# PCSX-Redux

**None**

*None*

*None*

# Table of contents

# 1. Home

Welcome to the PCSX-Redux emulator documentation.

You can get the emulator for various platforms here: https://github.com/grumpycoders/pcsx-redux#where

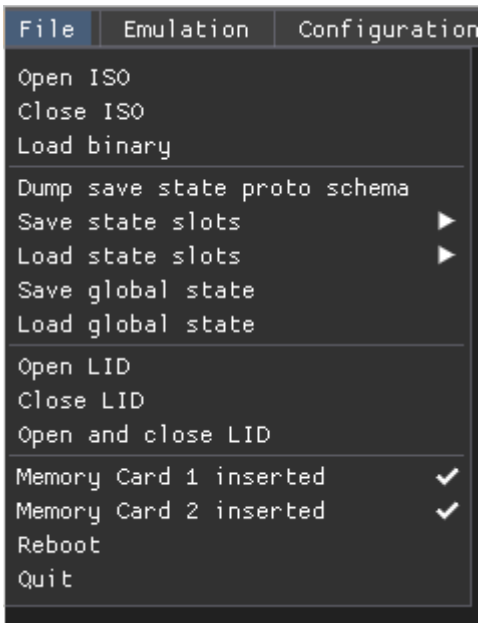
You can find a one page version of this site here: One page version

# 2. PCSX-Redux menus

The menu bar holds some informations :

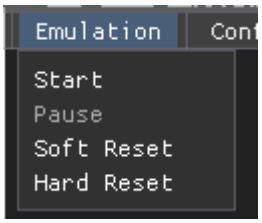

- CPU mode
- Game ID
- ImGui FPS counter (not psx internal fps)

# 2.1 File



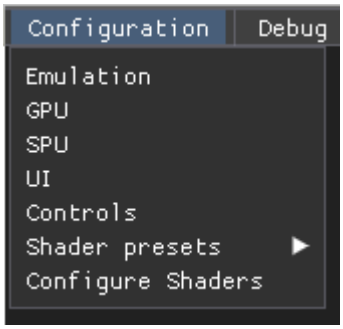

- Open ISO
- Close ISO
- Load Binary
- Dump save state proto schema
- Save state slots
- Load state slots
- Save global state
- Load global state
- Open Lid : Simulate open lid
- Close Lid : Simulate closed lid
- Open and Close Lid : Simulate opening then closing the lid
- MC1 inserted: Insert or remove Memory Card 1
- MC2 inserted: Insert or remove Memory Card 2
- Reboot : Restart emulator
- Quit

## 2.2 Emulation



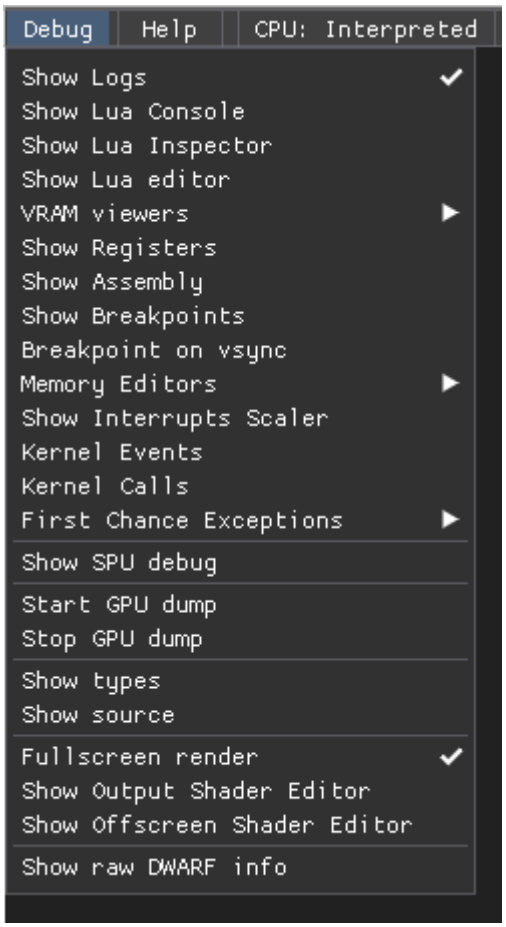

- Start (F5): Start execution
- Pause (F6): Pause execution
- Soft reset (F8): Calls Redux's CPU reset function, which jumps to the BIOS entrypoint (0xBFC00000), resets some COP0 registers and the general purpose registers, and resets some IO. Does not clear vram.
- Hard reset (Shift-F8): Similar to a reboot of the PSX.

## 2.3 Configuration




- Emulation : Emulation settings
- GPU : Graphics Processing Unit settings
- SPU : Sound Processing Unit settings
- UI : Change user interface settings (such as font size, language or UI theme)
- Controls : Edit KB/Pad controls
- Shader presets : Apply a shader preset
- Configure shaders : Show shader editor

## 2.4 Debug

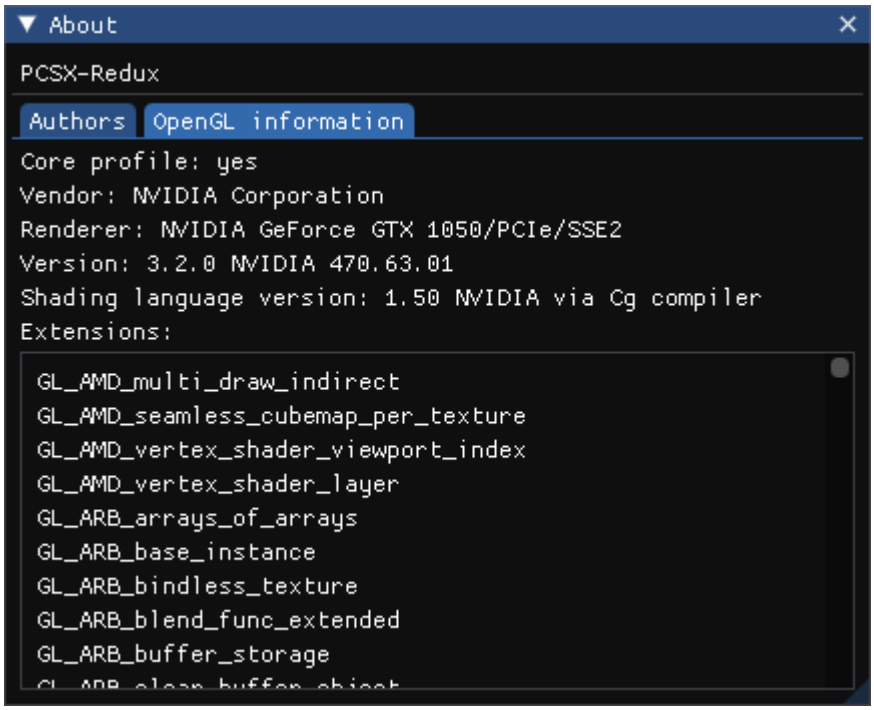



## 2.5 Help

- Show ImGui demo
- About

## 2.6 GPU information

The 'About' dialog available in the 'Help' menu has an 'OpenGL information' tab that displays information on the GPU currently used by the program, such as the supported OpenGL extensions.


About
PCSX-Redux
Authors
OpenGL information
Core profile: yes
Vendor: NVIDIA Corporation
Renderer: NVIDIA GeForce GTX 1050/PCIe/SSE2
Version: 3.2.0 NVIDIA 470.63.01
Shading language version: 1.50 NVIDIA via Cg compiler
Extensions:
GL_AMD_multi_draw_indirect
GL_AMD_seamless_cubemap_per_texture
GL_AMD_vertex_shader_viewport_index
GL_AMD_vertex_shader_layer
GL_ARB_arrays_of_arrays
GL_ARB_base_instance
GL_ARB_bindless_texture
GL_ARB_blend_func_extended
GL_ARB_buffer_storage

# 3. Compiling PCSX-Redux

## 3.1 Getting the sources

The only location for the source is on github. Clone recursively, as the project uses submodules:

`git clone https://github.com/grumpycoders/pcsx-redux.git --recursive`.

## 3.2 Windows

Install Visual Studio 2019 Community Edition.
Open the file `vsprojects\pcsx-redux.sln`, select `pcsx-redux -> pcsx-redux`, right click, `Set as Startup Project`, and hit `F7` to build.
The project follows the open-and-build paradigm with no extra step, so no specific dependency ought to be needed, as NuGet will take care of downloading them automatically for you on the first build.

Note: If you get an error saying `hresult e_fail has been returned from a call to a com component`, you might need to delete the .suo file in vsproject/vs, restart Visual Studio and retry.

#### Openbios

Using Visual Studio Code, one can use the task "make_openbios" to compile: CTRL-P then `task make_openbios` to compile.

# 3.3 Linux

## 3.3.1 Compiling with Docker

Run `./dockermake.sh`. You need docker for this to work.

```
# Debian derivative; Ubuntu, Mint...
sudo apt install docker
# Arch derivative; Manjaro...
sudo pacman -S docker
```

You will also need a few libraries on your system for this to work. Check the Dockerfile for a list of library packages to install.

## 3.3.2 Compiling with make

- Debian derivatives ( for full emulator compilation ):

```
sudo apt-get install -y build-essential git make pkg-config clang g++ g++-mipsel-linux-gnu cpp-mipsel-linux-gnu binutils-mipsel-linux-gnu libfreetype-dev libavcodec-dev libavformat-dev libavutil-dev libglfw3-dev libswresample-dev libuv1-dev zlib1g-dev
```

- Arch derivatives :

```
sudo pacman -S clang git make pkg-config ffmpeg libuv zlib glfw-x11 curl xorg-server-xvfb
```

You can then just enter the 'pcsx-redux' directory and compile without using docker with `make`.

If you have a different mips compiler, you'll need to override some variables, such as `PREFIX=mipsel-none-elf FORMAT=elf32-littlemips`.

### Openbios

Building OpenBIOS on Linux can be done with docker : `./dockermake.sh openbios`, or using `make`, with the `g++-mipsel-linux-gnu` package installed ; `make openbios`.

### 3.3.3 MacOS

You need MacOS Catalina with the latest XCode to build, as well as a few homebrew packages.
Run the brew installation script to get all the necessary dependencies.

Run `make` to build.

Compiling OpenBIOS will require a mips compiler, that you can generate using the following commands:

#### Openbios

```
brew install ./tools/macos-mips/mipsel-none-elf-binutils.rb
brew install ./tools/macos-mips/mipsel-none-elf-gcc.rb
```

Then, you can compile OpenBIOS using `make -C ./src/mips/openbios`.

## 3.4 Compiling PSX code

If you're only interested in compiling psx code, you can clone the PCSX-Redux repo;

```
git clone https://github.com/grumpycoders/pcsx-redux.git --recursive
```

then install a mips toolchain and get the converted PsyQ libraries in the `pcsx-redux/src/mips/psyq/` folder as per these instructions.

You can also find the pre-compiled converted Psyq libraries online.

### 3.4.1 Getting the toolchain on Windows

Download the MIPS toolchain here : https://static.grumpycoder.net/pixel/mips/g++-mipsel-none-elf-10.3.0.zip
and add the `bin` folder to your $PATH.
You can test it's working by launching a command prompt and typing `mipsel-none-elf-gcc.exe --version`. If you get a message like `mipsel-none-gnu-gcc (GCC) 10.3.0`, then it's working !

## 3.4.2 Getting the toolchain on GNU/Linux

**Debian derivative; Ubuntu, Mint...**

```
sudo apt install g++-mipsel-linux-gnu cpp-mipsel-linux-gnu binutils-mipsel-linux-gnu
```

**Arch derivative; Manjaro...**

The mipsel environment can be installed from AUR : cross-mipsel-linux-gnu-binutils and cross-mipsel-linux-gnu-gcc using your AURhelper of choice:

```
trizen -S cross-mipsel-linux-gnu-binutils cross-mipsel-linux-gnu-gcc
```

# 4. Command Line Flags

You can launch `pcsx-redux` with the following command line parameters:

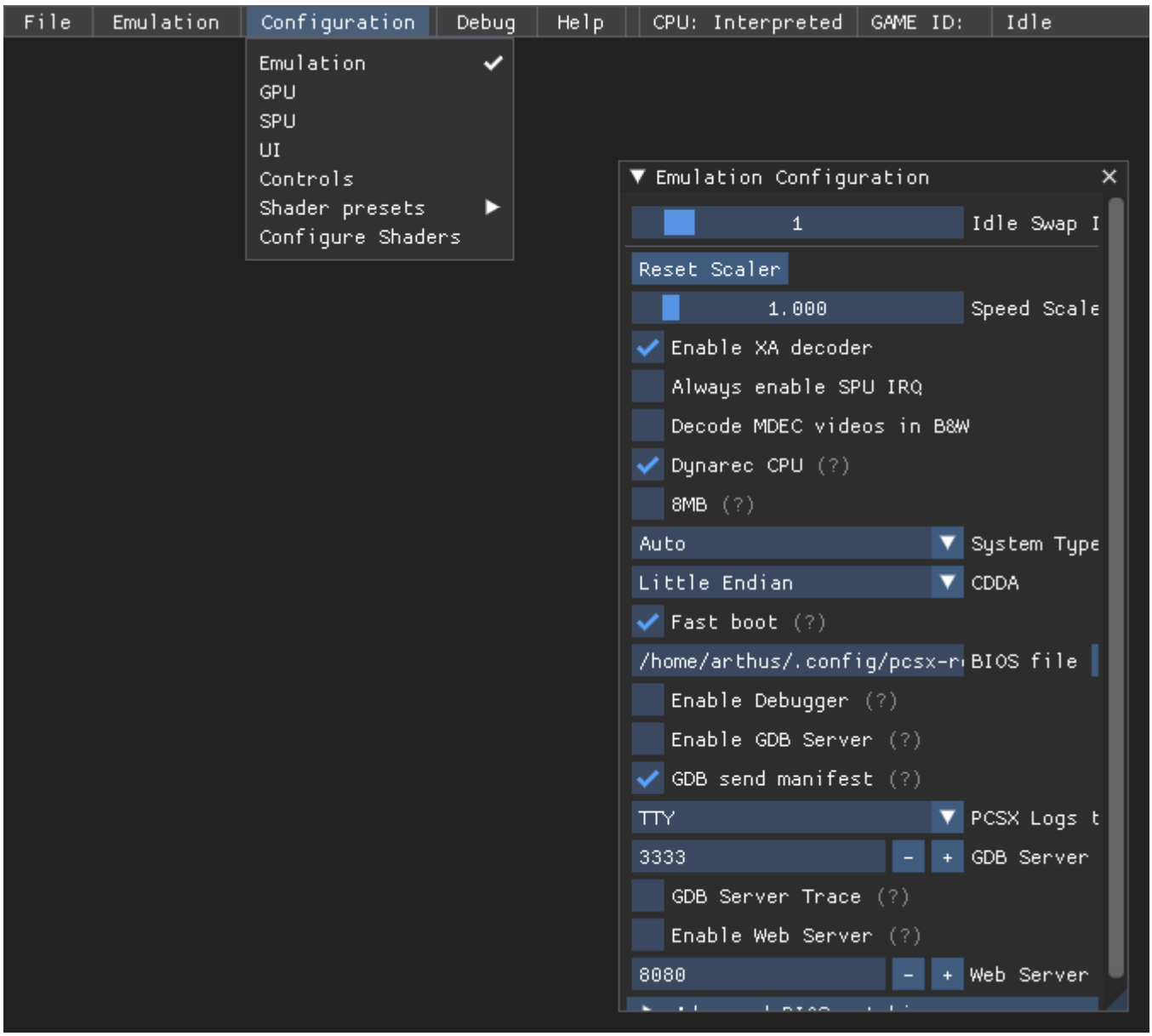
**The parsing code doesn't care about the number of dashes in the parameter's flag, so '-' can be used as well as '--', or any number of dashes.**

| Flag | Meaning |
|---|---|
| `-dumpproto` | Dump the protobuf schemas for PCSX-Redux on stdout and exit immediately. |
| `-run` | Begin execution immediately on startup. |
| `-stdout` | Redirect log output to stdout. |
| `-lua_stdout` | Redirect Lua's console output to stdout. |
| `-logfile` | Specify a file to log output to. |
| `-bios` | Specify a BIOS file. |
| `-testmode` | Interpret internal API's `pcsx_exit()` command and close the emulator. |
| `-exe` | Load a PSX exe. |
| `-loadexe` | Load a PSX exe. |
| `-iso` | Load a PSX disk image (iso, bin/cue). |
| `-loadiso` | Load a PSX disk image (iso, bin/cue). |
| `-memcard1` | Specify a memory card file to use as memory card slot 1. |
| `-memcard2` | Specify a memory card file to use as memory card slot 2. |
| `-pcdrv` | Enable the pcdrv device interface. (Access PC filesystem through SIO). |
| `-pcdrvbase` | Specify base directory for pcdrv. |
| `-safe` | Resets configuration to defaults. |
| `-interpreter` | Use the interpreter CPU core. |
| `-dynarec` | Use the dynamic recompiler CPU core. |

# 5. GDB server

The GDB server allows you to set breakpoints and control your PSX program's execution from your gdb compatible IDE.

## 5.1 Enabling the GDB server




In PCSX-Redux: `Configuration > Emulation > Enable GDB server`.

Make sure the debugger is also enabled.


# 5.2 GDB setup

You need `gdb-multiarch` on your system :

## 5.2.1 Windows

Download a pre-compiled version from here : https://static.grumpycoder.net/pixel/gdb-multiarch-windows/

## 5.2.2 GNU/Linux

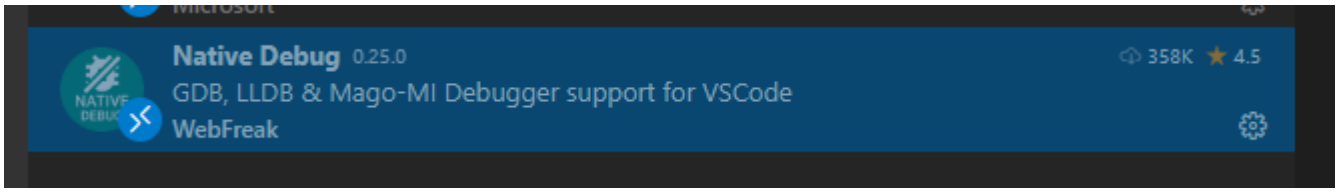
Install via your package manager :

```
# Debian derivative; Ubuntu, Mint...
sudo apt install gdb-multiarch
# Arch derivative; Manjaro
# 'gdb-multiarch' is available in aur : https://aur.archlinux.org/packages/gdb-multiarch/
sudo trizen -S gdb-multiarch
```

# 5.3 IDE setup

## 5.3.1 MS VScode

- Install the `Native debug` extension : https://marketplace.visualstudio.com/items?itemName=webfreak.debug




- Adapt your `launch.json` file to your environment :
  A sample `lanuch.json` file is available here.
  This should go in `your-project/.vscode/`.

You need to adapt the values of `"target"`, `"gdbpath"` and `"autorun"` according to your system :

#### target

This is the path to your `.elf` executable :

```
1       "target": "HelloWorld.elf",
```

https://github.com/NDR008/VSCodePSX/blob/d70658b5ad420685367de4f3c18b89d72535631e/get_started/.vscode/launch.json#L9

#### gdbpath

This the path to the `gdb-multiarch` executable:

```
1       "gdbpath": "/usr/bin/gdb-multiarch",
```

https://github.com/NDR008/VSCodePSX/blob/d70658b5ad420685367de4f3c18b89d72535631e/get_started/.vscode/launch.json#L10

#### autorun

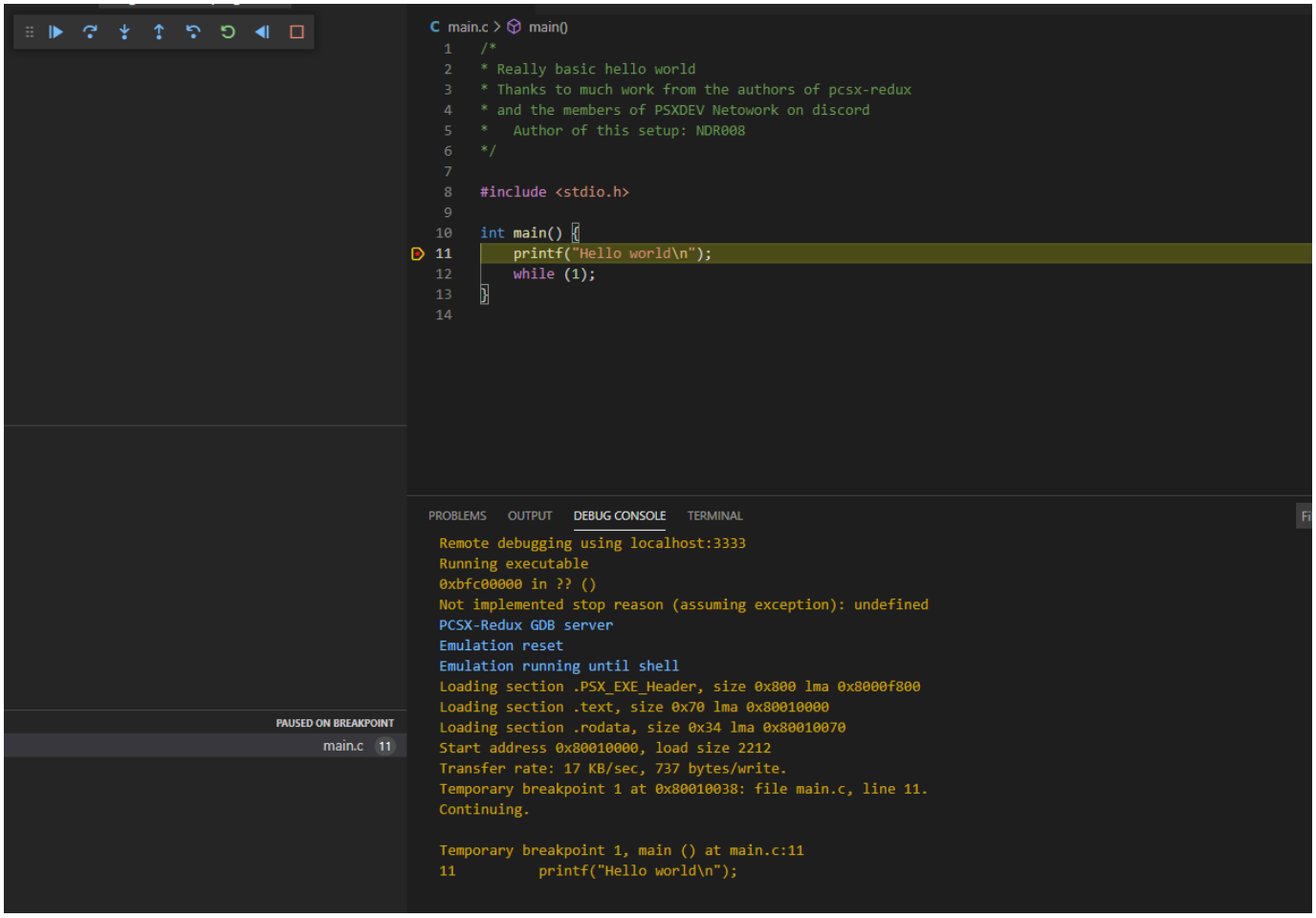
```
1       "autorun": [
2        "target remote localhost:3333",
3        [...]
4        "load HelloWorld.elf",
```

Make sure that `"load your-file.elf"` corresponds to the `"target"` value.

https://github.com/NDR008/VSCodePSX/blob/d70658b5ad420685367de4f3c18b89d72535631e/get_started/.vscode/launch.json#L15

By default, using `localhost` should work, but if encountering trouble, try using your computer's local IP (e.g; 192.168.x.x, 10.0.x.x, etc.)

https://github.com/NDR008/VSCodePSX/blob/d70658b5ad420685367de4f3c18b89d72535631e/get_started/.vscode/launch.json#L13

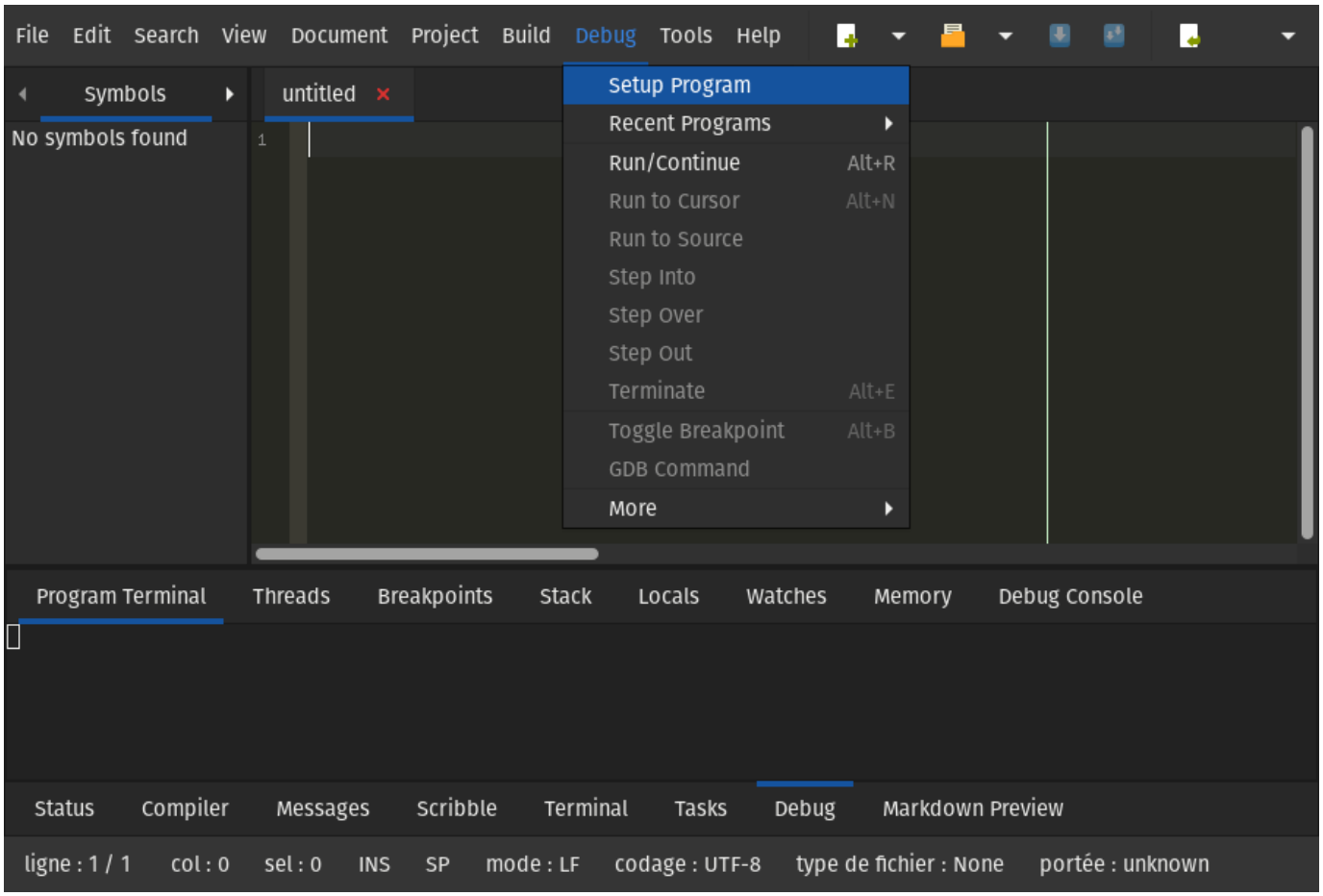



## 5.3.2 Geany

Make sure you installed the official plugins and enable the `Scope debugger`.

To enable the plugin, open Geany, go to `Tools > Plugin manager` and enable `Scope Debugger`.

You can find the debugging facilities in the `Debug` menu ;




You can find the plugin's documentation here : https://plugins.geany.org/scope.html

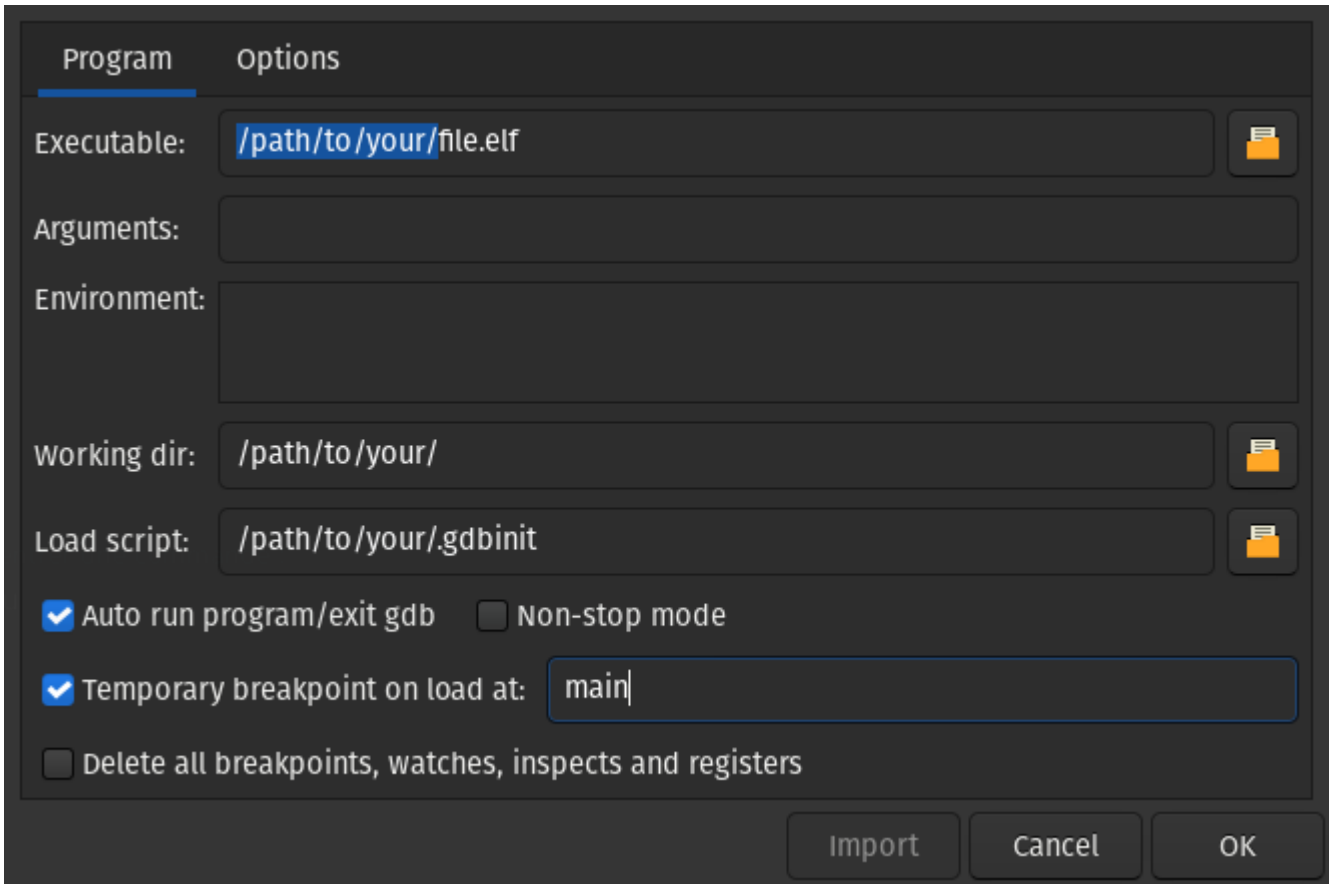
### .gdbinit

Create a `.gdbinit` file at the root of your project with the following content, adapting the path to your `elf` file and the gdb server's ip.

```
target remote localhost:3333
symbol-file load /path/to/your/executable.elf
monitor reset shellhalt
load /path/to/your/executable.elf
```

### Plugin configuration

In Geany : `Debug > Setup Program` :




## 5.4 Beginning Debugging

Launch `pcsx-redux`, then run the debugger from your IDE. It should load the `elf` file, and execute until the next breakpoint.

### 5.4.1 Starting debugging in Geany

Your browser does not support the video tag.

Source :
https://archive.org/details/pcsx_redux_geany_gdb

## 5.5 Additional tools

https://github.com/cyrus-and/gdb-dashboard/

# 6. Mips API

## 6.1 Description

PCSX-Redux has a special API that mips binaries can use :

```
static __inline__ void pcsx_putc(int c) { *((volatile char* const)0x1f802080) = c; }
static __inline__ void pcsx_debugbreak() { *((volatile char* const)0x1f802081) = 0; }
static __inline__ void pcsx_exit(int code) { *((volatile int16_t* const)0x1f802082)
= code; }
static __inline__ void pcsx_message(const char* msg) { *((volatile char**
const)0x1f802084) = msg; }

static __inline__ int pcsx_present() { return *((volatile uint32_t*
const)0x1f802080) == 0x58534350; }
```

Source : https://github.com/grumpycoders/pcsx-redux/blob/main/src/mips/common/hardware/pcsxhw.h#L31-L36

The API needs DEV8/EXP2 (1f802000 to 1f80207f), which holds the hardware register for the bios POST status, to be expanded to 1f8020ff.
Thus the need to use a custom `crt0.s` if you plan on running your code on real hardware.
The default file provided with the Nugget+PsyQ development environment does that:

```
_start:
    lw    $t2, SBUS_DEV8_CTRL
    lui   $t0, 8
    lui   $t1, 1
_check_dev8:
    bge   $t2, $t0, _store_dev8
    nop
    b     _check_dev8
    add   $t2, $t1
_store_dev8:
    sw    $t2, SBUS_DEV8_CTRL
```

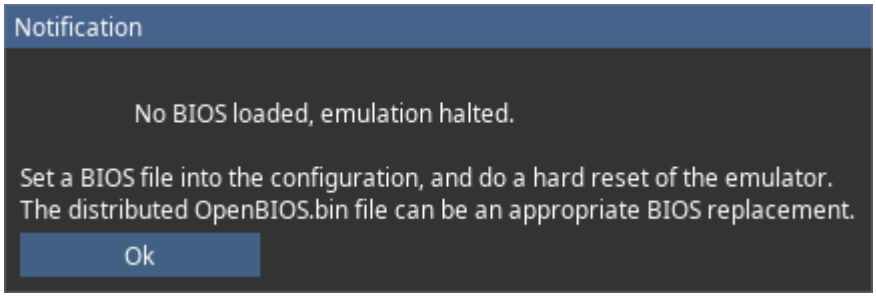
Source : https://github.com/grumpycoders/pcsx-redux/blob/main/src/mips/common/crt0/crt0.s#L36-L46

# 6.2 Functions

The following functions are available :

| Function | Usage |
| --- | --- |
| `pcsx_putc(int c)` | Print ASCII character with code `c` to console/stdout. |
| `pcsx_debugbreak()` | Break execution (Pause emulation). |
| `pcsx_exit(int code)` | Exit emulator and forward `code` as exit code. |
| `pcsx_message(const char* msg)` | Create a UI dialog displaying `msg` |
| `pcsx_present()` | Returns 1 if code is running in PCSX-Redux |

Example of a UI dialog created with `pcsx_message()` :



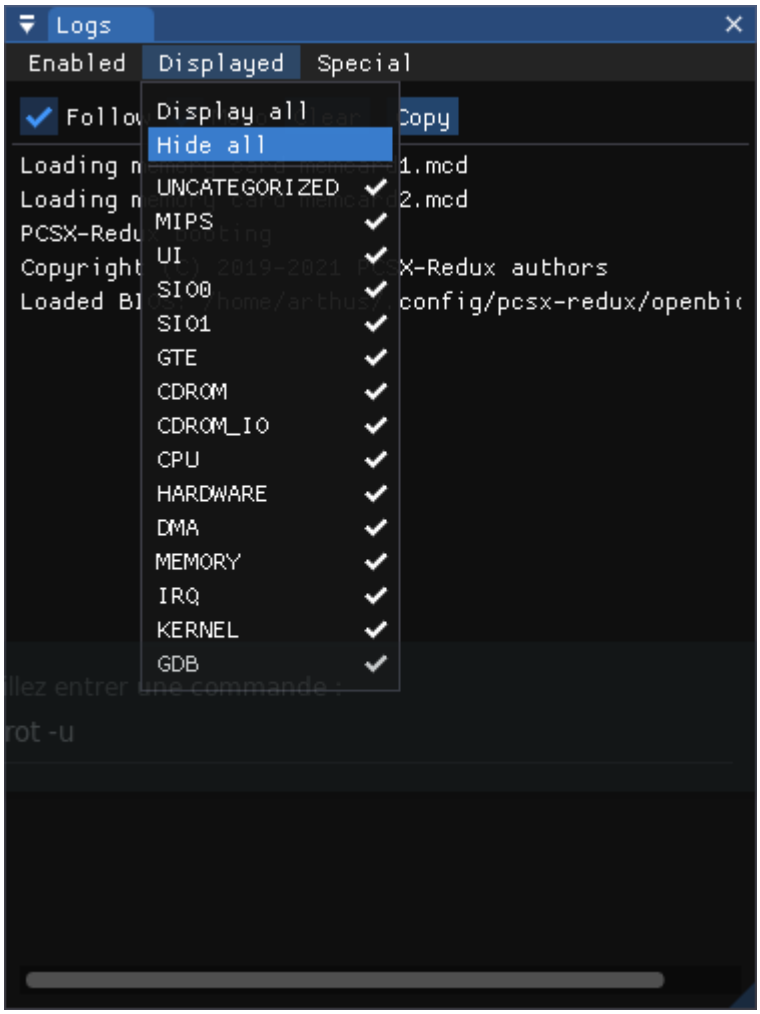
# 7. Dumping a CPU trace to a file

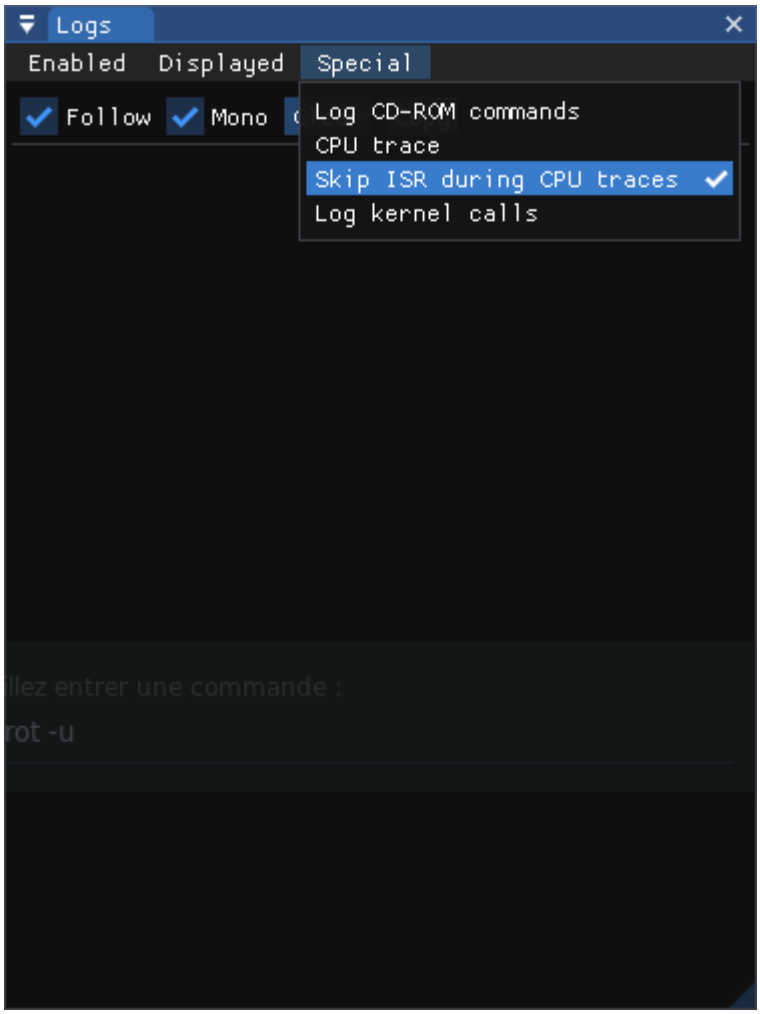
## 7.1 Setup

In PCSX-Redux, make sure `Debug > Show logs` is enabled.

In the 'Logs' window, hide all logs : `Displayed > Hide all`

To avoid unnecessary noise, you can also skip ISR during CPU traces : `Special > Skip ISR during CPU traces`


Logs
Enabled
Displayed
Special
Display all
Hide all
UNCATEGORIZED
MIPS
UI
SIO0
SIO1
GTE
CDROM
CDROM_IO
CPU
HARDWARE
DMA
MEMORY
IRQ
KERNEL
GDB
Copy




## 7.2 Begin dump

To dump the CPU traces, launch pcsx-redux with the following command :

```
pcsx-redux -stdout -logfile log.txt
# Alternatively, you can use -stdout on its own and pipe the output to a file.
pcsx-redux -stdout >> log.txt
```

You can use additional flags to launch an executable/disk image in one go, e.g :

```
pcsx-redux -stdout -logfile tst.log -iso image.cue -run
```

# 7.3 Source

https://discord.com/channels/642647820683444236/663664210525290507/882608398993063997

# 8. Web server

A web server can be activated. This allows the use of a REST api to access various features.

## 8.1 Activation

You can activate the web server by going to `Configuration > Emulation > Enable Web Server`

## 8.2 REST API

By default, the server listens for incoming connection on `localhost:8080`.

| URL | Function |
| --- | --- |
| http://localhost:8080/api/v1/gpu/vram/raw | Dump VRAM |
| http://localhost:8080/api/v1/cpu/ram/raw | Dump RAM |

# 9. Lua API

PCSX-Redux features a Lua API that's available through either a direct Lua console, or a Lua editor, both available through the Debug menu.

## 9.1 Lua engine

The Lua engine that's being used is LuaJIT 2.1.0-beta3. The Lua 5.1 user manual and LuaJIT user manual are recommended reads. In particular, the bindings heavily make use of LuaJIT's FFI capabilities, which allows for direct memory access within the emulator's process. This means there is little protection against dramatic crashes the LuaJIT FFI engine can cause into the emulator's process, and the user must pay extra attention while manipulating FFI objects. Despite that, the code tries as much as possible to sandbox what the Lua code does, and will prevent crashes on any recoverable exception, including OpenGL and ImGui exceptions.

## 9.2 Lua console

All of the messages coming from Lua should display into the Lua console directly. The input text there is a single line execution, so the user can type one-liner Lua statements and get an immediate result.

## 9.3 Lua editor

The editor allows for more complex, multi-line statements to be written, such as complete functions. The editor will by default auto save its contents on the disc under the filename `pcsx.lua`, which can potentially be a problem if the last statement typed crashed the emulator, as it'll be reloaded on the next startup. It might become necessary to either edit the file externally, or simply delete it to recover from this state.

The auto-execution of the editor permits for rapid development loop, with immediate feedback of what's done.

# 9.4 API

## 9.4.1 Basic Lua

The LuaJIT extensions are fully loaded, and can be used globally. Most of the standard Lua libraries are loaded, and are usable. The `require` keyword however doesn't exist at the moment. As a side-effect of Luv, Lua-compat-5.3 is loaded.

## 9.4.2 Dear ImGui

A good portion of ImGui is bound to the Lua environment, and it's possible for the Lua code to emit arbitrary widgets through ImGui. It is advised to consult the user manual of ImGui in order to properly understand how to make use of it. The list of current bindings can be found within the source code. Some usage examples will be provided within the case studies.

## 9.4.3 OpenGL

OpenGL is bound directly to the Lua API through FFI bindings, loosely inspired and adapted from LuaJIT-OpenCL. Some usage examples can be seen in the CRT-Lottes shader configuration page.

## 9.4.4 Luv

For network access and interaction, PCSX-Redux uses libuv internally, and this is exposed to the Lua API through Luv

## 9.4.5 Zlib

The Zlib C-API is exposed through FFI bindings.

## 9.4.6 PCSX-Redux

#### ImGui interaction

PCSX-Redux will periodically try to call the Lua function `DrawImguiFrame` to allow the Lua code to draw some widgets on screen. The function will be called exactly once per actual UI frame draw, which, when the emulator is running, will correspond to the

emulated GPU's vsync. If the function throws an exception however, it will be disabled until recompiled with new code.

### Memory and registers

The Lua code can access the emulated memory and registers directly through some FFI bindings:

- `PCSX.getMemPtr()` will return a `cdata[uint8_t*]` representing up to 8MB of emulated memory. This can be written to, but careful about the emulated i-cache in case code is being written to.
- `PCSX.getRomPtr()` will return a `cdata[uint8_t*]` representing up to 512kB of the BIOS memory space. This can be written to.
- `PCSX.getScratchPtr()` will return a `cdata[uint8_t*]` representing up to 1kB for the scratchpad memory space.
- `PCSX.getRegisters()` will return a structured cdata representing all the registers present in the CPU:

```
typedef union {
    struct {
        uint32_t r0, at, v0, v1, a0, a1, a2, a3;
        uint32_t t0, t1, t2, t3, t4, t5, t6, t7;
        uint32_t s0, s1, s2, s3, s4, s5, s6, s7;
        uint32_t t8, t9, k0, k1, gp, sp, s8, ra;
        uint32_t lo, hi;
    } n;
    uint32_t r[34];
} psxGPRRegs;

typedef union {
    uint32_t r[32];
} psxCP0Regs;

typedef union {
    uint32_t r[32];
} psxCP2Data;

typedef union {
    uint32_t r[32];
} psxCP2Ctrl;

typedef struct {
    psxGPRRegs GPR;
    psxCP0Regs CP0;
    psxCP2Data CP2D;
    psxCP2Ctrl CP2C;
    uint32_t pc;
} psxRegisters;
```

### Execution flow

The Lua code has the following 4 API functions available to it in order to control the execution flow of the emulator:

- `PCSX.pauseEmulator()`
- `PCSX.resumeEmulator()`
- `PCSX.softResetEmulator()`
- `PCSX.hardResetEmulator()`

### Messages

The globals `print` and `printError` are available, and will display logs in the Lua Console. You can also use `PCSX.log` to display a line in the general Log window. All three functions should behave the way you'd expect from a `print` function in Lua.

### GUI

You can move the cursor within the assembly window and the first memory view using the following two functions:

- `PCSX.GUI.jumpToPC(pc)`
- `PCSX.GUI.jumpToMemory(address[, width])`

### Breakpoints

If the debugger is activated, and while using the interpreter, the Lua code can insert powerful breakpoints using the following API:

```
1   PCSX.addBreakpoint(address, type, width, cause, invoker)
```

**Important**: the return value of this function will be an object that represents the breakpoint itself. If this object gets garbage collected, the corresponding breakpoint will be removed. Thus it is important to store it somewhere that won't get garbage collected right away.

The only mandatory argument is `address`, which will by default place an execution breakpoint at the corresponding address. The second argument `type` is an enum which

can be represented by one of the 3 following strings: `'Exec'`, `'Read'`, `'Write'`, and will set the breakpoint type accordingly. The third argument `width` is the width of the breakpoint, which indicates how many bytes should intersect from the base address with operations done by the emulated CPU in order to actually trigger the breakpoint. The fourth argument `cause` is a string that will be displayed in the logs about why the breakpoint triggered. It will also be displayed in the Breakpoint Debug UI. And the fifth and final argument `invoker` is a Lua function that will be called whenever the breakpoint is triggered. By default, this will simply call `PCSX.pauseEmulator()`. If the invoker returns `false`, the breakpoint will be permanently removed, permitting temporary breakpoints for example.

The returned object will have a few methods attached to it:

- `:disable()`
- `:enable()`
- `:isEnabled()`
- `:remove()`

A removed breakpoint will no longer have any effect whatsoever, and none of its methods will do anything. Remember it is possible for the user to still manually remove a breakpoint from the UI.

# 9.5 Case studies

## 9.5.1 Spyro: Year of the Dragon

By looking up some of the gameshark codes for this game, we can determine the following memory addresses:

- `0x8007582c` is the number of lives.
- `0x80078bbc` is the health of Spyro.
- `0x80075860` is the number of unspent jewels available to the player.
- `0x80075750` is the number of dragons Spyro released so far.

With this, we can build a small UI to visualize and manipulate these values in real time:

```lua
-- Declare a helper function with the following arguments:
--   mem: the ffi object representing the base pointer into the main RAM
--   address: the address of the uint32_t to monitor and mutate
--   name: the label to display in the UI
--   min, max: the minimum and maximum values of the slider
--
-- This function is local as to not pollute the global namespace.
local function doSliderInt(mem, address, name, min, max)
  -- Clamping the address to the actual memory space, essentially
  -- removing the upper bank address using a bitmask. The result
  -- will still be a normal 32-bits value.
  address = bit.band(address, 0x1fffff)
  -- Computing the FFI pointer to the actual uint32_t location.
  -- The result will be a new FFI pointer, directly into the emulator's
  -- memory space, hopefully within normal accessible bounds. The
  -- resulting type will be a cdata[uint8_t*].
  local pointer = mem + address
  -- Casting this pointer to a proper uint32_t pointer.
  pointer = ffi.cast('uint32_t*', pointer)
  -- Reading the value in memory
  local value = pointer[0]
  -- Drawing the ImGui slider
  local changed
  changed, value = imgui.SliderInt(name, value, min, max, '%d')
  -- The ImGui Lua binding will first return a boolean indicating
  -- if the user moved the slider. The second return value will be
  -- the new value of the slider if it changed. Therefore we can
  -- reassign the pointer accordingly.
  if changed then pointer[0] = value end
end

-- Utilizing the DrawImguiFrame periodic function to draw our UI.
-- We are declaring this function global so the emulator can
-- properly call it periodically.
function DrawImguiFrame()
  -- This is typical ImGui paradigm to display a window
  local show = imgui.Begin('Spyro internals', true)
  if not show then imgui.End() return end

  -- Grabbing the pointer to the main RAM, to avoid calling
  -- the function for every pointer we want to change.
  -- Note: it's not a good idea to hold onto this value between
  -- calls to the Lua VM, as the memory can potentially move
  -- within the emulator's memory space.
  local mem = PCSX.getMemPtr()

  -- Now calling our helper function for each of our pointer.
  doSliderInt(mem, 0x8007582c, 'Lives', 0, 9)
  doSliderInt(mem, 0x80078bbc, 'Health', -1, 3)
  doSliderInt(mem, 0x80075860, 'Jewels', 0, 65000)
  doSliderInt(mem, 0x80075750, 'Dragons', 0, 70)

  -- Don't forget to close the ImGui window.
```

```
    imgui.End()
end
```

You can see this code in action in this demo video.

## 9.5.2 Crash Bandicoot

Using exactly the same as above, we can repeat the same sort of cheats for Crash Bandicoot. Note that when the CPU is being emulated, the `DrawImguiFrame` function will be called at least when the emulation is issuing a vsync event. This means that cheat codes that regularly write to memory during vsync can be applied naturally.

```
local function crash_Checkbox(mem, address, name, value, original)
  address = bit.band(address, 0x1fffff)
  local pointer = mem + address
  pointer = ffi.cast('uint32_t*', pointer)
  local changed
  local check
  local tempvalue = pointer[0]
  if tempvalue == original then check = false end
  if tempvalue == value then check = true else check = false end
  changed, check = imgui.Checkbox(name, check)
  if check then pointer[0] = value else pointer[0] = original end
end

function DrawImguiFrame()
  local show = imgui.Begin('Crash Bandicoot Mods', true)
  if not show then imgui.End() return end
  local mem = PCSX.getMemPtr()
  crash_Checkbox(mem, 0x80027f9a, 'Neon Crash', 0x2400, 0x100c00)
  crash_Checkbox(mem, 0x8001ed5a, 'Unlimited Time Aku', 0x0003, 0x3403)
  crash_Checkbox(mem, 0x8001dd0c, 'Walk Mid-Air', 0x0000, 0x8e0200c8)
  crash_Checkbox(mem, 0x800618ec, '99 Lives at Map', 0x6300, 0x0200)
  crash_Checkbox(mem, 0x80061949, 'Unlock all Levels', 0x0020, 0x00)
  crash_Checkbox(mem, 0x80019276, 'Disable Draw Level', 0x20212400, 0x20210c00)
  imgui.End()
end
```

## 9.5.3 Crash Bandicoot - Using Conditional BreakPoints

This example will showcase using the BreakPoints and Assembly UI, as well as using the Lua console to manipulate breakpoints.

Crash Bandicoot 1 has several modes of execution. These modes tell the game what to do, such as which level to load into, or to load back into the map. These modes are passed to the main game loop routine as an argument. Due to this, manually manipulating memory at the right time with the correct value to can be tricky to ensure the desired result.

The game modes are listed here.

In Crash 1, there is a level that was included in the game but cut from the final level selection due to difficulty, 'Stormy Ascent'. This level can be accessed only by manipulating the game mode value that is passed to the main game routine. There is a gameshark code that points us to the memory location and value that needs to be written in order to set the game mode to the Story Ascent level.

- `30011DB0 0022` - This is telling us to write the value 0x0022 at memory location `0x8001db0` 0x0022 is the value of the Stormy Ascent level we want to play.

The issue is that GameShark uses a hook to achieve setting this value at the correct time. We will set up a breakpoint to see where the main game routine is.

Setting the breakpoint can be done through the Breakpoint UI or in the Lua console. There is a link to a video at the bottom of the article showing the entire procedure.

Breakpoints can alternatively be set through the Lua console. In PCSX-Redux top menu, click Debug → Show Lua Console

We are going to add a breakpoint to pause execution when memory address 0x8001db0 is read. This will show where the main game loop is located in memory.

In the Lua console, paste the following hit enter.

```
1   bp = PCSX.addBreakpoint(0x80011db0, 'Read', 1, 'Find main loop')
```

You should see where the breakpoint was added in the Lua console, as well as in the Breakpoints UI. Note that we need to assign the result of the function to a variable to avoid garbage collection.

Now open Debug → Show Assembly

Start the emulator with Crash Bandicoot 1 SCUS94900

Right before the BIOS screen ends, the emulator should pause. In the assembly window we can see a yellow arrow pointing to `0x80042068`. We can see this is a `lw` instruction that is reading a value from `0x8001db0`. This is the main game loop reading the game mode value from memory!

Now that we know where the main game loop is located in memory, we can set a conditional breakpoint to properly set the game mode value when the main game routine is executed.

This breakpoint will be triggered when the main game loop at `0x80042068` is executed, and ensure the value at `0x80011db0` is set to `0x0022`

In the Lua console, paste the following and hit enter.

```
bp = PCSX.addBreakpoint(0x80042068, 'Exec', 4, 'Stormy Ascent', function()
PCSX.getMemPtr()[0x11db0] = 0x22 end)
```

We can now disable/remove our Read breakpoint using the Breakpoints UI, and restart the game. Emulation → Hard Reset

If the Emulator status shows Idle, click Emulation → Start

Once the game starts, instead of loading into the main menu, you should load directly into the Stormy Ascent level.

You can see this in action in this demo video.

# 10. Openbios

Openbios is, as its name implies, an open-source alternative to a retail PSX bios that can be non-trivial to dump.

## 10.1 Purposes of Openbios

- Educational
- Ease of distribution
- Automated testing

See this page for more details.

## 10.2 Building

It is compiled together with `pcsx-redux` or can be compiled on its own.

See the corresponding sections in Compiling for instructions.

The result of the compilation should be a file called `openbios.elf` that contains all useful debugging symbols,
and a file called `openbios.bin` which can be used in emulators or even burned to a chip and placed on a retail console.

## 10.3 Status

This subproject is still under construction, but is fairly functional and usable. OpenBIOS does almost all the same things as the retail BIOS does when booting, and implements most of its features.
Many games are booting and working properly with this code.
It can be used in emulators or on the real console, either while replacing the rom chip, or by using the "cart" build and programming the flash chip of a cheat cart with the result.

## 10.4 Organization

The BIOS is split in two major parts: the low level code for the bios itself, and the shell, which is the binary that's being loaded into memory at boot time by the bios, to display the SONY sound and logo, and has a small utility menu for playing audio discs, or shuffling around memory cards.

While the first part is the main one that's being targeted here, the second one isn't currently present. This may change in the future, but this isn't currently the focus of this project.

The original code was most likely chunked into several sub-projects, that were all linked together like a giant patchwork. This approach is less readable, and for this reason, we're not going to do this.
However this will result in the ROM/RAM split to be less obvious, and slower at times than the original. Tuning of the hot functions is eventually required.

## 10.5 Technicalities

The code has been rewritten based off the reverse engineering of a dump of the BIOS of an american **SCPH-7001** machine. *MD5sum: 1e68c231d0896b7eadcad1d7d8e76129*

The ghidra database for it is currently being hosted on a server, alongside a few other pieces of software being reversed. Contact one of the authors if you want access.

## 10.6 Commentary

The retail PlayStation BIOS code is a constellation of bugs and bad design.
The fact that the retail console boots at all is nothing short of a miracle. Half of the provided libc in the A0 table is buggy.
The BIOS code is barely able to initialize the CD-Rom, and read the game's binary off of it to boot it; anything beyond that will be crippled with bugs.
And this only is viable if you respect a very strict method to create your CD-Rom. The memory card and gamepad code is a steaming-hot heap of human excrement.
The provided GPU stubs are inefficient at best.

The only sane thing that any software running on the PlayStation ought to do is to immediately disable interrupts, grab the function pointer located at *0x00000310* for

`FlushCache`,
in order put it inside a wrapper that disables interrupts before calling it, and then trash the whole memory to install its own code.
The only reason `FlushCache` is required from the retail code is because since the function will unplug the main memory bus off the CPU in order to work, it HAS to run from the *0xbfc* memory map, which will still be connected.
Anything else from the retail code is virtually useless, and shouldn't be relied upon.

## 10.7 Legality

*Disclaimer: the author is not a lawyer, and the following statement hasn't been reviewed by a professional of the law, so the rest of this document cannot be taken as legal advice.*

As explained above, this code has been written using disassembly and reverse engineering of a retail bios the author dumped from a second hand console. The same exact methodology was employed by Connectix for their PS1 bios. The conclusion of their lawsuit, and that of Sega v. Accolade seems to indicate that this project here follows and is impacted by the same doctrine.