Access to Retro Developer Guide

Table of Contents

1. Introduction & Overview

Access to Retro frontend uses “virtual consoles” to emulate games. The concept of a “virtual console” was implemented to abstract emulators from the users making them seem like something much more familiar – a video game console. However, a virtual console is just an emulator that was compiled to a dynamic library with specific symbols defined and the Access to Retro developer library is a link that connects the frontend to the virtual console, this document aims to explain the features of the library however it should not be used instead of the documentation.

The library itself was written in C and can be compiled using any compiler that supports the C17 standard (which includes every modern compiler), as for the dependencies the only dependency needed by the library is the SDL2 library as there are many helper functions that deal with SDL2 graphical output. In the future there might be more dependencies as support for more graphical outputs is implemented and this guide will be updated.

The library will always be compiled as static, due to the nature of how it works there will be no support for dynamic linking.

2. Creating a Virtual Console project

While you can create the virtual console is any compiled programming language that interfaces with C, the default languages for virtual consoles are considered to be C and C++. There is a useful script, in the scripts folder in the repository, named “create-virtual-console-app.py” that will generate a C/C++ CMake project for you (support for other languages is planned). Notice the name is similar to a popular script “create-react-app” used in React framework, this script works very similarly to that one, it allows you to define basic information about the virtual console and it will create a project with basic code for you.

You can also create the project by yourself however you need to make sure that you link with the Access to Retro Developer Library AND SDL2.

To use the library, you should only include the main header, for example:

“#include <access-to-retro-dev/access-to-retro-dev.h>”

When using the script, the required symbols are abstracted using macros and it is very strongly recommended that, even when you make the project yourself, to use those macros however when using another language, it is not possible and they have to be defined by hand with C calling convention (for example: in C++ you can use “extern ‘C’” before function definition). Here are all the required symbols and what they represent (please see documentation for the macros for more in-depth description of these symbols):

Note: To define required symbol use AR_DEFINE_REQUIRED_FN_MACRO(SYMBOL_MACRO) where symbol macro is the first part of below headings.

2.1. AR_DEFINE_FN / _ar_vc_define

“ar_error_code _ar_vc_define(void)”
->
Function returning error code (32-bit integer) named _ar_vc_define that takes no arguments.

This function should be used to define basic information about the virtual console as it will be run whenever the emulator is loaded into the frontend (at startup), a function named “ar_define” NEEDS to be called within it. Graphics method (the way in which the virtual console will handle graphics) should also be selected here using “ar_graphics_set_method”. The return value of this function should be 0 if no error happened or any other integer on error.

Here is an example of _ar_vc_define from Access to Retro CHIP8 virtual console:

```
AR_DEFINE_REQUIRED_FN(AR_DEFINE_FN)
{
    ar_define(NAME, SYSTEM, AUTHOR, ROM_EXT, VERSION,
              FRAME_RATE, DEFAULT_WINDOW_WIDTH, DEFAULT_WINDOW_HEIGHT);

    // Set graphics method to SDL
    ar_graphics_set_method( method: ar_graphics_method_sdl);

    return 0;
}
```

2.2. AR_STARTUP_FN / _ar_vc_startup

“ar_error_code _ar_vc_startup(void)”
->
Function returning error code (32-bit integer) named _ar_vc_startup that takes no arguments.

This function should be used to prepare the emulator for start-up, things like loading a binary file into the emulator (or creating the emulator object itself) should happen in this function (“ar_get_executable”) as the function will run whenever the emulator and game are launched by the user. “ar_init” which initialises the library itself also NEEDS to be called within it. The return value of this function should be 0 if no error happened or any other integer on error.

Here is an example of _ar_vc_startup from Access to Retro CHIP8 virtual console:

```
AR_DEFINE_REQUIRED_FN(AR_STARTUP_FN)
{
    ar_init();

    ar::chip8::emulator::create_global_emulator();

    ar::chip8::emulator::get_global_emulator()->access_ram().load_binary( exec

    return 0;
}
```

2.3. AR_QUIT_FN / _ar_vc_quit

“void _ar_vc_quit(void)”
->
Function returning nothing named _ar_vc_quit that takes no arguments.

This function should be used to clean-up resources after the emulator quits and will run when the user exits the emulator, “ar_quit” also NEEDS to be called within it as it cleans up the resources of the developer library.

Here is an example of _ar_vc_quit from Access to Retro CHIP8 virtual console:

```
AR_DEFINE_REQUIRED_FN(AR_QUIT_FN)
{
    ar_quit();
}
```

2.4. AR_THREAD_MAIN_FN / _ar_vc_main_thread

“void _ar_vc_main_thread(void)”
->
Function returning nothing named _ar_vc_main_thread(void) that takes no arguments.

The code inside this function will run in a separate thread when the emulator runs – you can consider it a main thread of the emulator so, for example: instructions should be executed here. This code will run in a specific interval, for example if the frame rate of the emulator is defined to be 60 then this means that the code inside the function will run once every 16.6ms (frame time), the time taken for the code to run will be deducted from the 16.6ms so if the code inside the function takes 2ms to run then it will be run again in 14.6ms.

Here is an example of _ar_vc_main_thread from Access to Retro CHIP8 virtual console:

```
AR_DEFINE_REQUIRED_FN(AR_THREAD_MAIN_FN)
{
    ar::chip8::cpu& cpu = ar::chip8::emulator::get_global_emulator()->access_cpu();

    /*
     * To calculate how many instructions need to be executed per frame it can be calculated using
     * formula: CLOCK_SPEED / FRAME_RATE
     *
     * Clock speed is how many instructions are executed per second
     * Frame rate is how many frames are shown per second (how many times this function gets called too)
     *
     * So, for framerate of 60 this function gets called 60 times per second and for clock speed of 600hz
     * 600 instructions needs to be executed per second therefore 600 / 60 = 10;
     */
    for (auto i = 0; i < std::floor( lcpp_x: ar::chip8::CLOCK_SPEED / ar::chip8::FRAME_RATE); i++)
    {
        // Execute one instruction (one cpu tick)
        cpu.tick();
    }

    // Timers should tick at constant 60hz and not 600hz that cpu runs on so tick timers here and not in the loop
    cpu.tick_timers();
}
```

2.5. AR_THREAD_RENDER_FN / _ar_vc_render_thread

“void _ar_vc_render_thread(void)”
->
Function returning nothing named _ar_vc_render_thread(void) that takes no arguments.

The code inside this function will run in a separate thread when the emulator runs – you can consider it a rendering thread of the emulator so, for example: graphics should be renderer here. This code will run in a specific interval, for example if the frame rate of the emulator is defined to be 60 then this means that the code inside the function will run once every 16.6ms (frame time), the time taken for the code to run will be deducted from the 16.6ms so if the code inside the function takes 2ms to run then it will be run again in 14.6ms.

Here is an example of _ar_vc_render_thread from Access to Retro CHIP8 virtual console:

```
AR_DEFINE_REQUIRED_FN(AR_THREAD_RENDER_FN)
{
    ar::chip8::gpu& gpu = ar::chip8::emulator::get_global_emulator()->access_gpu();

    if (gpu.get_draw_flag())
    {
        // If draw flag is set then draw to the screen
        gpu.render();

        // Let emulator know rendering happened (clear CPU's draw flag)
        gpu.set_draw_flag( new_value: false);
    }
}
```

2.6. AR_THREAD_INPUT_FN / _ar_vc_input_thread

“void _ar_vc_input_thread(void)”
->
Function returning nothing named _ar_vc_input_thread(void) that takes no arguments.

The code inside this function will run in a separate thread when the emulator runs – you can consider it a input thread of the emulator so, for example: input should be handled here. This code will run in a specific interval, for example if the frame rate of the emulator is defined to be 60 then this means that the code inside the function will run once every 16.6ms (frame time), the time taken for the code to run will be deducted from the 16.6ms so if the code inside the function takes 2ms to run then it will be run again in 14.6ms.

Here is an example of _ar_vc_input_thread from Access to Retro CHIP8 virtual console:

```
AR_DEFINE_REQUIRED_FN(AR_THREAD_INPUT_FN)
{
    ar::chip8::controller& controller = ar::chip8::emulator::get_global_emulator()->access_controller();


    controller.set_key_status( key: ar::chip8::key::key_4,  status: ar_get_unified_controller_key_status
            ( key: ar_unified_controller_left_analog_left));

    controller.set_key_status( key: ar::chip8::key::key_5,  status: ar_get_unified_controller_key_status
            ( key: ar_unified_controller_right_trigger));

    controller.set_key_status( key: ar::chip8::key::key_6,  status: ar_get_unified_controller_key_status
            ( key: ar_unified_controller_left_analog_right));
}
```

3. Graphical Method (Graphics API Abstraction)

To allow a great amount of flexibility for the developers the graphics abstraction is not something that has to be used, especially if there is no abstraction for the specific graphical API/method that the developer wants to use in his virtual console. There are three graphical methods available (that have an abstraction):

- SDL2 Library
- OpenGL
- Frame Buffer

3.1. SDL2

SDL2 abstraction transfers the ownership of the SDL objects and the burden of initialising them to the frontend.

This allows the developer to then use “ar_graphics_get_sdl_renderer” or “ar_graphics_get_sdl_window” to use those objects however they will be managed by the frontend taking the burden of that away from the developer – this includes things such as:

window resizing or GPU compatibility checks – all things that would take a lot of time and code to set-up in projects.

3.2. OpenGL

OpenGL abstraction is fairly thin – it provides “ar_graphics_get_gl_context” function to get the OpenGL context from the frontend window. Just like SDL2 abstraction it saves a lot of time and effort for the developer by providing them a pre-made ready OpenGL context making it, so the developer doesn’t have to create it by himself as, for example: the windowing is handled by the frontend itself.

3.3. Frame Buffer

For many projects a simple frame buffer that gets rendered to the screen is the best solution, instead of making the developer implement the frame buffer object by themselves and then use either SDL2 or some graphics API to show it (which can, sometimes, be a surprisingly complicated task) it provides the user with a pre-made object that is easily created and automatically shown on the screen in the frontend. “ar_graphics_create_frame_buffer” can be used to create it, please see documentation for the usage.

4. Unified Controller Model (Input API Abstraction)

Implementation of input is one of most mundane and challenging parts of developing an emulator as they need to support a vast variety of devices that the user has and could want to use for emulation, some might want to use original emulated console controller while others prefer to use modern controllers which do not always match their older counterparts of the console that is emulated – Unified Controller Model is designed to fix that.

Unified Controller Model is an abstraction of the input system, it allows the user to use any device and configure it in a way that they want, this configuration then gets “translated” into a virtual controller device (based on Xbox 360 controller) which is the only device that virtual console developer has to support, significantly affecting the amount of time spent on the input handling.

The whole abstraction revolves around a single function “ar_get_unified_controller_key_status” which takes in an ID of a key from the Unified Controller Model and returns whether it is pressed or not. The user’s input device is completely abstracted from the developer and the user can set-up his device to translate the inputs in any way they want. The only thing that developers need to do is to “map” actions of the virtual console to the buttons on the unified controller model.

I strongly suggest seeing the CHIP8 virtual console that comes with the project to see how it was implemented there (in input thread function symbol) as it might be slightly confusing at the beginning but the benefits of this abstraction and truly immense.

5. Helpers & Other Abstractions

There are many other helper functions and abstractions available in the library, this document only explained the main ones, please see documentation for description of every abstraction and helper available.