WAMR Integration and Extension Guide

Intel Micro Runtime Team,

IAGS/SSP,

Intel

Revision: 3/1/2019

Contents

[1 Integration process 3](#_Toc2328988)

[2 Platform layer adaption 3](#_Toc2328989)

[3 WASM Application APIs 3](#_Toc2328990)

[3.1 WAMR built-in application API 3](#_Toc2328991)

[3.2 Platform APIs 4](#_Toc2328992)

[4 Develop a WASM Application 7](#_Toc2328993)

[5 WASM VM Interface 7](#_Toc2328994)

WAMR is the WebAssembly VM designed for embedded environment, so it took the portability and customization as the key design goals.

There are two necessary steps for integrating the WAMR for a target board:

1. Add or modify the WAMR platform layer for the board OS, and complete the full board image build with WAMR integrated
2. Extend the application APIs to develop more complicated WASM applications

Notes: This document is mostly about C based WASM application. There is no technical issues to support other programming languages such as RUST.

# Integrate WAMR to new platforms

Please refer to the repo folder “runtime/platform” to find out different platforms support source code. You will find Linux and Zephyr OS as integration samples.

# Integrate WAMR in app framework

Board vendors would like to develop its own app framework, such as dynamic WASM app installation from cloud, multiple WASM app management etc.

WAMR VM exposes a set of APIs to the framework to load WASM binaries and execute specified WASM functions.

They are defined in the header file wasm-export.h

|  |
| --- |
| /\* Uninstantiated WASM module loaded from WASM binary file \*/  struct WASMModule;  typedef struct WASMModule \*wasm\_module\_t;  /\* Instantiated WASM module \*/  struct WASMModuleInstance;  typedef struct WASMModuleInstance \*wasm\_module\_inst\_t;  /\* Function instance \*/  struct WASMFunctionInstance;  typedef struct WASMFunctionInstance \*wasm\_function\_inst\_t;  /\* WASM section \*/  typedef struct wasm\_section {  struct wasm\_section \*next;  /\* section type \*/  int section\_type;  /\* section body, not include type and size \*/  uint8\_t \*section\_body;  /\* section body size \*/  uint32\_t section\_body\_size;  } wasm\_section\_t, \*wasm\_section\_list\_t;  /\* Execution environment, e.g. stack info \*/  typedef struct WASMExecEnv {  uint8\_t \*stack;  uint32\_t stack\_size;  } \*wasm\_exec\_env\_t;  /\* Package Type \*/  typedef enum {  Wasm\_Module\_Bytecode = 0,  Wasm\_Module\_AoT,  Package\_Type\_Unknown = 0xFFFF  } package\_type\_t;  /\*\*  \* Initialize the WASM runtime environment.  \*  \* @return true if success, false otherwise  \*/  bool  wasm\_runtime\_init();  /\*\*  \* Destroy the WASM runtime environment.  \*/  void  wasm\_runtime\_destroy();  /\*\*  \* Get the package type of a buffer.  \*  \* @param buf the package buffer  \* @param size the package buffer size  \*  \* @return the package type, return Package\_Type\_Unknown if the type is unknown  \*/  package\_type\_t  get\_package\_type(const uint8\_t \*buf, uint32\_t size);  /\*\*  \* Load a WASM module from a specified byte buffer.  \*  \* @param buf the byte buffer which contains the WASM binary data  \* @param size the size of the buffer  \* @param error\_buf output of the exception info  \* @param error\_buf\_size the size of the exception string  \*  \* @return return WASM module loaded, NULL if failed  \*/  wasm\_module\_t  wasm\_runtime\_load(const uint8\_t \*buf, uint32\_t size,  char \*error\_buf, uint32\_t error\_buf\_size);  /\*\*  \* Load a WASM module from a specified WASM section list.  \*  \* @param section\_list the section list which contains each section data  \* @param error\_buf output of the exception info  \* @param error\_buf\_size the size of the exception string  \*  \* @return return WASM module loaded, NULL if failed  \*/  wasm\_module\_t  wasm\_runtime\_load\_from\_sections(wasm\_section\_list\_t section\_list,  char \*error\_buf, uint32\_t error\_buf\_size);  /\*\*  \* Unload a WASM module.  \*  \* @param module the module to be unloaded  \*/  void  wasm\_runtime\_unload(wasm\_module\_t module);  /\*\*  \* Instantiate a WASM module.  \*  \* @param module the WASM module to instantiate  \* @param stack\_size the default stack size of the module instance, a stack  \* will be created when function wasm\_runtime\_call\_wasm() is called  \* to run WASM function and the exec\_env argument passed to  \* wasm\_runtime\_call\_wasm() is NULL. That means this parameter is  \* ignored if exec\_env is not NULL.  \* @param heap\_size the default heap size of the module instance, a heap will  \* be created besides the app memory space. Both wasm app and native  \* function can allocate memory from the heap. If heap\_size is 0, the  \* default heap size will be used.  \* @param error\_buf buffer to output the error info if failed  \* @param error\_buf\_size the size of the error buffer  \*  \* @return return the instantiated WASM module instance, NULL if failed  \*/  wasm\_module\_inst\_t  wasm\_runtime\_instantiate(const wasm\_module\_t module,  uint32\_t stack\_size, uint32\_t heap\_size,  char \*error\_buf, uint32\_t error\_buf\_size);  /\*\*  \* Deinstantiate a WASM module instance, destroy the resources.  \*  \* @param module\_inst the WASM module instance to destroy  \*/  void  wasm\_runtime\_deinstantiate(wasm\_module\_inst\_t module\_inst);  /\*\*  \* Load WASM module instance from AOT file.  \*  \* @param aot\_file the AOT file of a WASM module  \* @param aot\_file\_size the AOT file size  \* @param error\_buf buffer to output the error info if failed  \* @param error\_buf\_size the size of the error buffer  \*  \* @return the instantiated WASM module instance, NULL if failed  \*/  wasm\_module\_inst\_t  wasm\_runtime\_load\_aot(uint8\_t \*aot\_file, uint32\_t aot\_file\_size, uint32\_t heap\_size,  char \*error\_buf, uint32\_t error\_buf\_size);  /\*\*  \* Lookup an exported function in the WASM module instance.  \*  \* @param module\_inst the module instance  \* @param name the name of the function  \* @param signature the signature of the function, use "i32"/"i64"/"f32"/"f64"  \* to represent the type of i32/i64/f32/f64, e.g. "(i32i64)" "(i32)f32"  \*  \* @return the function instance found, if the module instance is loaded from  \* the AOT file, the return value is the function pointer  \*/  wasm\_function\_inst\_t  wasm\_runtime\_lookup\_function(const wasm\_module\_inst\_t module\_inst,  const char \*name, const char \*signature);  /\*\*  \* Create execution environment.  \*  \* @param stack\_size the stack size to execute a WASM function  \*  \* @return the execution environment  \*/  wasm\_exec\_env\_t  wasm\_runtime\_create\_exec\_env(uint32\_t stack\_size);  /\*\*  \* Destroy the execution environment.  \*  \* @param env the execution environment to destroy  \*/  void  wasm\_runtime\_destory\_exec\_env(wasm\_exec\_env\_t env);  /\*\*  \* Call the given WASM function of a WASM module instance with  \* arguments (bytecode and AoT).  \*  \* @param module\_inst the WASM module instance which the function belongs to  \* @param exec\_env the execution environment to call the function. If the module  \* instance is created by AoT mode, it is ignored and just set it to NULL.  \* If the module instance is created by bytecode mode and it is NULL,  \* a temporary env object will be created  \* @param function the function to be called  \* @param argc the number of arguments  \* @param argv the arguments. If the function method has return value,  \* the first (or first two in case 64-bit return value) element of  \* argv stores the return value of the called WASM function after this  \* function returns.  \*  \* @return true if success, false otherwise and exception will be thrown,  \* the caller can call wasm\_runtime\_get\_exception to get exception info.  \*/  bool  wasm\_runtime\_call\_wasm(wasm\_module\_inst\_t module\_inst,  wasm\_exec\_env\_t exec\_env,  wasm\_function\_inst\_t function,  uint32\_t argc, uint32\_t argv[]);  /\*\*  \* Get exception info of the WASM module instance.  \*  \* @param module\_inst the WASM module instance  \*  \* @return the exception string  \*/  const char\*  wasm\_runtime\_get\_exception(wasm\_module\_inst\_t module\_inst);  /\*\*  \* Clear exception info of the WASM module instance.  \*  \* @param module\_inst the WASM module instance  \*/  void  wasm\_runtime\_clear\_exception(wasm\_module\_inst\_t module\_inst);  /\*\*  \* Attach the current native thread to a WASM module instance.  \* A native thread cannot be attached simultaneously to two WASM module  \* instances. The WASM module instance will be attached to the native  \* thread which it is instantiated in by default.  \*  \* @param module\_inst the WASM module instance to attach  \* @param thread\_data the thread data that current native thread requires  \* the WASM module instance to store  \*  \* @return true if SUCCESS, false otherwise  \*/  bool  wasm\_runtime\_attach\_current\_thread(wasm\_module\_inst\_t module\_inst,  void \*thread\_data);  /\*\*  \* Detach the current native thread from a WASM module instance.  \*  \* @param module\_inst the WASM module instance to detach  \*/  void  wasm\_runtime\_detach\_current\_thread(wasm\_module\_inst\_t module\_inst);  /\*\*  \* Get the thread data that the current native thread requires the WASM  \* module instance to store when attaching.  \*  \* @return the thread data stored when attaching  \*/  void\*  wasm\_runtime\_get\_current\_thread\_data();  /\*\*  \* Get current WASM module instance of the current native thread  \*  \* @return current WASM module instance of the current native thread, NULL  \* if not found  \*/  wasm\_module\_inst\_t  wasm\_runtime\_get\_current\_module\_inst();  /\*\*  \* Allocate memory from the heap of WASM module instance  \*  \* @param module\_inst the WASM module instance which contains heap  \* @param size the size bytes to allocate  \*  \* @return the allocated memory address, which is a relative offset to the  \* base address of the module instance's memory space, the value range  \* is (-heap\_size, 0). Note that it is not an absolute address.  \* Return non-zero if success, zero if failed.  \*/  int32\_t  wasm\_runtime\_module\_malloc(wasm\_module\_inst\_t module\_inst, uint32\_t size);  /\*\*  \* Free memory to the heap of WASM module instance  \*  \* @param module\_inst the WASM module instance which contains heap  \* @param ptr the pointer to free  \*/  void  wasm\_runtime\_module\_free(wasm\_module\_inst\_t module\_inst, int32\_t ptr);  /\*\*  \* Allocate memory from the heap of WASM module instance and initialize  \* the memory with src  \*  \* @param module\_inst the WASM module instance which contains heap  \* @param src the source data to copy  \* @param size the size of the source data  \*  \* @return the allocated memory address, which is a relative offset to the  \* base address of the module instance's memory space, the value range  \* is (-heap\_size, 0). Note that it is not an absolute address.  \* Return non-zero if success, zero if failed.  \*/  int32\_t  wasm\_runtime\_module\_dup\_data(wasm\_module\_inst\_t module\_inst,  const char \*src, uint32\_t size);  /\*\*  \* Validate the app address, check whether it belongs to WASM module  \* instance's address space, or in its heap space or memory space.  \*  \* @param module\_inst the WASM module instance  \* @param app\_offset the app address to validate, which is a relative address  \* @param size the size bytes of the app address  \*  \* @return true if success, false otherwise. If failed, an exception will  \* be thrown.  \*/  bool  wasm\_runtime\_validate\_app\_addr(wasm\_module\_inst\_t module\_inst,  int32\_t app\_offset, uint32\_t size);  /\*\*  \* Validate the native address, check whether it belongs to WASM module  \* instance's address space, or in its heap space or memory space.  \*  \* @param module\_inst the WASM module instance  \* @param native\_ptr the native address to validate, which is an absolute  \* address  \* @param size the size bytes of the app address  \*  \* @return true if success, false otherwise. If failed, an exception will  \* be thrown.  \*/  bool  wasm\_runtime\_validate\_native\_addr(wasm\_module\_inst\_t module\_inst,  void \*native\_ptr, uint32\_t size);  /\*\*  \* Convert app address(relative address) to native address(absolute address)  \*  \* @param module\_inst the WASM module instance  \* @param app\_offset the app adress  \*  \* @return the native address converted  \*/  void \*  wasm\_runtime\_addr\_app\_to\_native(wasm\_module\_inst\_t module\_inst,  int32\_t app\_offset);  /\*\*  \* Convert native address(absolute address) to app address(relative address)  \*  \* @param module\_inst the WASM module instance  \* @param native\_ptr the native address  \*  \* @return the app address converted  \*/  int32\_t  wasm\_runtime\_addr\_native\_to\_app(wasm\_module\_inst\_t module\_inst,  void \*native\_ptr);  /\*\*  \* Find the unique main function from a WASM module instance  \* and execute that function.  \*  \* @param module\_inst the WASM module instance  \* @param argc the number of arguments  \* @param argv the arguments array  \*  \* @return true if the main function is called, false otherwise.  \*/  bool  wasm\_application\_execute\_main(wasm\_module\_inst\_t module\_inst,  int argc, char \*argv[]);  /\*\*  \* Find the specified function in argv[0] from WASM module of current instance  \* and execute that function.  \*  \* @param module\_inst the WASM module instance  \* @param name the name of the function to execute  \* @param argc the number of arguments  \* @param argv the arguments array  \*  \* @return true if the specified function is called, false otherwise.  \*/  bool  wasm\_application\_execute\_func(wasm\_module\_inst\_t module\_inst,  const char \*name, int argc, char \*argv[]); |

Below is the sample code to load WASM app which is in AOT mode:

|  |
| --- |
| char \*file\_buffer, err[256];  int file\_size, err\_size = 256;  package\_type\_t type;  wasm\_module\_inst\_t inst;  wasm\_function\_inst\_t func;  uint32 argv\_buf[1] = {35};  bh\_memory\_init\_with\_pool(global\_heap\_buf, sizeof(global\_heap\_buf));    read\_file\_to\_buffer("./test.aot", &file\_buffer, &file\_size);  type = get\_package\_type((unsigned char \*)file\_buffer, file\_size);  assert(type == Wasm\_Module\_AoT);    wasm\_runtime\_init();  inst = wasm\_runtime\_load\_aot(file\_buffer, file\_size, 0, err, err\_size);  func = wasm\_runtime\_lookup\_function(inst, "fib", "(i32)i32");    // The 2nd parameter is ignored in AOT mode, so just set it to NULL  if (!wasm\_runtime\_call\_wasm(inst, NULL, func, 1, argv\_buf)) {  printf(“Got exception running wasm code: %s\n”, wasm\_runtime\_get\_exception(inst));  wasm\_runtime\_clear\_exception(inst);  }    wasm\_runtime\_deinstantiate(inst);  wasm\_runtime\_destroy();  wasm\_free(file\_buffer); |

Below is the sample code to load WASM app which is in interpreter/bytecode mode:

|  |
| --- |
| char \*file\_buffer, err[256];  int file\_size, err\_size = sizeof(err), stack\_size = 1024;  package\_type\_t type;  wasm\_module\_t module;  wasm\_module\_inst\_t inst;  wasm\_function\_inst\_t func;  wasm\_exec\_env\_t env;  uint32 argv\_buf[1] = {35};  bh\_memory\_init\_with\_pool(global\_heap\_buf, sizeof(global\_heap\_buf));  read\_file\_to\_buffer("./test.wasm", &file\_buffer, &file\_size);  type = get\_package\_type((unsigned char \*)file\_buffer, file\_size);  assert(type == Wasm\_Module\_Bytecode);  wasm\_runtime\_init();  module = wasm\_runtime\_load((unsigned char \*)file\_buffer, file\_size, err, err\_size);  inst = wasm\_runtime\_instantiate(module, 0, 0, err, err\_size);  func = wasm\_runtime\_lookup\_function(inst, "fib", "(i32)i32");  env = wasm\_runtime\_create\_exec\_env(stack\_size);  if (!wasm\_runtime\_call\_wasm(inst, env, func, 1, argv\_buf) ) {  printf(“Got exception running wasm code: %s\n”, wasm\_runtime\_get\_exception(inst));  wasm\_runtime\_clear\_exception(inst);  }  wasm\_runtime\_destory\_exec\_env(env);  wasm\_runtime\_deinstantiate(inst);  wasm\_runtime\_unload(module);  wasm\_runtime\_destroy();  wasm\_free(file\_buffer); |

# Extend WAMR Application library

## Application library sources

There are 3 sources of APIs for programming the WASM application:

1. **Built-in API**s: WAMR has already provided a minimal API set for developers. The minimal API includes:
   * Libc APIs, which is the minimal Libc APIs like memory allocation and string copy etc. It is defined in lib/app-libs/libc/lib-base.h;
   * Base library, which is the basic support like communication, timers and request/sub etc. It is defined is lib/app-libs/base/wasm\_app.h;
   * Extension library, which is a reference code of library extension. Currently we provide an example of extending library to support sensors, the header file lib/app-libs/extension/sensor/sensor.h. It is a reference implementation for board vendors.
2. **3rd party APIs**: Programmer can download include any 3rd party C source code, and added into their own WASM app source tree.
3. **Platform native APIs**: The board vendors define these APIs during their making board firmware. They are provided WASM application to invoke like built-in and 3rd party APIs. In this way board vendors extend APIs which can make programmers develop more complicated WASM apps.

## Built-in APIs

By using built-in APIs, programmers are able to develop WASM apps to fulfill basic usage scenarios like WASM app life cycle management, inter-app messaging and sensor monitor etc.

The built-in APIs are defined in “lib/app-libs”, it includes three parts:

**Part 1. Libc APIs.**

The header files is “lib/app-libs/libc/lib-base.h”. The API set is listed as below:

|  |
| --- |
| void \*malloc(size\_t size);  void \*calloc(size\_t n, size\_t size);  void free(void \*ptr);  int memcmp(const void \*s1, const void \*s2, size\_t n);  void \*memcpy(void \*dest, const void \*src, size\_t n);  void \*memmove(void \*dest, const void \*src, size\_t n);  void \*memset(void \*s, int c, size\_t n);  int putchar(int c);  int snprintf(char \*str, size\_t size, const char \*format, ...);  int sprintf(char \*str, const char \*format, ...);  char \*strchr(const char \*s, int c);  int strcmp(const char \*s1, const char \*s2);  char \*strcpy(char \*dest, const char \*src);  size\_t strlen(const char \*s);  int strncmp(const char \* str1, const char \* str2, size\_t n);  char \*strncpy(char \*dest, const char \*src, unsigned long n); |

**Part 2. Base library.**

The header files is “lib/app-libs/base/wasm-app.h”, it includes:

* request and response APIs
* event pub/sub APIs
* timer APIs

The API set is listed as below:

|  |
| --- |
| typedef void(\*request\_handler\_f)(request\_t \*) ;  typedef void(\*response\_handler\_f)(response\_t \*, void \*) ;  // Request APIs  void init\_resource\_register();  bool api\_register\_resource\_handler(const char \*url, request\_handler\_f);  void api\_send\_request(request\_t \* request, response\_handler\_f response\_handler, void \* user\_data);  void api\_response\_send(response\_t \*response);  // event API  bool api\_publish\_event(const char \*url, int fmt, void \*payload, int payload\_len);  bool api\_subscribe\_event(const char \* url, request\_handler\_f handler);  struct user\_timer;  typedef struct user\_timer \* user\_timer\_t;  // Timer APIs  user\_timer\_t api\_timer\_create(int interval, bool is\_period, bool auto\_start, void(\*on\_user\_timer\_update)(user\_timer\_t  ));  void api\_timer\_cancel(user\_timer\_t timer);  void api\_timer\_restart(user\_timer\_t timer, int interval); |

**Part 3. Library extension reference.**

Currently we provide an example of sensor, the header file lib/app-libs/extension/sensor/sensor.h, the API set is listed as below:

|  |
| --- |
| sensor\_t sensor\_open(const char\* name, int index,  void(\*on\_sensor\_event)(sensor\_t, attr\_container\_t \*, void \*),  void \*user\_data);  bool sensor\_config(sensor\_t sensor, int interval, int bit\_cfg, int delay);  bool sensor\_config\_with\_attr\_container(sensor\_t sensor, attr\_container\_t \*cfg);  bool sensor\_close(sensor\_t sensor); |

## APIs extension

API extension means to export new “Platform API” to WASM apps, to develop more complicated WASM application for this platform. “Platform API” can be any function defined by the platform OS or the board firmware code.

***[Security attention] The WebAssembly application is supposed to access its own memory space. If the exposed platform API includes the pointers to system memory space which out of the app memory space, the integrator should carefully design some wrapper function to ensure the memory boundary is not broken.***

WAMR implemented a framework for developers to export APIs. The procedure to expose the platform APIs in three steps:

**Step 1. Create a header file**

Declare the APIs for WASM application source project to include.

**Step 2. Create a source file**

Export the platform APIs, for example in products/linux/ext-lib-export.c

|  |
| --- |
| #include "lib-export.h"  static NativeSymbol extended\_native\_symbol\_defs[] =  {  };  #include "ext-lib-export.h" |

**Step 3. Register new APIs**

Use macro EXPORT\_WASM\_API and EXPORT\_WASM\_API2 to add exported APIs into the array of extended\_native\_symbol\_defs.

The pre-defined two MACROs below should be used to declare a function export:

|  |
| --- |
| #define EXPORT\_WASM\_API(symbol) {#symbol, symbol}  #define EXPORT\_WASM\_API2(symbol) {#symbol, symbol\_##wrapper} |

The type of array extended\_native\_symbol\_defs[] is defined as below:

|  |
| --- |
| typedef struct NativeSymbol {  const char \*symbol;  void \*func\_ptr;  } NativeSymbol; |

Below code example shows how to extend the library to support GPIO pin operations on Zephyr OS:

**lib-export-impl.c**

|  |
| --- |
| #include <zephyr.h>  #include <kernel.h>  #include <gpio.h>  static void customized()  {  // your code  }  static int  gpio\_pin\_configure\_wrapper(struct device \*port, u32\_t pin, int flags)  {  return gpio\_pin\_configure(port, pin, flags); // a Zephyr OS API  } |

**lib-export-dec.h**

|  |
| --- |
| #ifndef \_LIB\_EXPORT\_DEC\_H\_  #define \_LIB\_EXPORT\_DEC\_H\_  #ifdef \_\_cplusplus  extern "C" {  #endif  void customized();  int gpio\_pin\_configure\_wrapper(struct device \*port, u32\_t pin, int flags);  #ifdef \_\_cplusplus  }  #endif  #endif |

**ext-lib-export.c**

|  |
| --- |
| #include "lib-export.h"  #include "lib-export-dec.h"  static NativeSymbol extended\_native\_symbol\_defs[] =  {  EXPORT\_WASM\_API(customized),  EXPORT\_WASM\_API2(gpio\_pin\_configure)  };  #include "ext-lib-export.h" |

## Use extended library

In the application source project, it includes the WAMR built-in APIs header file and platform extension header files.

Assume the board vendor extend the library which added a API called customized(). The WASM application would be like this:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <stdlib.h>  #include <stdlib.h>  #include “lib-base.h” // provided by WAMR  #include “lib-export-dec.h” // provided by platform vendor  int main(int argc, char \*\*argv)  {  int I;  char \*buf = “abcd”;  i = strlen(buf); // common API provided by WAMR  customized(); // customized API provided by platform vendor  return i;  } |