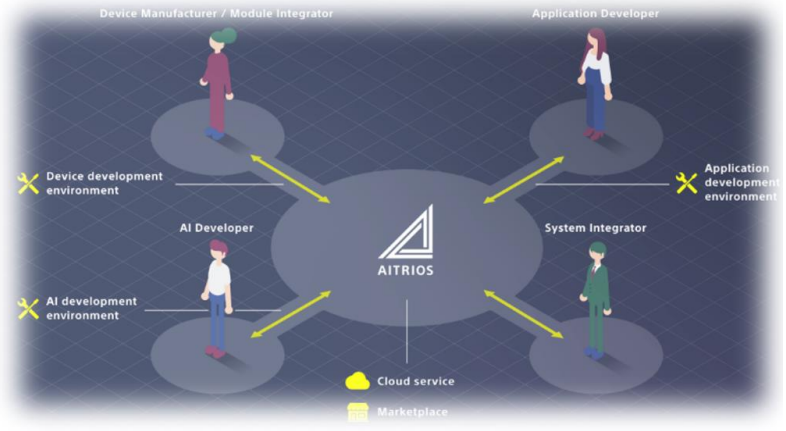
Yan Dongsheng

AITRIOS, Edge Device and ESP32

https://www.aitrios.sony-semicon.com

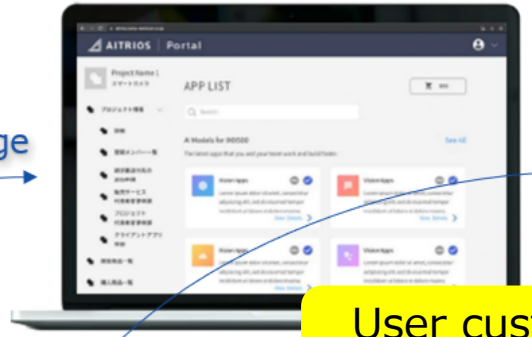
https://www.aitrios.sony-semicon.com/en

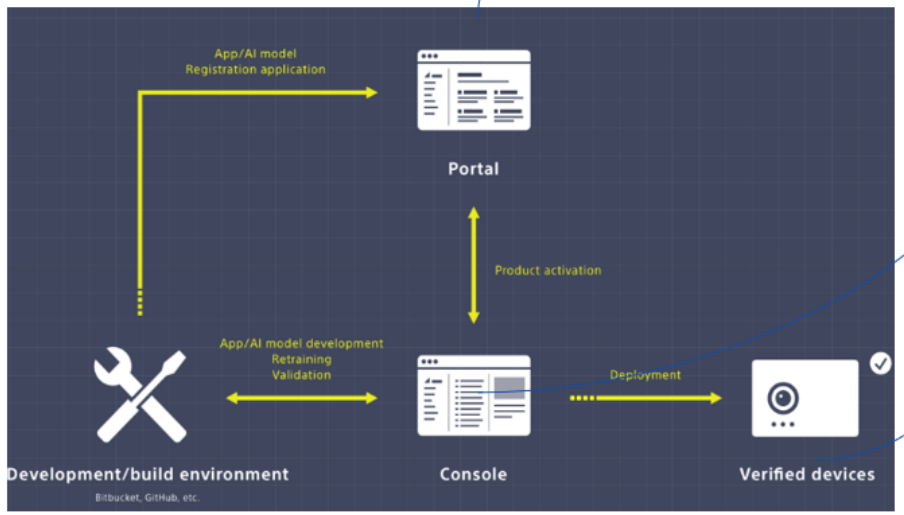
Product introduction – one-stop solution development



Helping partners efficiently develop and implement high-performance applications and solutions that meet their specific needs and easily build sensing solutions across edge devices and cloud services.

- ✓ Device - a family of devices that fit into different scenarios.
- ✓ Portal - maintains a portal for user logins and project applications, registers approved project members, and manages application and purchase information for commodity markets.
- ✓ Console - for building edge-to-cloud solutions quickly and efficiently.





User customized PPL program is compiled into .aot running based on WAMR

Sony AITRIOS announced 3 SmartCamera devices based on Espressif ESP32

Looking Back – Wamr bring up for Esp32

Wasm VM lectotype

Name	URL	Language	License	Type	Drawbacks	ESP32 compatible	OS	Priority
WAMR(AOT)	https://github.com/bytecodealliance/wasm-micro-runtime	C	Apache-2.0	AoT		Yes	NuttX	1
WASM3	https://github.com/wasm3/wasm3	C	MIT	Interpreter		Yes	NuttX	1
WAMR(Interp)	https://github.com/bytecodealliance/wasm-micro-runtime	C	Apache-2.0	Interpreter		Yes	NuttX	1
WAMR(JIT)	https://github.com/bytecodealliance/wasm-micro-runtime	C	Apache-2.0	JIT		Yes	NuttX	1
Fizzy	https://github.com/wasmx/fizzy	C++	Apache-2.0	Interpreter		Yes	Linux/Win	2
inNative	https://github.com/innative-sdk/innative	C++	Apache-2.0	AoT		No	Linux/Win	3
SSVM	https://github.com/second-state/ssvm	C++	Apache-2.0	AoT		No	Linux/Win	4
VMIR	https://github.com/andoma/vmir	C	MIT	JIT	out-dated			
WasmRT	https://github.com/rhitchcock/wasmrt	C++		Interpreter	out-dated			
EOSVM	https://github.com/EOSIO/eos-vm	C++		Interpreter	out-dated			
WAC	https://github.com/kanaka/wac	C	MPL-2.0	Interpreter	out-dated			
WasmVM	https://github.com/WasmVM/WasmVM	C++	BSD	Interpreter	EarlyDevelopment			
TWVM	https://github.com/Becavalier/TWVM	C++	MIT	Interpreter	EarlyDevelopment			
Wasmer	https://github.com/wasmerio/wasmer	Rust	MIT	AoT	Non C/C++			
WAVM	https://github.com/WAVM/WAVM	C++	BSD	JIT	64-bit only			
CMM of WASM	https://github.com/SimonJF/cmm_of_wasm	Ocaml	MIT	AoT	Non C/C++			
FDVM	https://github.com/funcdef/fdvm	JS		JIT	Non C/C++			
GraalWasm	https://github.com/oracle/graal/tree/master/wasm	Java		JIT	Non C/C++			
Happy new moon with report	https://github.com/fishjd/HappyNewMoonWithReport	Java	Apache-2.0	Interpreter	Non C/C++			
Life	https://github.com/perlin-network/life	Go	MIT	Interpreter	Non C/C++			
Lucet	https://github.com/fastly/lucet	Rust	Apache-2.0	AoT	Non C/C++			
Motor	https://github.com/penberg/motor	Rust	Apache-2.0	JIT	Non C/C++			
py-wasm	https://github.com/ethereum/py-wasm	Python		Interpreter	Non C/C++			
Serverless Wasm	https://github.com/Geal/serverless-wasm	Rust	MIT	JIT	Non C/C++			
Swam	https://github.com/satabin/swam	Scala	Apache-2.0	Interpreter	Non C/C++			
wagon	https://github.com/go-interpreter/wagon	Go	BSD	Interpreter	Non C/C++			
WAKit	https://github.com/akkyie/WAKit	Swift	MIT	Interpreter	Non C/C++			
Warpy	https://github.com/kanaka/warpy	Rpython	MPL-2.0	JIT	Non C/C++			
Wasmi	https://github.com/paritytech/wasmi	Rust	Apache-2.0	Interpreter	Non C/C++			
Wasmo	https://github.com/appcypher/wasmo	Rust	Apache-2.0	JIT	Non C/C++			
Wasmtime	https://github.com/CraneStation/wasmtime	Rust	Apache-2.0	JIT	Non C/C++			
webassembly	https://github.com/dcodeIO/webassembly	JS	MIT	JIT	Non C/C++			

- ✓ Nuttx support both WAMR and WASM already
- ✓ WAMR support AOT, but WASM3 does not
- ✓ WAMR & WASM3 are the preferred wasm runtime, WAMR has higher priority than WASM3

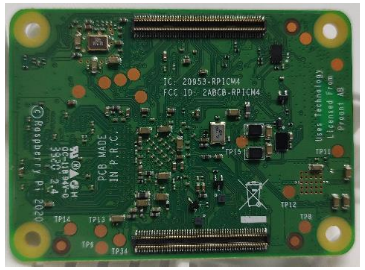
- ✓ Wamr support nuttx
- ✓ Wamr support C/C++
- ✓ Wamr support Aot
- ✓ Wamr has lower footprint

bytecodealliance / wasm-micro-runtime Public
WebAssembly Micro Runtime (WAMR)

+

Esp32 based device

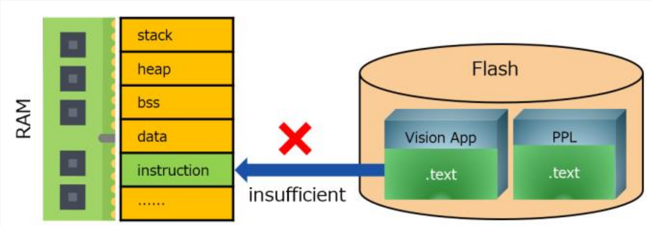
- ✓ Resource limited device
- ✓ 320K RAM
- ✓ 4M flash
- ✓ 4M PSRAM

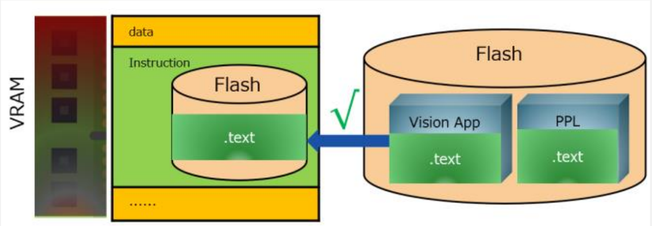


Bringup WAMR on resource limited device

1. Insufficient IRAM issue

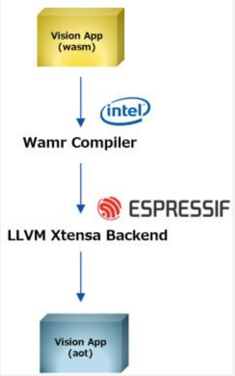
● Not contribute, a temp solution

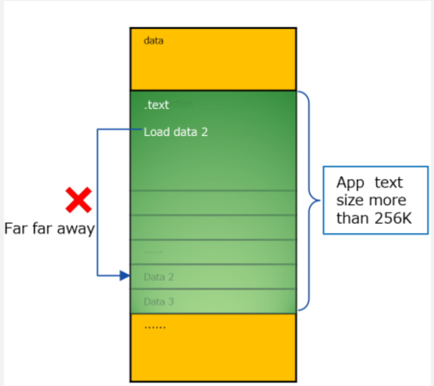


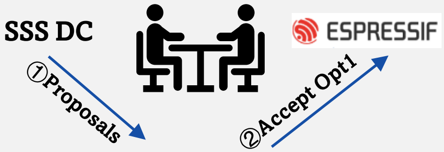


2. 256K limitation issue

● Contributed







1. Enable the --text-section-literal, to separate the .literal into several sub .literal sections and distributed in .text segment, to avoid far distance reference
2. Enable the "Extended L32R option" in its backend for xtensa instruction generation

3. XIP bring up on ESP32, 4 bytes alignment read in VRAM

● Contributed

Current Concerns

- To implement XIP on ESP32.
  - Stability issue debug tools
    - Any general way to debug the .aot, IR debug tools? (the general IR interpreter like lli cannot simulate the exec_env context)
  - General way to eliminate external reference.
    - Case by case currently. risky to deploy user's .aot when it has external reference.
    - Better to provide tools to check if the .aot have the reference in XIP mode before it's deployed.
  - Performance concerns.
    - Any way to improve the table lookup speed? Is it possible to make const data in cache?

```
static bool
load_native_symbol_section(const uint8 *buf, const uint8 *buf_end,
                           AOTModule *module, bool is_load_from_file_buf,
                           char *error_buf, uint32 error_buf_size)
{
        ......
    if (cnt > 0) {
        module->native_symbol_list = wasm_runtime_malloc(cnt * sizeof(void *));
        if (module->native_symbol_list == NULL) {
            set_error_buf(error_buf, error_buf_size,
                          "malloc native symbol list failed");
            goto fail;
        }

        for (i = cnt - 1; i >= 0; i--) {
            read_string(p, p_end, symbol);
            if (!strncmp(symbol, "f32#", 4) || !strncmp(symbol, "i32#", 4)) {
                uint32 u32;
                /* Resolve the raw int bits of f32 const */
                if (!str2uint32(symbol + 4, &u32)) {
                    set_error_buf_v(error_buf, error_buf_size,
                                    "resolve symbol %s failed", symbol);
                    goto fail;
                }
                *(uint32 *)(&module->native_symbol_list[i]) = u32;
            }
            else if (!strncmp(symbol, "f64#", 4)
                     || !strncmp(symbol, "i64#", 4)) {
                uint64 u64;
                /* Resolve the raw int bits of f64 const
                if (!str2uint64(symbol + 4, &u64)) {
                    set_error_buf_v(error_buf, error_buf
                                    "resolve symbol %s f    ed", symbol);
                    goto fail;
                }
                *(uint64 *)(&module->native_symbol_list[i]) = u64;
            }
        }

    }
}
```

32bit array in 32bit system

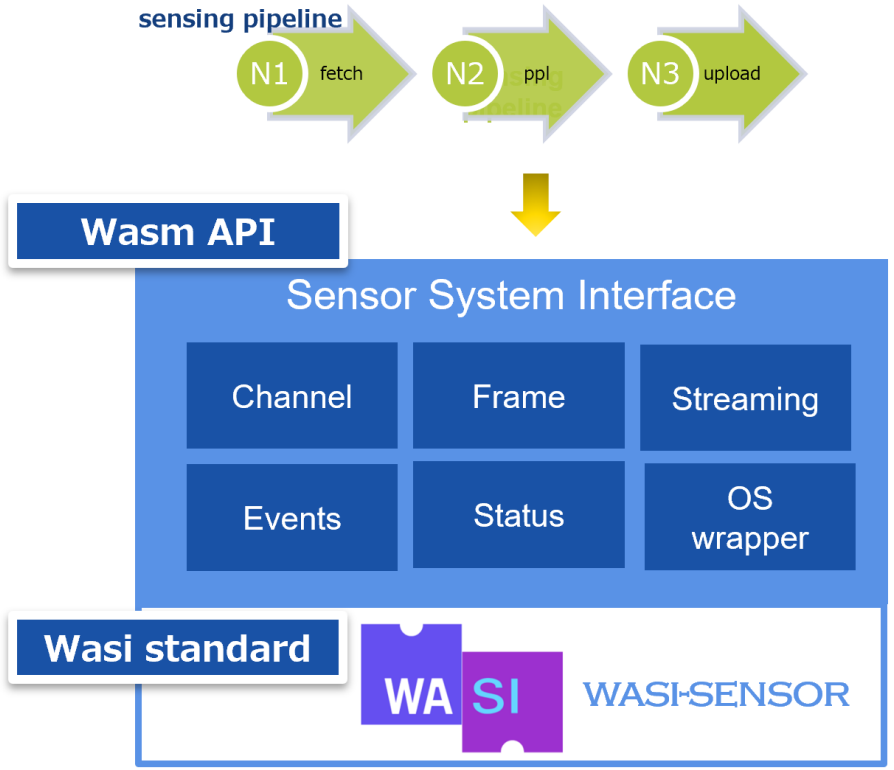
32bit writing

64bit writing

The value has overlap while 32bit element is next to its 64bit neighbor.

Road Map in Future for SSS (Maybe Add More Later)

- Run spec test using C runner rather than python script on resource limited device.
  - We already have one, can contribute it in future
- To support memory sharing between modules
  - Sensing data would be processed in a pipeline, to share the data from one node(each would be a wasm) to anther is reasonable demand. Expect to share the data crossing modules in a general way.
- Wasi-sensor standardization
  - SSS already has a set of API that could provide general service of sensor, such as SmartCamera, iTof, Viewing sensor. And the API has C/C++/Java/Python version, we'd like make it work for wasm also, then the standardization for wasi is necessary.



That’s all Thanks