iWASM Source Code User Guide

Intel Micro Runtime Team, SSG

Contents

[1 Integration process 1](#_Toc2178120)

[2 Platform layer adaption 1](#_Toc2178121)

[3 WASM Application APIs 2](#_Toc2178122)

[3.1 iWASM built-in application API 2](#_Toc2178123)

[3.2 Platform APIs 3](#_Toc2178124)

[4 Develop a WASM Application 5](#_Toc2178125)

[5 WASM VM Interface 6](#_Toc2178126)

# Integration process

iWASM is a WebAssembly VM designed for embedded environment, so it took the portability and customization as the a key design goal.

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

1. Add or modify the iWASM platform layer for the board OS, and complete the full image build with iWASM integrated
2. Export the application APIs for WASM application development

Notes: This document is mostly about C based WASM application. It should be no technical problem for supporting other programming languages such as RUST.

# Platform layer adaption

To be finished.

# WASM Application APIs

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

1. **iWASM built-in API**: iWASM has already provided a minimal API set for applications. The minimal API declaration header file is “lib/lib-base.h”.
2. **Application ported API**: The application can include any C source code from any 3rd party library, and use the emcc to compile it and integrate the implementation of referred APIs into the application package.
3. **Platform APIs**: The board vendor can choose what platform APIs to be exposed to WASM applications during making the board firmware.

## iWASM built-in application API

The built-in APIs are defined in “lib/lib-base.h”, WASM application project should include this header file. 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); |

## Platform APIs

This section explains how to export “Platform API” for developing WASM application in the target platform. “Platform API” can be any function defined by the OS or the board firmware code.

iWASM implemented a framework for developers to export APIs. The procedure to expose the platform APIs:

1. Create a header file to declare the APIs for WASM application source project to include.

2. Modify “lib/lib-export-template.c” from the iWASM repo, and add the exported function names into the array “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; |

**NOTE:** please **DO NOT** edit any other files except for “lib/lib-export-template.c” under iWASM repo.

Example for Zephyr OS:

a. lib/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  } |

b. lib/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 |

c. lib/lib-export-template.c

|  |
| --- |
| #include "lib-export-template.c"  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include "lib-export.h"  NativeSymbol extended\_native\_symbol\_defs[] = {  EXPORT\_WASM\_API(customized),  EXPORT\_WASM\_API2(gpio\_pin\_configure)  }; |

# Develop a WASM Application

Application can include the iWASM built-in APIs header file and platform provided header files in the source project, then build the WASM package by using the emcc compiler.

An WASM application example for Zephyr OS:

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

# WASM VM Interface

This section specifies the interface that IWASM VM exposed for the system to load WASM package and execute the functions from the WASM program.

The header file wasm-export.h:

|  |
| --- |
| #include <inttypes.h>  #include <stdbool.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;  /\* Execution environment, e.g. stack info \*/  typedef struct WASMExecEnv {  uint8\_t \*stack;  uint32\_t stack\_size;  } \*wasm\_exec\_env\_t;  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);  /\*\*  \* 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 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,  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(char\* aot\_file, uint32\_t aot\_file\_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 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); |

Sample code for 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};    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, 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();  free(file\_buffer); |

Sample code for 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};  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, 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();  free(file\_buffer); |