# PCSX-Redux

**None**

*None*

*None*

# Table of contents

# 1. Home

Welcome to the PCSX-Redux emulator documentation.

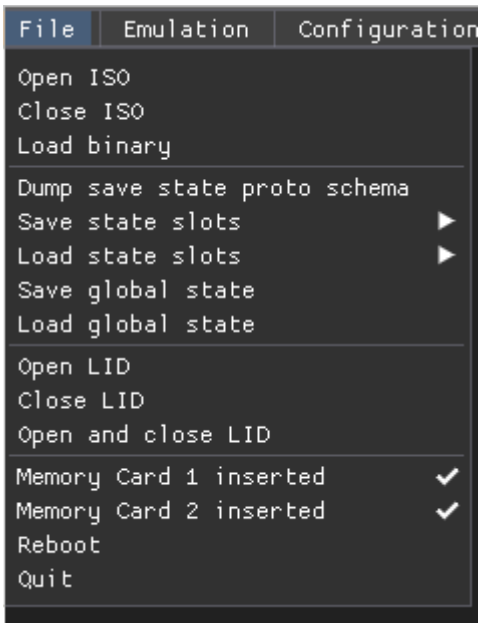
You can get the emulator for various platforms here: https://github.com/grumpycoders/pcsx-redux#where

# 2. PCSX-Redux menus

The menu bar holds some informations :

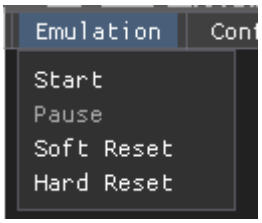

- CPU mode
- Game ID
- ImGui FPS counter (not psx internal fps)

# 2.1 File



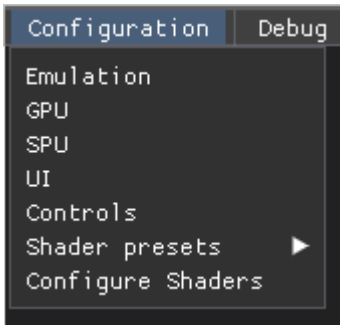

- Open ISO
- Close ISO
- Load Binary
- Dump save state proto schema
- Save state slots
- Load state slots
- Save global state
- Load global state
- Open Lid : Simulate open lid
- Close Lid : Simulate closed lid
- Open and Close Lid : Simulate opening then closing the lid
- MC1 inserted: Insert or remove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

## 2.2 Emulation



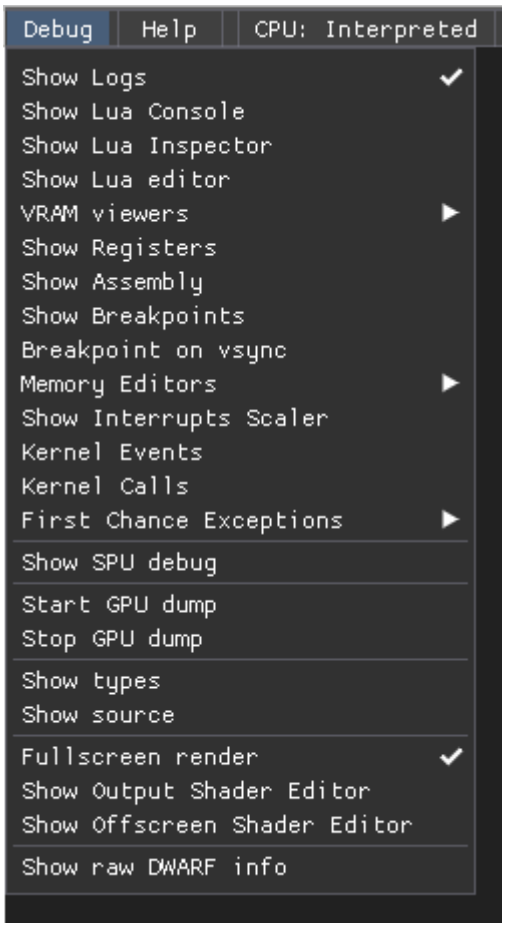

- Start (F5): Start execution
- Pause (F6): Pause execution
- Soft reset (F8): Calls Redux's CPU reset function, which jumps to the BIOS entrypoint (0xBFC00000), resets some COP0 registers and the general purpose registers, and resets some IO. Does not clear vram.
- Hard reset (Shift-F8): Similar to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

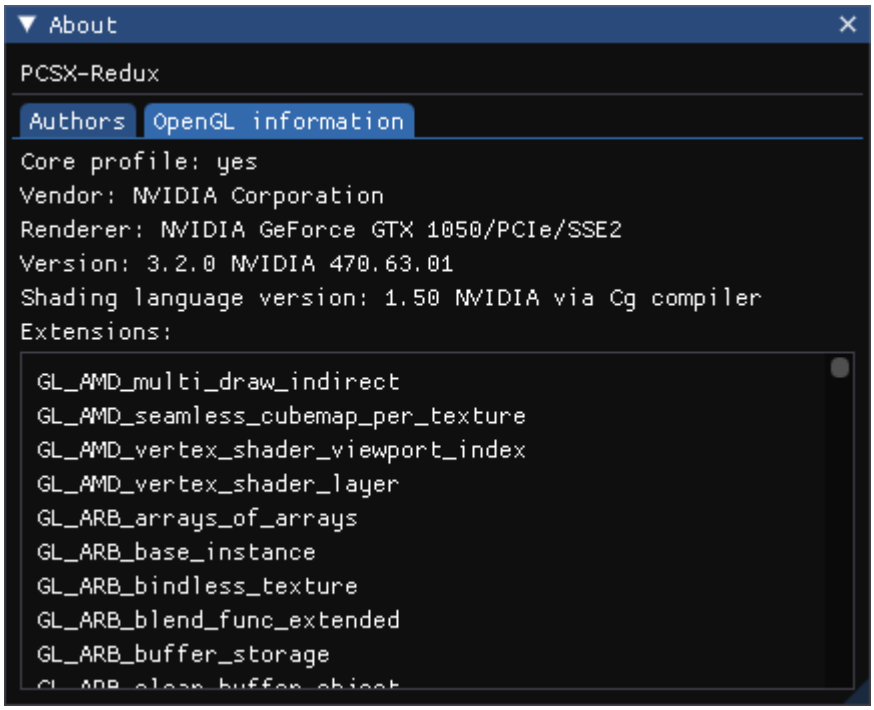



## 2.5 Help

- 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 libcurl4-openssl-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 as a request to exit the emulator instead of pausing, and close the emulator. Implies `-safe`, `-no-gui-log`, and will also disable first chance exceptions. Use only when doing unit testing. |
| `-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. |
| `-resetui` | Resets the UI to its defaults. |
| `-kiosk` | Enables kiosk mode, disabling UI interaction. Will change the saved setting. |
| `-no-kiosk` | Disables kiosk mode, allowing the user to interact with the UI. Will change the saved setting. |
| `-interpreter` | Use the interpreter CPU core. |
| `-dynarec` | Use the dynamic recompiler CPU core. |
| `-debugger` | Activates the debugger. Will change the saved setting. |
| `-no-debugger` | Deactivates the debugger. Will change the saved setting. |
| `-fastboot` | Skips the BIOS logo and boot animation. Will change the saved setting. |
| `-no-fastboot` | Shows the BIOS logo and boot animation. Will change the saved setting. |
| `-gdb` | Activates the gdb server. Will change the saved setting. |
| `-no-gdb` | Deactivates the gdb server. Will change the saved setting. |
| `-gdb-port` | Sets the TCP port the gdb server is listening on. Will change the saved setting. |
| `-trace` | Activates the CPU trace logging. Will change the saved setting. |
| `-no-trace` | Deactivates the CPU trace logging. Will change the saved setting. |
| `-no-gui-log` | Fully disables logs to be sent to the GUI. |

# 5. Debugging

## 5.1 Debugging with PCSX-Redux

PCSX-Redux has strong debugging capabilities. It has a built-in GDB server, which allows you to connect to it with a GDB client, such as gdb itself when targeting MIPS, a vscode connector, IDA Pro, or Ghidra, and debug the MIPS CPU. See debugging with Ghidra for more information on debugging with Ghidra.

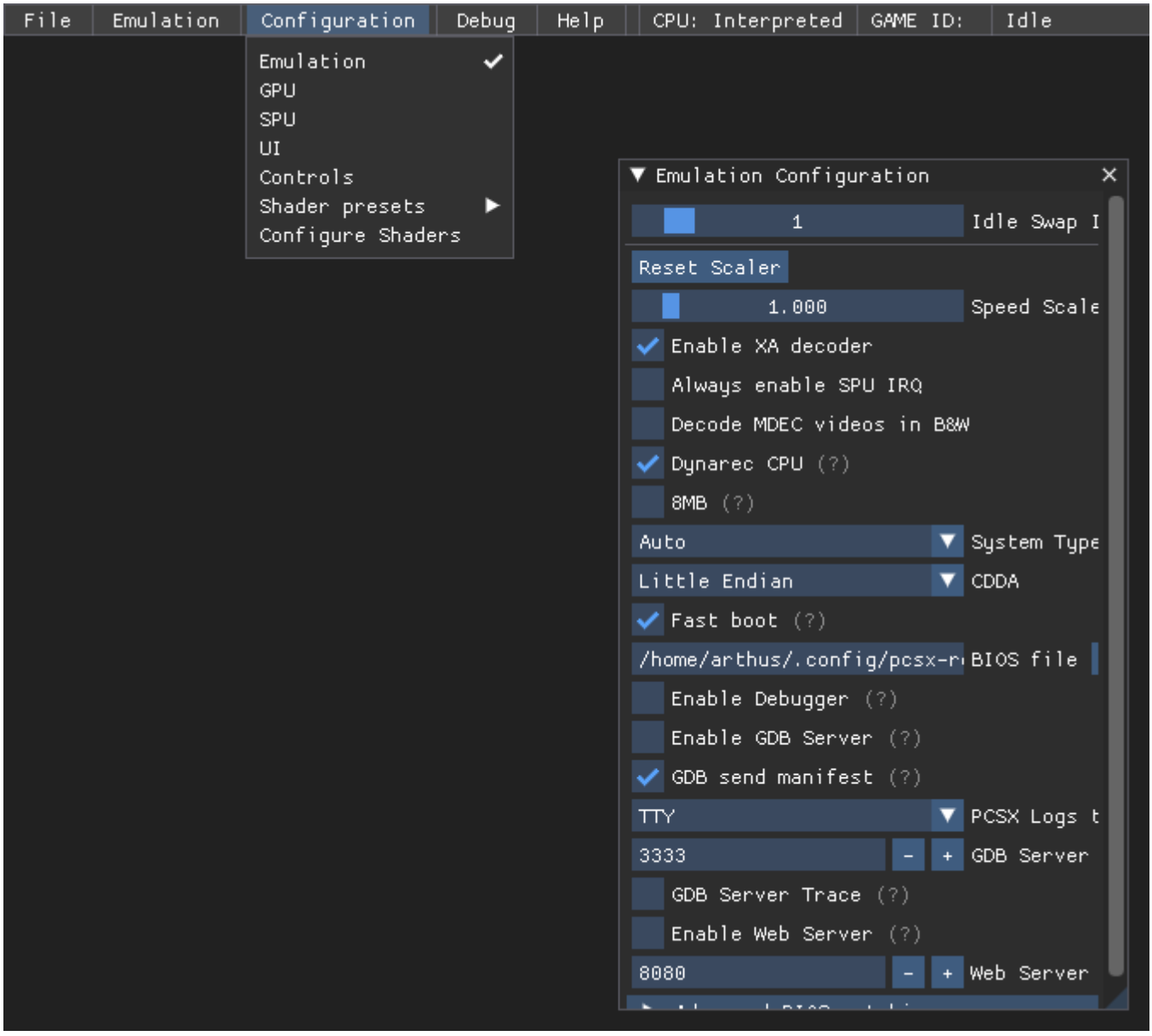
There are also built-in debugging tools, available in the Debug menu. Most of the CPU debugging features will require switching the Dynarec off from the Emulation configuration menu, as the Dynarec is not compatible with the debugging features. Additionally, the debugger needs to be enabled, also in the Emulation configuration menu.

The GPU debugging tools can work with the Dynarec enabled, and thus will be much faster than when the interpreter is used.

# 5.2 GDB server

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

## 5.2.1 Enabling the GDB server




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

Make sure the debugger is also enabled.


## 5.2.2 GDB setup

You need `gdb-multiarch` on your system :

### Windows

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

### GNU/Linux

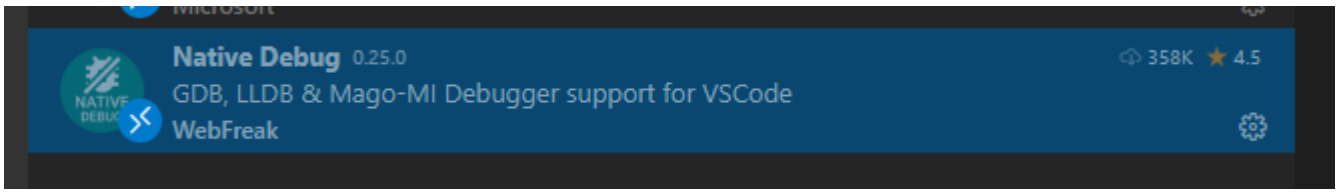
Install via your package manager :

```
# 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
```

## 5.2.3 IDE setup

### 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 `"executable"`, `"gdbpath"` and `"autorun"` according to your system :

#### EXECUTABLE

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

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

https://github.com/grumpycoders/pcsx-redux/blob/a3bebd490388130e924124cdfeff3bc46b6149d9/.vscode/launch.json#L153

#### GDBPATH

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

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

https://github.com/grumpycoders/pcsx-redux/blob/a3bebd490388130e924124cdfeff3bc46b6149d9/.vscode/launch.json#L154-L157

#### AUTORUN

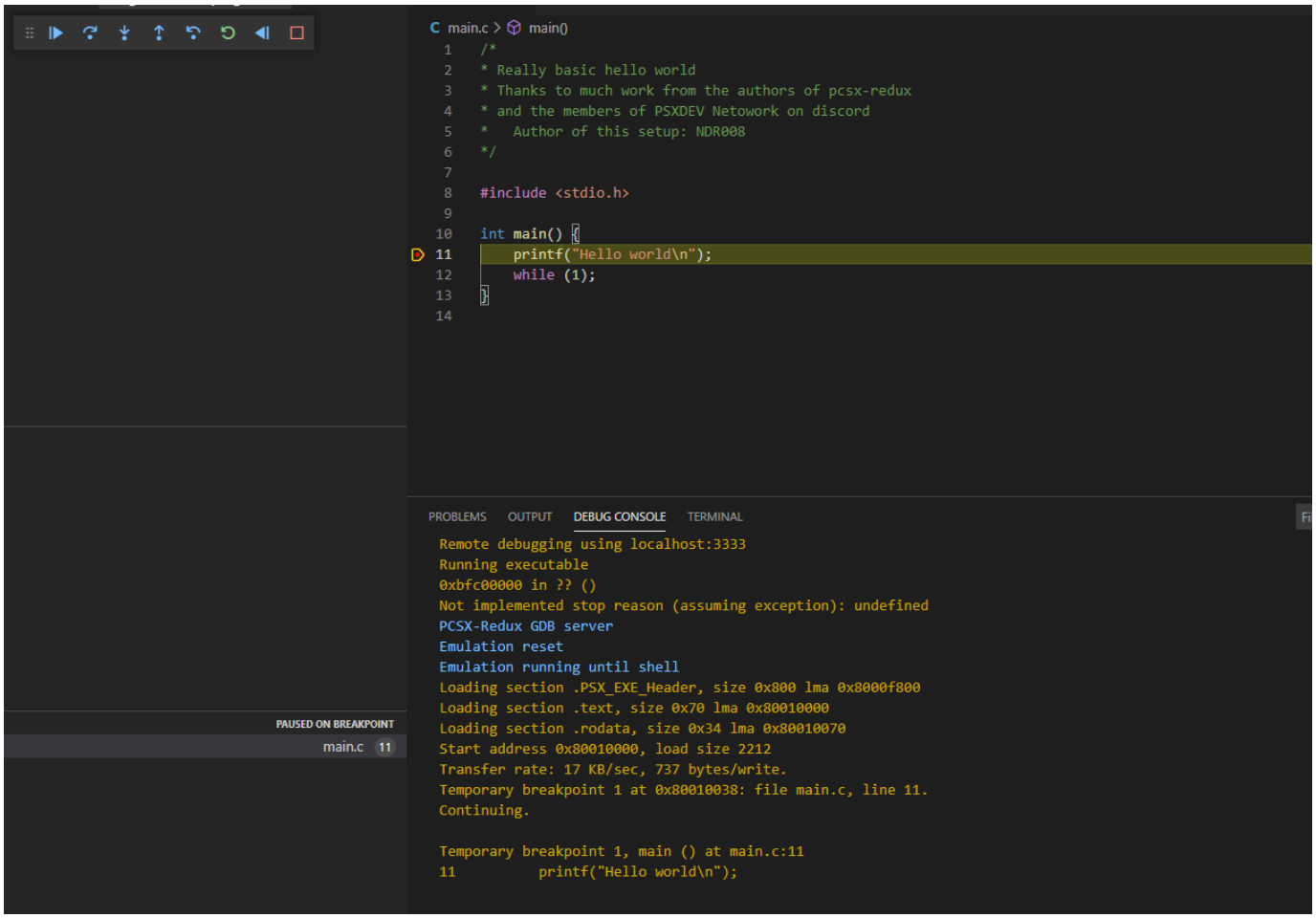
```
1       "autorun": [
2        "monitor reset shellhalt",
3        [...]
4        "load your-file.elf",
```

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

https://github.com/grumpycoders/pcsx-redux/blob/a3bebd490388130e924124cdfeff3bc46b6149d9/.vscode/launch.json#L159-L165

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/grumpycoders/pcsx-redux/blob/a3bebd490388130e924124cdfeff3bc46b6149d9/.vscode/launch.json#L150

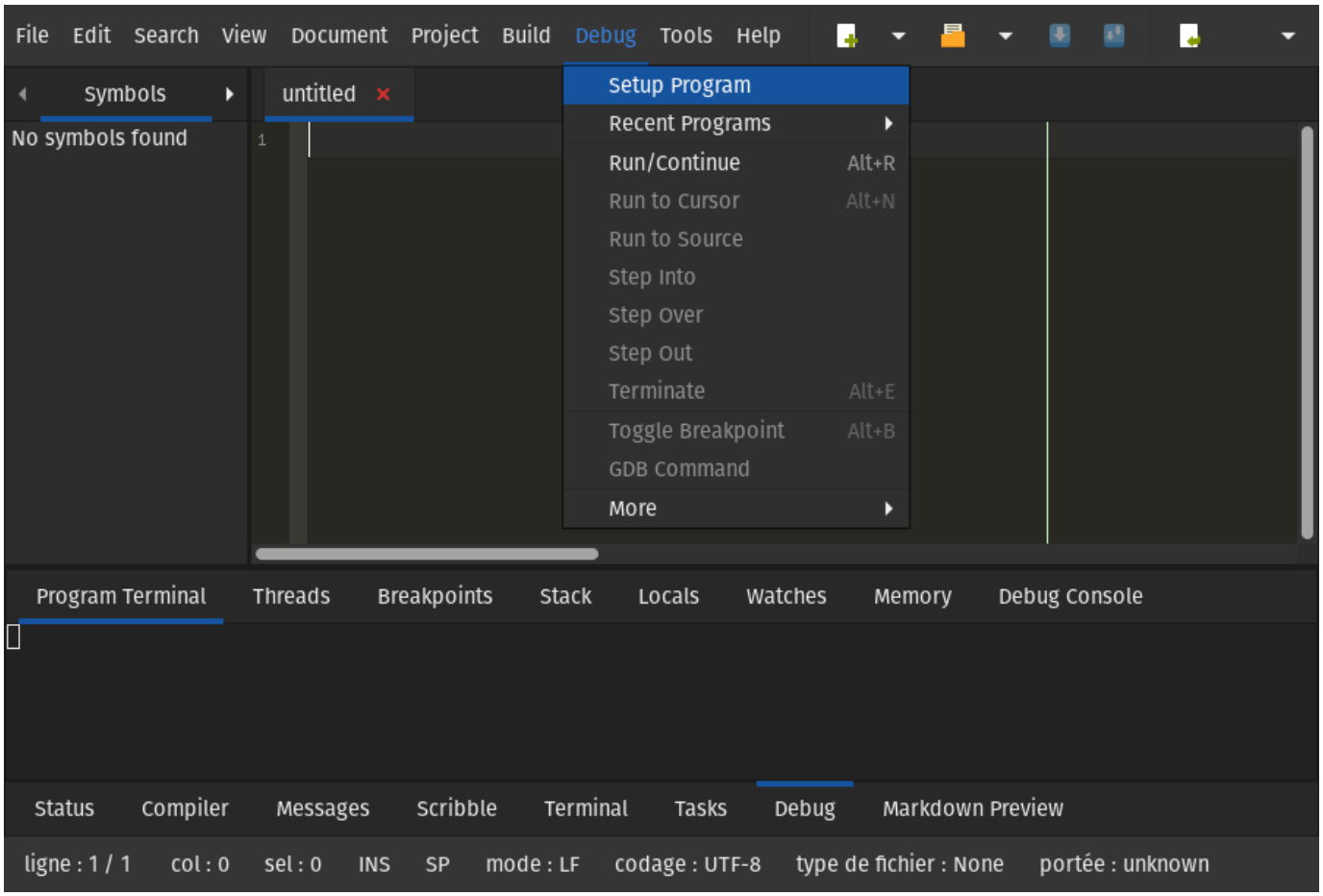



### Geany

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

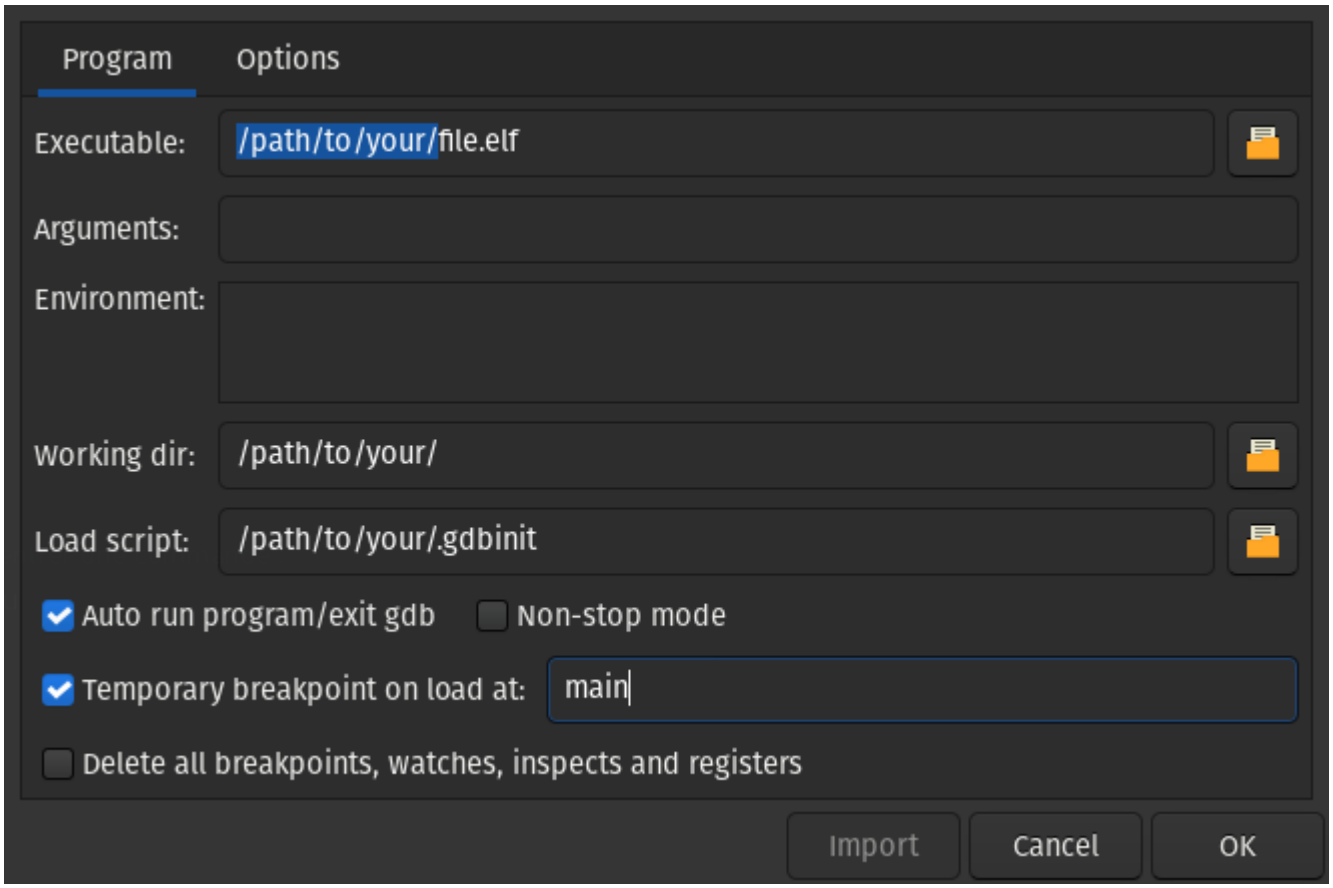
#### .GDBINIT

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
```

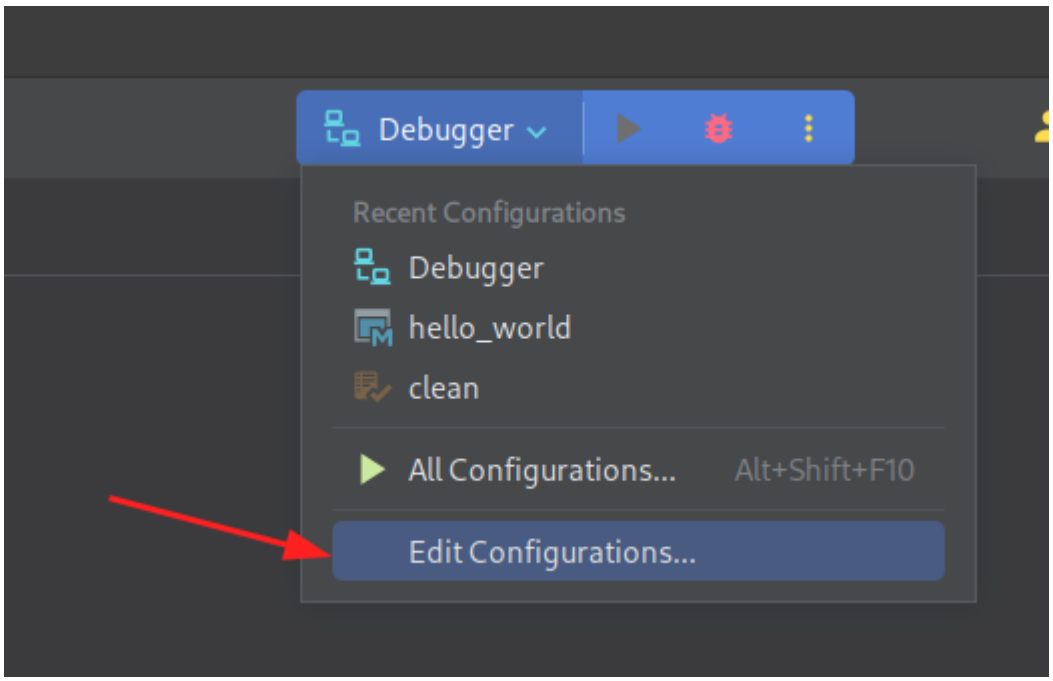
#### PLUGIN CONFIGURATION

In Geany : `Debug > Setup Program` :

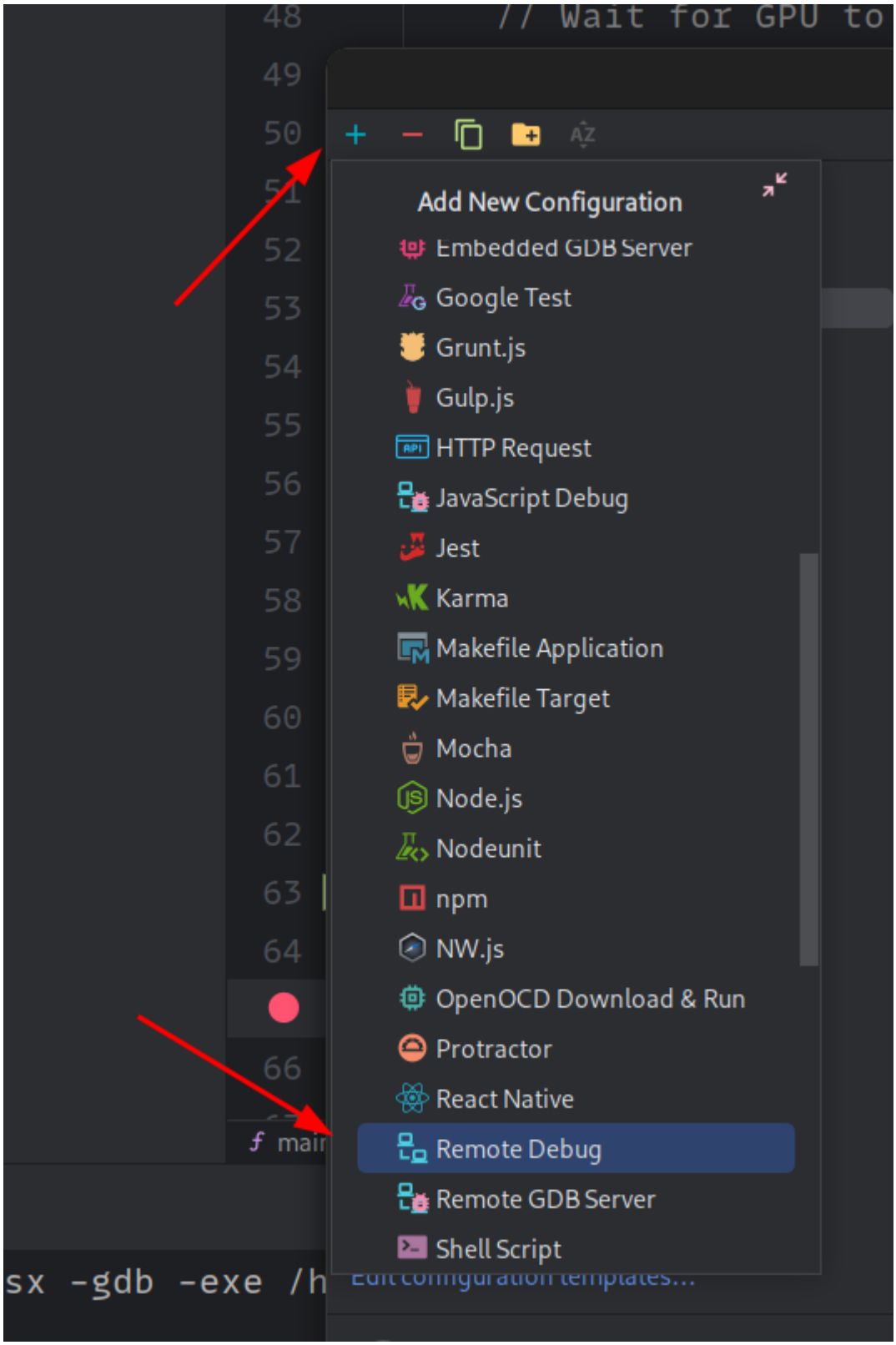



### CLion

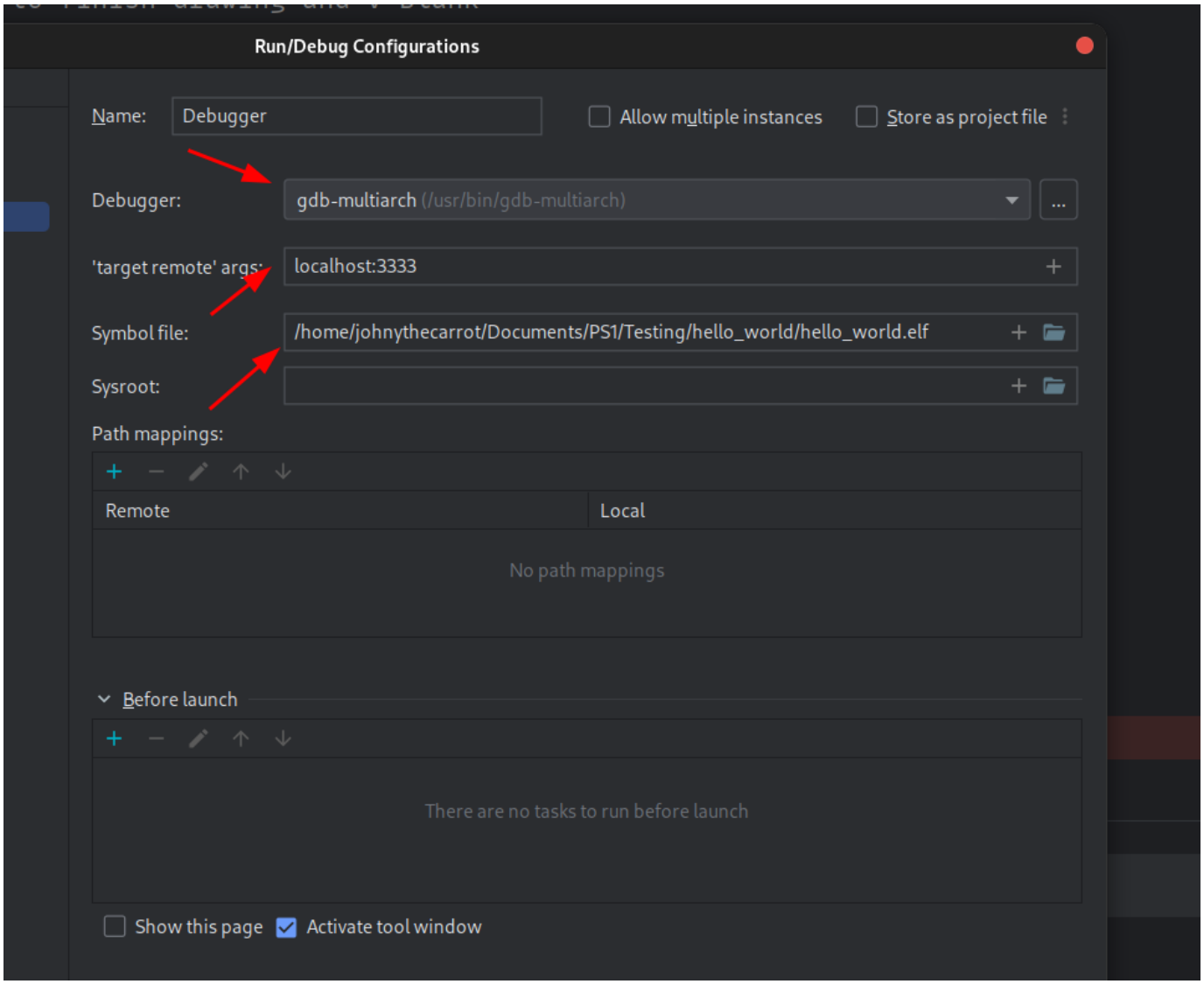
Open the Run/Debug Configurations menu, which you can find here:




Then, add a new Remote Debug configuration:




Finally, set your new configuration up:




#### .GDBINIT

Create a `.gdbinit` file at the root of your project with the following content, adapting the path to your `elf` file.

```
define target remote
target extended-remote $arg0
symbol-file /path/to/your/executable.elf
monitor reset shellhalt
load /path/to/your/executable.elf
end
```

## 5.2.4 Beginning Debugging

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

**Starting debugging in Geany**

Your browser does not support the video tag.

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

## 5.2.5 Additional tools

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

# 5.3 Connecting Ghidra to PCSX-Redux

Since version 10.3, Ghidra now supports debugging MIPS targets. This allows for a much more powerful reverse engineering experience than what was previously possible with the GDB server. This document will explain how to set up Ghidra to debug PCSX-Redux, as it is not entirely straightforward.

## 5.3.1 Prerequisites

- A gdb "multiarch" binary is required. For Windows, you can get it from here. For Linux, you can get it from your distribution's package manager; on Ubuntu and Debian, this is the package `gdb-multiarch`. And for MacOS, you can use the brew package manager to install it; this is the package `gdb`.
- Ghidra 10.3 or newer. You can get it from here.
- PCSX-Redux either configured to disable Dynarec, enable the debugger, and enable the gdb server, or started using the following command-line arguments: `-interpreter -debugger -gdb`.
- The following file downloaded somewhere on your computer.

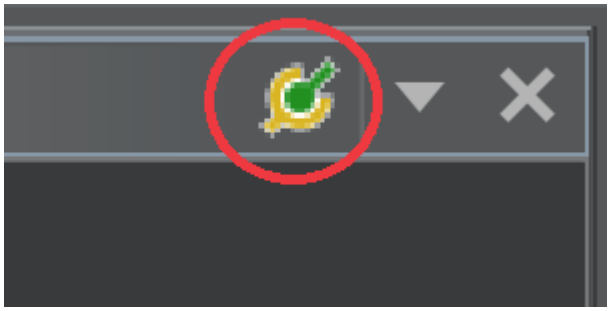
## 5.3.2 Setting up Ghidra

Before starting Ghidra, until version 10.3.3, the MIPS CPU isn't terribly well defined. One needs to go to the installation files of Ghidra, and edit the file `Ghidra/Processors/MIPS/data/languages/mips.ldef`. In this file, find the lines `<external_name tool="gnu" name="mips:4000"/>`, and change them to `<external_name tool="gnu" name="mips:3000"/>`. This will allow Ghidra to properly recognize the MIPS CPU used by the PlayStation 1. This step is no longer necessary starting with Ghidra 10.3.3.

## 5.3.3 Setting up Ghidra's debugger

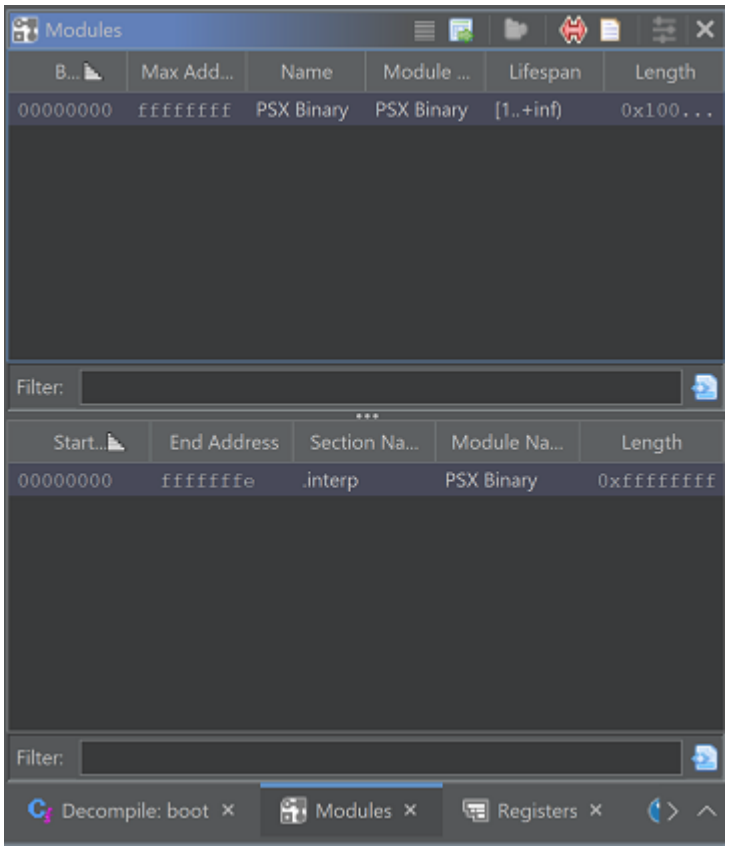
When in the main view of Ghidra, right click on the project you want to debug, and in the context menu, select `Open With > Debugger`. This will open the debugger tool instead of the default disassembler tool.

First, identify the Debugger Targets window, and click its top right button:



This will open the debugger connector window. In the drop down, select `gdb`, and as the launch command, enter the path to the gdb multiarch binary, followed by `-i mi2`. For example, on Windows, this could be `C:/gdb-multiarch/bin/gdb-multiarch.exe -i mi2`. Click `Connect`.

A new `Interpreter` window should open on the right, with the prompt `(gdb)` allowing you to type commands. First, you need to source the `ghidra_debugger_scripts` file from before. To do this, type `source <path to ghidra_debugger_scripts>`. For example, on Windows, this could be `source C:/Users/Pixel/Downloads/ghidra_debugger_scripts`. Then, you need to connect to the PCSX-Redux gdb server. To do this, type `target remote localhost:3333`. Finally, locate the `Modules` tab in the right window, next to the `Interpreter` tab, which should look like this:



Select the top line, right click on it, and in the context menu, select `Map Module to <name of your project>`. In the new window that appears, simply click `Ok`.

At this point, Ghidra should be fully connected to PCSX-Redux, and should be able to place breakpoint, resume or pause execution, inspect variables, etc. Please be aware that, as of Ghidra 10.3, many features of the debugger are still work in progress, and won't necessarily be stable.

# 5.4 Misc Features

## 5.4.1 Mapping breakpoints

PCSX-Redux has a feature that allows mapping the memory of the console while the software is running, and to set breakpoints on the mapped memory. This can for instance help in finding codepath when performing certain activities when running code.

First, map the kind of action you want to discover, such as executing code, reading memory, or writing memory. Then, run the code for some time without performing the specific action you want to discover. Finally, activate the map breakpoint mode, and then perform the action you want to discover. The breakpoint should be triggered when the action is performed.

For example, say that in a game, you want to know what code is executed when you press the "X" button. First, check the `Map execution` checkbox. Then, run the game for a while without pressing the "X" button. This will map enough of the memory that's being run in a normal way. Finally, activate the `Break on execution map` checkbox, and press the "X" button. If the game takes a new codepath that hasn't been executed yet, the breakpoint should be triggered.

Breakpoints are always checked before mapping the memory, so it's safe to keep both checkboxes on at the same time.

Click the `Clear maps` button to zero out all of the maps, when starting anew.

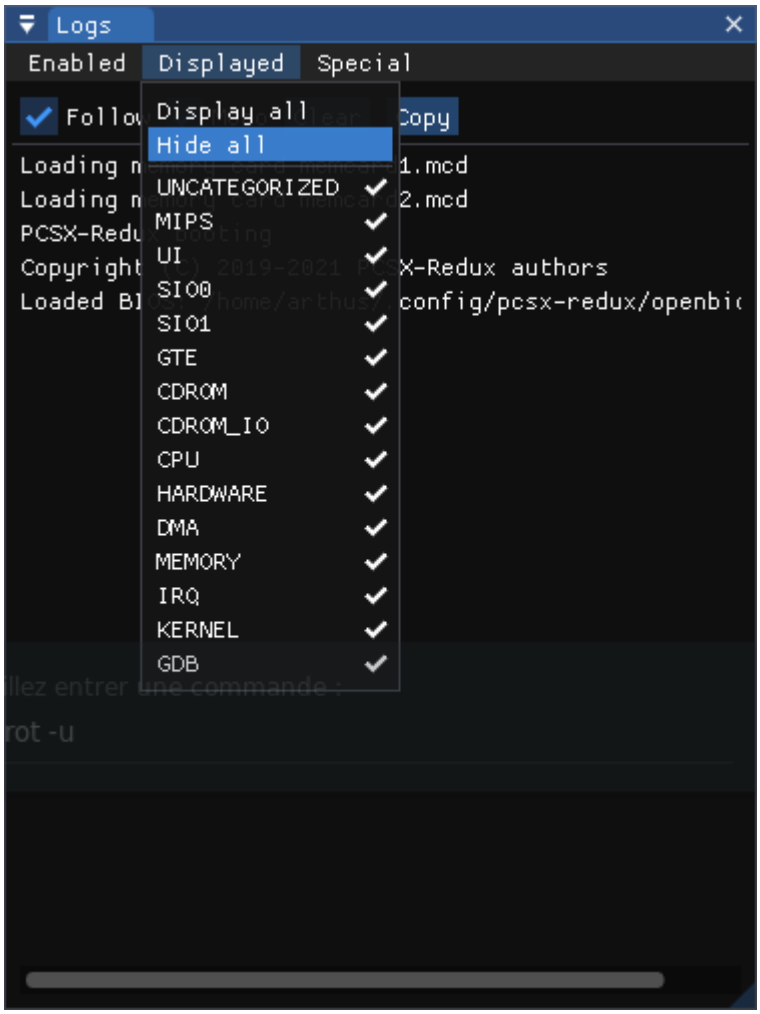
## 5.4.2 CPU trace dump

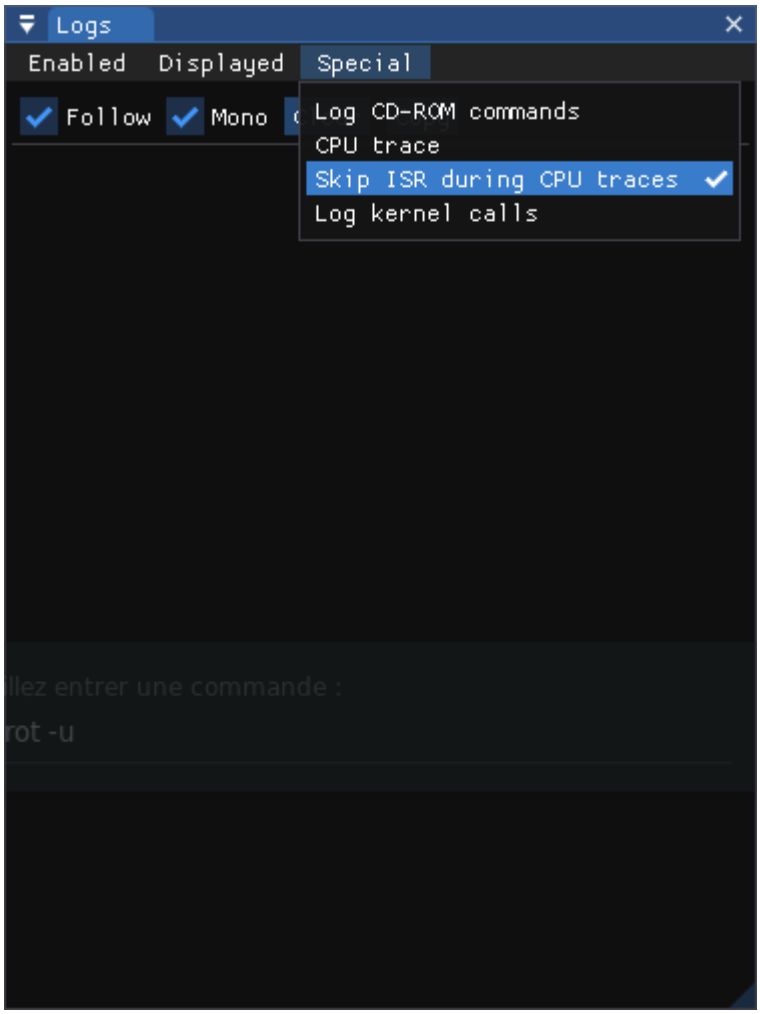
**Setup**

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




### 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
```

### Source

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

# 5.5 VRAM viewer

## 5.5.1 Navigating

Holding the middle button, or both the left and right buttons, allows you to pan the view around. Using the wheel allows you to zoom in and out, at the location of the mouse cursor.

## 5.5.2 Lensing

Holding the CTRL key of your keyboard will bring up a lens, which will show you a locally zoomed version of the VRAM at the location of the mouse cursor. The lens can be resized by using the wheel while holding the CTRL key. Holding the CTRL and Shift buttons while using the wheel will change the size of the lens. The lens can be closed by releasing the CTRL key.

## 5.5.3 The various viewers

There are different viewers available from the main menu, which can be used to visualize the VRAM in different ways. The main viewer will let you see the VRAM using various CLUTs. The CLUT viewer will let you select a CLUT to use for the main VRAM viewer. In order to do this, first select the 8-bits or 4-bits view in the main viewer. Then, in the CLUT viewer, select `View -> Select a CLUT`. At this point, hovering the CLUT viewer will automatically change the main viewer to use the hovered CLUT. Once the proper view is found, simply click on the first pixel of the CLUT viewer to select the CLUT more permanently.

The GPU logger will also select CLUTs and change the main viewer's mode automatically, depending on the GPU commands being inspected.

# 5.6 GPU Logger

The GPU logger is a tool that allows you to see the GPU commands being executed by the emulator, and the resulting VRAM changes. It can be used to debug the GPU, and to understand how the executed software is rendering the scene. The logger will have a full frame worth of primitives, and will automatically clear the log when a new frame is started. Note that the notion of a frame may span over multiple vsyncs, if the PlayStation software isn't running at full FPS.

Note that it can be fairly resource intensive, and may significantly slow down the emulation, depending on the context.

The top of the GPU Logger window will have the following checkboxes:

- GPU Logging - Enable or disable the GPU logging.
- Breakpoint on vsync - Pause the emulation when a vsync occurs, allowing to inspect the current frame.
- Replay frame - Enables the replay of the current frame. See below for details.
- Show origins - Show the data path of the primitives. This will show the origin of the data, and the path it took to reach the GPU. For example, a sequence of primitives may be sent to the GPU via chained DMA.

## 5.6.1 Understanding the logs

The top of the logger can be expanded to display rough frame statistics. These values aren't necessarily too accurate, and are only meant to give a rough idea of the frame complexity.

Each row of the logger displays one command sent to the GPU. The first button and checkbox will be used for the replay system. The next three buttons and checkboxes will be used for the highlighting system. The next column will display the command name, and opening the tree node will expand the command parameters.

The expanded node may have buttons which will affect the main VRAM viewer, either by selecting CLUTs, or zooming in on the corresponding region. The VRAM viewer will also be updated when the replay system is used.

### 5.6.2 Highlighting Primitives

The GPU logger can highlight primitives in the VRAM viewer. One or more primitives may be selected, and the corresponding VRAM regions will be outlined. The highlighting will be cleared when a new frame is started. The default outlined colors will be red for written pixels, and green for read pixels. The colors can be changed in the main VRAM viewer settings.

Checking the `Highlight on hover` checkbox will temporarily outline a primitive when hovering it in the logger. This can be useful to quickly identify the corresponding primitive in the VRAM viewer by flicking the mouse over the logger.

Checking the second checkbox in a logger node will permanently highlight the corresponding primitive in the VRAM viewer. The `[B]` and `[E]` buttons will select the beginning and the end of a span of primitives, and highlight them in the VRAM viewer.

### 5.6.3 Replay System

Once a frame has been logged properly, and the emulator is paused, the replay system can be used to replay the frame. The replay system will constantly replay the frame as long as it is activated, and it will update the VRAM viewer accordingly. By default, all nodes in the logger will be selected for replaying. Unselecting the first checkbox in a node will prevent it from being replayed, and the VRAM viewer will show what happens when this primitive isn't executed, and potentially see what is underneath it. Clicking the `[T]` button of a node will select all nodes for replaying until this node, allowing to easily see the frame being built up to this point.

# 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
```

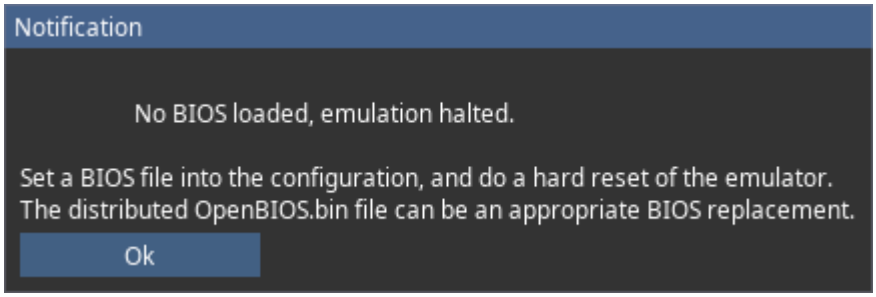
Source : https://github.com/grumpycoders/pcsx-redux/blob/main/src/mips/common/crt0/crt0.s#L36-L46

# 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. Web server

A web server can be activated. This allows the use of a REST api to access various features. The server only handles up to HTTP/1.1, without SSL support.

## 7.1 Activation

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

## 7.2 REST API

By default, the server listens for incoming connection on `localhost:8080`. The port can be changed in the same settings above.

These GET methods are available:

| URL | Function |
|---|---|
| /api/v1/gpu/vram/raw | Dump VRAM |
| /api/v1/cpu/ram/raw | Dump RAM |
| /api/v1/execution-flow | Emulation Status |

The following POST methods are available:

`/api/v1/gpu/vram/raw?x=<value>&y=<value>&width=<value>&height=<value>`

The above needs to also send a form with binary contents. This will partially update the VRAM with the corresponding pixels. The updated rectangle has to be within the 1024x512 16bpp VRAM. The pixels need to be in 16bpp format, meaning the server is expecting exactly `width * height * 2` bytes in the form data. The server will properly parse requests with `Content-Type: multipart/form-data`, but raw bytes in the request body without this header is also acceptable. Any invalid query will result in a 400 error.

`/api/v1/cpu/ram/raw?offset=<value>&size=<value>`

The above needs to also send a form with binary contents, which will update the RAM at the specified offset. Offset is expected to be a number from [0, 0x1FFFFF] in case of running redux with 2MB RAM, or [0, 0x7FFFFF] in case the 8MB memory expansion is enabled. The value of size + offset must not exceed the total space in the RAM.

`/api/v1/assembly/symbols?function=<value>`

| Value | Function |
|---|---|
| reset | Resets the symbols loaded in redux |
| upload | Uploads a `.map` file to redux |

The above expects a `.map` file with symbols and addresses, which will be merged with the current symbols already loaded in redux. The map file should contain a pair of `symbol address` for each line. e.g `Foo 80010000` would load the symbol `Foo` in the address `0x80010000`.

`/api/v1/cpu/cache?function=<value>`

| Value | Function |
|---|---|
| flush | Flushes the CPU cache |

`/api/v1/execution-flow?function=<value>&type=<value>`

| Value | Type | Function |
|---|---|---|
| pause | - | Pauses the emulator |
| start | - | Starts/Resumes the emulator |
| resume | - | Starts/Resumes the emulator |
| reset | hard | Hard resets the emulator |
| reset | soft | Soft resets the emulator |

# 8. Lua

## 8.1 Introduction

PCSX-Redux features a Lua API that is available through either a direct Lua console, or a Lua editor, both available through the Debug menu. The Lua VM runs on the main thread, the same one as the UI and the emulated MIPS CPU. As a result, care must be taken to not stall for too long, or the UI will become unresponsive. Using coroutines to handle long-running tasks is recommended, yielding periodically to let the UI perform some work too. The UI is probably going to run at 60FPS or so, which gives a ballpark of 15ms per frame.

### 8.1.1 Lua engine

The Lua engine that's being used is LuaJIT 2.1.0-beta3 compiled in Lua 5.2 compatibility mode. 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.

### 8.1.2 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.

### 8.1.3 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.

For complex projects however, it is recommended to split your work into sub-modules, and use the `loadfile` function to load them in your main code. This implies working on your project using an external editor.

# 8.2 Loaded libraries

## 8.2.1 Basic Lua

The LuaJIT extensions are fully loaded, and can be used globally. The standard Lua libraries are loaded, and are usable. The `require` function exists, but isn't recommended as the loading of external DLLs might be difficult to properly accomplish. Loading pure Lua files is fine. The `ffi` table is loaded globally, there is no need to `require` it, but it'll work nonetheless. As a side-effect of Luv, Lua-compat-5.3 is loaded.

## 8.2.2 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.

## 8.2.3 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.

## 8.2.4 NanoVG

The NanoVG library is mostly bound to the Lua API through FFI bindings, with some additional glue code. More explanation can be found in the rendering page.

## 8.2.5 Luv

For network access and interaction, PCSX-Redux uses libuv internally, and is exposed to the Lua API through Luv, tho its loop is tied to the main thread one, meaning it'll run only once per frame. There is another layer of network API available through the File API, which is more convenient and faster for simple tasks.

### 8.2.6 Zlib

The Zlib C-API is exposed through FFI bindings. There is another layer of Zlib API available through the File API, which is more convenient and faster for simple tasks.

### 8.2.7 FFI-Reflect

The FFI-Reflect library is loaded globally as the `reflect` symbol. It's able to generate reflection objects for the LuaJIT FFI module.

### 8.2.8 PPrint

The PPrint library is loaded globally as the `pprint` symbol. It's a more powerful `print` function than the one provided by Lua, and can be used to print tables in a more readable way.

### 8.2.9 Lua-Protobuf

The Lua-Protobuf library is available, but not loaded by default. All of its documented API should be usable straight with no additional work. It has been slightly modified, but nothing that should be visible to the user. There is some limited glue between its API and PCSX's.

### 8.2.10 luafilesystem

The luafilesystem library is loaded globally as the `lfs` symbol. It's a library that provides access to the filesystem.

### 8.2.11 LPeg

The LPeg library is available, but not loaded by default. It's a library that provides a pattern-matching library for Lua, which can be useful to create ad-hoc arbitrary parsers.

# 8.3 Redux basic API

## 8.3.1 Settings

All of the settings are exposed to Lua via the `PCSX.settings` table. It contains pseudo-tables that are reflections of the internal objects, and can be used to read and write the settings. The exact list of settings can vary quickly over time, so making a full list here would be fruitless. It is possible however to traverse the settings using `pprint` for example. The semantic of the settings is the same as from within the GUI, with the same caveats. For example, disabling the dynamic recompiler requires a reboot of the emulator.

## 8.3.2 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.

## 8.3.3 Events Engine interaction & Execution Contexts

LuaJIT C callbacks aren't called from a safe execution context that can allow for coroutine resuming, and luv's execution context doesn't have any error handling.

It is possible to defer executing code to the main loop of PCSX-Redux, which can (a) resume coroutines and (b) execute code in a safe context. The function

`PCSX.nextTick(func)` will execute the given function in the next main loop iteration. Here's some examples of how to use it:

```
    local captures = {}
    captures.current = coroutine.running()
    captures.callback = function()
        PCSX.nextTick(function()
            captures.callback:free()
            coroutine.resume(captures.current)
        end)
    end
    captures.callback = ffi.cast('void (*)()', captures.callback)
    -- use the C callback somewhere...
```

```
function createClient(ip, port)
  client = luv.new_tcp()

  luv.tcp_connect(client, ip, port, function (err)
    PCSX.nextTick(function()
      assert(not err, err)

      luv.read_start(client, function (err, chunk)
        PCSX.nextTick(function()
          pprint("received at client", {err=err, chunk=chunk})
          assert(not err, err)
          if chunk then
            -- do something with the client
          else
            luv.close(client)
          end
        end)
      )

      pprint("writing from client")
      luv.write(client, "Hello")
      luv.write(client, "World")

    end
  end)
  return client
end
```

Of course, this can also delay processing significantly, as the main loop is usually bound to the speed of the UI, which can mean up to 20ms of delay.

## 8.3.4 Constants

The table `PCSX.CONSTS` contains numerical constants used throughout the rest of the API. Keeping an up to date list here is too exhausting, and it's simpler to print them using `pprint(PCSX.CONSTS)`.

## 8.3.5 Pads

You can access the pads API through `PCSX.SIO0.slots[s].pads[p]` where `s` is the slot number and `p` is the pad number, both indexed from 1, Lua-style. So `PCSX.SIO0.slots[1].pads[1]` accesses the first pad, and `PCSX.SIO0.slots[2].pads[1]` accesses the second pad.

Each Pad table has the following functions:

```
getButton(button)     -- Returns true if the specified button is pressed.
setOverride(button)   -- Overrides the specified button.
clearOverride(button) -- Clears the override for the specified button.
```

The button constants can be found in `PCSX.CONSTS.PAD.BUTTON`.

You can for instance press the button Down on the first pad using the following code:

```
PCSX.SIO0.slots[1].pads[1].setOverride(PCSX.CONSTS.PAD.BUTTON.DOWN)
```

## 8.3.6 Execution flow

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

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

It's also possible to manipulate savestates using the following functions:

- `PCSX.createSaveState()    -- returns a slice representing the savestate`
- `PCSX.loadSaveState(slice)`
- `PCSX.loadSaveState(file)`

Additionally, the following function returns a string containing the .proto file used to serialize the savestate:

- `PCSX.getSaveStateProtoSchema()`

Note that the actual savestates made from the UI are gzip-compressed, but the functions above don't compress or decompress the data, so if trying to reload a savestate made from the UI, it'll need to be decompressed first, possibly through the zReader File object.

Overall, this means the following is possible:

```
local compiler = require('protoc').new()
local pb = require('pb')

local state = PCSX.createSaveState()
compiler:load(PCSX.getSaveStateProtoSchema())

local decodedState = pb.decode('SaveState', Support.sliceToPBSlice(state))
print(string.format('%08x', decodedState.registers.pc))
```

## 8.3.7 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 the normal `print` function in mainstream Lua.

## 8.3.8 GUI

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

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

## 8.3.9 GPU

You can take a screenshot of the current view of the emulated display using the following:

- `PCSX.GPU.takeScreenShot()`

This will return a struct that has the following fields:

```
struct ScreenShot {
    Slice data;
    uint16_t width, height;
    enum { BPP_16, BPP_24 } bpp;
};
```

The `Slice` will contain the raw bytes of the screenshot data. It's meant to be written out using the `:writeMoveSlice()` method on a `File` object. The `width` and `height` will be the width and height of the screenshot, in pixels. The `bpp` will be either `BPP_16` or `BPP_24`, depending on the color depth of the screenshot. The size of the `data` Slice will be `height * width` multiplied by the number of bytes per pixel, depending on the `bpp`.

## 8.3.10 Miscellaneous

- `PCSX.quit([code])` schedules the emulator to quit. It's not instantaneous, and will only quit after the current block of Lua code has finished executing, which will be before the next main loop iteration. The `code` parameter is optional, and will be the exit code of the emulator. If not specified, it'll default to 0.

# 8.4 Rendering

PCSX-Redux is entirely running as an OpenGL3 application. All of its aspects, including the UI elements, are rendered using OpenGL primitives. This means there is very little boundaries between the various rendered elements on the screen.

The rendering of the UI is done through ImGui, and a chunk of its API is bound is to Lua using bindings.

A good portion of the OpenGL3 API is also bound to Lua, as well as the nanovg library.

## 8.4.1 Emulated GPU rendering pipeline

The content of the Output region is rendered in two steps. The first step is called the "Offscreen rendering", and is done during the emulated GPU vsyncs. Its job is to flush the contents of the VRAM texture to an offscreen texture, which may be of a different resolution. The resolution of the offscreen texture should be pixel perfect with that of the Output region. By default, the associated shader with this operation should only do a simple copy and interpolation, but as the first stage of the rendering pipeline, this can be used for some first pass output effect such as the first pass of a crt shader.

The second step is called the "Output rendering", and is done every time the UI wants to refresh its display, which may or may not be at the same time as the emulated vsync. The resolution of the input will match exactly the resolution of the input texture, and the default shader should simply copy all the texels without any sort of interpolation, but as the second stage of the rendering pipeline, this can still be used for the second pass output effect.

The crt-lottes implementation leverages these two passes to do the full CRT-like output.

## 8.4.2 Shader editor

The shader editor is a simple text editor that allows to edit the shader code. It is not a full IDE, and it is not meant to be. Its point is to do quick iterations on the shader code, and to be able to see the result of the changes in real time.

The shader editor is split in 3 regions:

- The left tab is the vertex shader code. It is technically editable, but there shouldn't be much reason to edit it.
- The middle tab is the fragment shader code. This is the main shader code. It is editable, and the changes will be reflected in real time.
- The right tab is the Lua invoker code. This is the code that will be executed under multiple circumstances. It is editable, and the changes will be reflected in real time.

The Lua invoker code will be compiled and executed in a soft sandbox environment. The code can still access already created globals and mutate them, but any newly created global will be kept within the sandbox and won't be accessible from other Lua code. All these globals will be saved and restored with the normal emulator settings.

When the shaders are compiled, the Vertex and Fragment shader code will be compiled together, and if the resulting program is valid, the Lua invoker code will be compiled and executed. If the Lua code fails to compile or execute, the shader will be considered invalid and the error will be displayed in the shader editor.

This compilation order allows the Lua code to access the shader program uniforms, and to set them up as needed. The global `shaderProgramID` will be available to the Lua code, and will contain the ID of the shader program.

The code is expected to export a few functions:

- `Draw`, which will be called periodically within the ImGui context, allowing to draw UI elements. The global `configureme` will be set to true when the user selects the "Configure Shaders" menu item. This allows to display a configuration UI to the user during this function call.
- `Image(textureID, srcSizeX, srcSizeY, dstSizeX, dstSizeY)`, which will be called periodically within the ImGui context, when the emulator needs to draw the texture `textureID` at the given size. The texture ID is the OpenGL texture ID, and the size is in pixels. The code is at best expected to do a simple call to `imgui.Image(textureID, dstSizeX, dstSizeY, 0, 0, 1, 1)` to draw the texture. For the Emulated GPU Pipeline, this function will only be called on the Output shader, when being drawn to the Output region. As the function will be called during the ImGui context, it can capture certain ImGui state, such as the current ImGui cursor position, and use it to draw additional UI elements. Note that as with any normal ImGui function, this isn't the moment when the UI elements are actually drawn, but rather when the UI elements are queued to be drawn, meaning this isn't when the shader program will be executed, which is the point of the next function.
- `BindAttributes(textureID, shaderProgramID, srcLocX, srcLocY, srcSizeX, srcSizeY, dstSizeX, dstSizeY)` will be called when the shader program is about to be executed, and needs to bind the attributes. The texture ID is the OpenGL texture ID, and the shader program ID is the OpenGL shader program ID. The location and sizes are in pixels, but are only used for the Emulated GPU Pipeline, when the Offscreen shader is being executed, as it needs to grab a portion of the VRAM texture to be rendered to the offscreen texture.

Additionally, it is possible to programmatically set the content of the editors using the following methods:

```
PCSX.GUI.OffscreenShader.setDefaults()
PCSX.GUI.OffscreenShader.setTextVS(text)
PCSX.GUI.OffscreenShader.setTextPS(text)
PCSX.GUI.OffscreenShader.setTextL(text)
PCSX.GUI.OutputShader.setDefaults()
PCSX.GUI.OutputShader.setTextVS(text)
PCSX.GUI.OutputShader.setTextPS(text)
PCSX.GUI.OutputShader.setTextL(text)
```

The `setDefaults` method will set the default shader code, and the `setText*` methods will set the shader code to the given string. The `text` argument can be either an actual string, or a `File` object.

## 8.4.3 ImGui

The ImGui API is bound to Lua, and can be used to draw UI elements. The ImGui API is documented on the ImGui source code. There is also an interactive manual available.

Not all functions are necessarily bound to Lua, and one can check the bindings code to see which functions are bound, and why some functions are not bound.

The main reason for not binding a function is that its arguments or return values are not trivial to bind. For example, the `ImGui::Text` C++ function is not bound, as it takes a variadic number of arguments, which is not possible to bind in Lua easily. Instead, the `ImGui::TextUnformatted` C++ function is bound, which takes a single string argument.

The emulator will periodically try to call the global function `DrawImguiFrame` with no arguments. If the function is not defined, nothing will happen. If the function fails to execute, it will be removed from the global environment, and the emulator will stop trying to call it until a new global is defined.

The `DrawImguiFrame` function is expected to call the `imgui.Begin` function to create a new ImGui window, as there is no default window created by the emulator for the Lua context. The `DrawImguiFrame` function is also expected to call the `imgui.End` function as normal with the ImGui API.

Some extra functions are bound to Lua beyond the API listed above:

- `imgui.extra.ImVec2.New(x, y)` will create a new FFI `ImVec2` object. The `ImVec2` object is a simple struct with two fields, `x` and `y`. The `New` function takes two optional arguments, the `x` and `y` values, and returns the new `ImVec2` object.
- `imgui.extra.getCurrentViewportId()` will return the current viewport ID. Viewports in ImGui are a way to split the ImGui context into multiple independent contexts, and the viewport ID is a unique identifier for each viewport. Basically, each viewport is a physical window from the operating system, and it can contain one or more ImGui windows.
- `imgui.extra.getViewportFlags(id)` will return the viewport flags for the specified viewport. The viewport flags are of the type `ImGuiViewportFlags_` in the ImGui C++ API, and is a bitmask of flags, which are exposed as individual values in the Lua generated bindings.
- `imgui.extra.setViewportFlags(id, flags)` will set the viewport flags for the specified viewport. The proper usage of this function is to call `imgui.extra.getViewportFlags` to get the current flags, modify the flags as needed, and then call `imgui.extra.setViewportFlags` to set the new flags.
- `imgui.extra.getViewportPos(id)` will return the position of the specified viewport. The position is returned as an `ImVec2` object.
- `imgui.extra.getViewportSize(id)` will return the size of the specified viewport. The size is returned as an `ImVec2` object.
- `imgui.extra.getViewportWorkPos(id)` will return the work position of the specified viewport. The work position is returned as an `ImVec2` object.
- `imgui.extra.getViewportWorkSize(id)` will return the work size of the specified viewport. The work size is returned as an `ImVec2` object.
- `imgui.extra.getViewportDpiScale(id)` will return the DPI scale of the specified viewport. The DPI scale is returned as a number. A value of 1.0 means that the DPI scale for this viewport is 100%.

## 8.4.4 NanoVG

The NanoVG library is bound to Lua, and can be used to draw arbitrary vector graphics on top of the emulator. The NanoVG API is documented on the NanoVG source code. The

API is very similar to the HTML5 Canvas API, meaning that one can use the MDN CanvasRenderingContext2D documentation and other related documentation to learn how to use it.

Using an HTML5 canvas toybox like this one is a good way to learn how to use this API safely.

Note that the NanoVG rendering will happen after the ImGui rendering, meaning that the NanoVG rendering will be on top of the ImGui rendering, regardless of the order in which the NanoVG and ImGui functions are called.

Most of the NanoVG API is bound to Lua, with the exception of the following functions:

- `nvgBeginFrame`
- `nvgCancelFrame`
- `nvgEndFrame`
- `nvgCreateImage`
- `nvgCreateImageMem`

In addition, the enums and some constructors for the structures used in NanoVG are available as extra values and functions. Please refer to the Lua source code for more details.

The general idea is that the emulator will call `nvgBeginFrame` and `nvgEndFrame` before and after the Lua code is executed, and the Lua code will be able to call the other functions to draw the vector graphics.

The proper way to use the NanoVG API is to call `nvg:queueNvgRender(function() ... end)`, when in an ImGui window in order to queue the NanoVG rendering for this specific window.

The `nvg:queueNvgRender` function takes a single argument, which is a function that will be called when the NanoVG rendering is being executed. The function will be called without argument.

All of the NanoVG functions are bound to the `nvg` object, which is a proxy object to the proper NanoVG context, meaning it is only valid within the function passed to `nvg:queueNvgRender`.

This allows the user to call the NanoVG functions without having to pass the NanoVG context as the first argument, as it is done automatically by the proxy object.

Note that the font used by the emulator is also loaded into the NanoVG context, meaning that it is possible to use `nvg:Text` without having to load a font first.

## 8.4.5 Example of using everything together

As the NanoVG rendering is very low level, and requires a viewport to draw to, it is required to use the ImGui API to draw some UI, grab the positions of the vector graphics to add, and then queue some NanoVG calls within some ImGui context to draw the wanted vector graphics.

The following example will draw a red rectangle in the middle of the Output region. The rectangle will be 100x100 pixels in size, and will be drawn on top of the emulator rendering. It should follow around the Output region when resizing or moving the window.

In order to work, this example requires the code to be executed in the `Image` function of the Output shader invoker, so we can get the position of the Output region to draw to.

```
function Image(textureID, srcSizeX, srcSizeY, dstSizeX, dstSizeY)
    -- This helper is provided by the emulator, and will properly calculate
    -- arbitrary coordinates within an ImGui image that is dstSizeX x dstSizeY
    -- in size. The first two arguments are the coordinates to convert, and
    -- the middle two arguments are the boundaries of the source image.

    -- Here, we are using (1.0, 1.0) as the source image size, but it could
    -- be any other size, as long as the coordinates are within the boundaries
    -- of the source image. For example, if the source image is 320x240, then
    -- the coordinates should be within (0, 0) and (320, 240), and the helper
    -- will properly convert the coordinates to the destination image size.

    local cx, cy = PCSX.Helpers.UI.imageCoordinates(0.5, 0.5, 1.0, 1.0, dstSizeX,
dstSizeY)

    -- As explained, we can't call NanoVG functions directly, so we need to
    -- queue the rendering of the vector graphics.
    nvg:queueNvgRender(function()
        nvg:beginPath()
        nvg:rect(cx - 50, cy - 50, 100, 100)
        nvg:fillColor(nvg.Color.New(1, 0, 0, 1))
        nvg:fill()
    end)
    imgui.Image(textureID, dstSizeX, dstSizeY, 0, 0, 1, 1)
end
```

# 8.5 File API

## 8.5.1 Introduction & Rationale

While the normal Lua io API is loaded, there's a more powerful API that's more tightly integrated with the rest of the PCSX-Redux File handling code. It's an abstraction class that allows seamless manipulation of various objects using a common API.

The File objects have different properties depending on how they are created and their intention. But generally speaking, the following rules apply:

- Files are reference counted. They will be deleted when the reference count reaches zero. The Lua garbage collector will only decrease the reference count.
- Whenever possible, writes are deferred to an asynchronous thread, making writes return basically instantly. This speed up comes at the trade off of data integrity, which means writes aren't guaranteed to be flushed to the disk yet when the function returns. Data will always have integrity internally within PCSX-Redux however, and when exiting normally, all data will be flushed to the disk.
- Some File objects can be cached. When caching, reads and writes will be done transparently, and the cache will be used instead of the actual file. This will make reads return basically instantly too.
- The Read and Write APIs can haul LuaBuffer objects. These are Lua objects that can be used to read and write data to the file. You can construct one using the `Support.NewLuaBuffer(size)` function. They can be cast to strings, and can be used as a table for reading and writing bytes off of it, in a 0-based fashion. The length operator will return the size of the buffer. The methods `:maxsize()` and `:resize(size)` are available. They also have a `.pbSlice` property that implicitly converts them to a Lua-Protobuf's `pb.slice`, which can then be passed to `pb.decode`.
- The Read and Write APIs can also function using Lua-Protobuf's buffers and slices respectively.
- If the file isn't closed when the file object is destroyed, it'll be closed then, but letting the garbage collector do the closing is not recommended. This is because the garbage collector will only run when the memory pressure is high enough, and the file handle will be held for a long time.
- When using streamed functions, unlike POSIX files handles, there's two distinct seeking pointers: one for reading and one for writing.

## 8.5.2 Common API for all File objects

All File objects have the following API attached to them as methods:

Closes and frees any associated resources. Better to call this manually than letting the garbage collector do it:

```
:close()
```

Reads from the File object and advances the read pointer accordingly. The return value depends on the variant used.

```
:read(size)            -- returns a LuaBuffer
:read(ptr, size)       -- returns the number of bytes read, ptr has to be a cdata of
pointer type
:read(buffer)          -- returns the number of bytes read, and adjusts the buffer's
size
:read(pb_buffer, size) -- returns the number of bytes read, while appending to the
pb_buffer's existing data
```

Reads from the File object at the specified position. No pointers are modified. The return value depends on the variant used, just like the non-At variants above.

```
:readAt(size, pos)
:readAt(ptr, size, pos)
:readAt(buffer, pos)
:readAt(pb_buffer, pos)
```

Writes to the File object. The non-At variants will advances the write pointer accordingly. The At variants will not modify the write pointer, and simply write at the requested location. Returns the number of bytes written. The `string` variants will in fact take any object that can be transformed to a string using `tostring()`.

```
:write(string)
:write(buffer)
:write(slice)
:write(pb_slice)
:write(ptr, size)
:writeAt(string, pos)
:writeAt(buffer, pos)
:writeAt(slice, pos)
:writeAt(pb_slice, pos)
:writeAt(ptr, size, pos)
```

Note that in this context, `pb_slice` and `pb_buffer` refer to Lua-Protobuf's `pb.slice` and `pb.buffer` objects respectively.

Some APIs may return a `Slice` object, which is an opaque buffer coming from C++. The `write` and `writeAt` methods can take a `Slice`. It is possible to write a slice to a file in a zero-copy manner, which will be more efficient:

```
:writeMoveSlice(slice)
:writeAtMoveSlice(slice, pos)
```

After which, the slice will be consumed and not reusable. The `Slice` object is convertible to a string using `tostring()`, and also has two members: `data`, which is a `const void*`, and `size`. Once consumed by the `MoveSlice` variants, the size of a slice will go down to zero.

Finally, it is possible to convert a `Slice` object to a `pb.slice` one using the `Support.sliceToPBSlice` function. However, the same caveats as for normal `pb.slice` objects apply: it is fragile, and will be invalidated if the underlying Slice is moved or destroyed, so it is recommended to use it as a temporary object, such as an argument to `pb.decode`. Still, it is a much faster alternative to calling `tostring()` which will make a copy of the underlying slice.

The following methods manipulate the read and write pointers. All of them return their corresponding pointer. The `wheel` argument can be of the values `'SEEK_SET'`, `'SEEK_CUR'`, and `'SEEK_END'`, and will default to `'SEEK_SET'`.

```
:rSeek(pos[, wheel])
:rTell()
:wSeek(pos[, wheel])
:wTell()
```

These will query the corresponding File object.

```
:size()      -- Returns the size in bytes, if possible. If the file is not seekable,
will throw an error.
:seekable()  -- Returns true if the file is seekable.
:writable()  -- Returns true if the file is writable.
:eof()       -- Returns true if the read pointer is at the end of file.
:failed()    -- Returns true if the file failed in some ways. The File object is
defunct if this is true.
:cacheable() -- Returns true if the file is cacheable.
:caching()   -- Returns true if caching is in progress or completed.
:cacheProgress() -- Returns a value between 0 and 1 indicating the progress of the
caching operation.
```

If applicable, this will start caching the corresponding file in memory.

```
:startCaching()
```

Same as above, but will suspend the current coroutine until the caching is done. Cannot be used with the main thread.

```
:startCachingAndWait()
```

Duplicates the File object. This will re-open the file, and possibly duplicate all ressources associated with it.

```
:dup()
```

Creates a read-only view of the file starting at the specified position, spanning the specified length. The view will be a new File object, and will be a view of the same underlying file. The default values of start and length are 0 and -1 respectively, which will effectively create a view of the entire file. The view may have less features than the underlying file, but will always be seekable, and keep its seeking position independent of the underlying file. The view will hold a reference to the underlying file.

```
:subFile([start[, length]])
```

In addition to the above methods, the File API has these helpers, that'll read or write binary values off their corresponding stream position for the non-At variants, or at the indicated position for the At variants. All the values will be read or stored in Little Endian, regardless of the host's endianness.

```
:readU8(), :readU16(), :readU32(), :readU64(),
:readI8(), :readI16(), :readI32(), :readI64(),
:readU8At(pos), :readU16At(pos), :readU32At(pos), :readU64At(pos),
:readI8At(pos), :readI16At(pos), :readI32At(pos), :readI64At(pos),
:writeU8(val), :writeU16(val), :writeU32(val), :writeU64(val),
:writeI8(val), :writeI16(val), :writeI32(val), :writeI64(val),
:writeU8At(val, pos), :writeU16At(val, pos), :writeU32At(val, pos), :writeU64At(val,
pos),
:writeI8At(val, pos), :writeI16At(val, pos), :writeI32At(val, pos), :writeI64At(val,
pos),
```

## 8.5.3 Creating File objects

The Lua VM can create File objects in different ways:

```
Support.File.open(filename[, type])
Support.File.buffer()
Support.File.buffer(ptr, size[, type])
Support.File.mem4g()
Support.File.uvFifo(address, port)
Support.File.zReader(file[, size[, raw]])
```

### Basic files

The `open` function will function on filesystem and network URLs, while the `buffer` function will generate a memory-only File object that's fully readable, writable, and seekable. The `type` argument of the `open` function will determine what happens exactly. It's a string that can have the following values:

- `READ`: Opens the file for reading only. Will fail if the file does not exist. This is the default type.
- `TRUNCATE`: Opens the file for reading and writing. If the file does not exist, it will be created. If it does exist, it will be truncated to 0 size.
- `CREATE`: Opens the file for reading and writing. If the file does not exist, it will be created. If it does exist, it will be left untouched.
- `READWRITE`: Opens the file for reading and writing. Will fail if the file does not exist.
- `DOWNLOAD_URL`: Opens the file for reading only. Will immediately start downloading the file from the network. The `filename` argument will be treated as a URL. The curl is the backend for this feature, and its url schemes are supported. The progress of the download can be monitored with the `:cacheProgress()` method.
- `DOWNLOAD_URL_AND_WAIT`: As above, but suspends the current coroutine until the download is done. Cannot be used with the main thread.

#### Buffers

When calling `.buffer()` with no argument, this will create an empty read-write buffer. When calling it with a cdata pointer and a size, this will have the following behavior, depending on type:

- `READWRITE` (or no type): The memory passed as an argument will be copied first.
- `READ`: The memory passed as an argument will be referenced, and the lifespan of said memory needs to outlast the File object. The File object will be read-only.
- `ACQUIRE`: It will acquire the pointer passed as an argument, and free it later using `free()`, meaning it needs to have been allocated using `malloc()` in the first place.

The `.mem4g()` constructor will return a sparse buffer that has a virtual 4GB span. It can be used to read and write data in the 4GB range, but will not actually allocate any memory until the data is actually written to. This is useful for doing operations that are similar to that of the PlayStation memory. The `.mem4g()` constructor will return a File object that's fully readable, writable, and seekable. Its size will always be 4GB. The returned object will have 3 additional methods:

- `:lowestAddress()`: Returns the lowest address that has been written to.
- `:highestAddress()`: Returns the highest address that has been written to.
- `:actualSize()`: Returns the size of the buffer, which is the highest address minus the lowest address.

This is a useful object to use with the `:subFile()` method, as it will allow you to create a view of a specific range of the 4GB memory. Specifically, `obj:subFile(obj:lowestAddress(), obj:actualSize())` will create a view of the entire memory that has been written to.

#### Network streams

The `uvFifo` function will create a File object that will read from and write to the specified TCP address and port after connecting to it. The `:failed()` method will return true in case of a connection failure. The address is a string, and must be a strict IP address, no hostnames allowed. The port is a number between 1 and 65535 inclusive. As the name suggests, this object is a FIFO, meaning that incoming bytes will be consumed by any read operation. The `:size()` method will return the number of bytes in the FIFO. Writes will be immediately sent over. There are no reception guarantees, as

the other side might have disconnected at any point. The `:eof()` method will return true when the opposite end of the stream has been disconnected and there's no more bytes in the FIFO. In addition to the normal `File` API, a `uvFifo` has a method called `:isConnecting()`, which returns a boolean indicating the fifo is still connecting, meaning it's possible to verify if the fifo has successfully connected using the boolean expression `not fifo:isConnecting() and not fifo:failed()`.

#### Compressed streams

The `zReader` function will create a read-only File object which decompresses the data from the specified File object. The `file` argument is a File object, and the `size` argument is an optional number that will be used to determine the size of the decompressed data. If not specified, the resulting file won't be seekable, and its `:size()` method won't work, but the file will be readable until `:eof()` returns true. The `raw` argument is an optional string that needs to be equal to `'RAW'`, and will determine whether the data is compressed using the raw deflate format, or the zlib format. Any other string means the zlib format will be used.

## 8.5.4 Iso files

There is some limited API for working with ISO files.

- `PCSX.getCurrentIso()` will return an `Iso` object representing the currently loaded ISO file by the emulator.
- `PCSX.openIso(pathOrFile)` will return an `Iso` object opened from the specified argument, which can either be a filesystem path, or a `File` object.

The following methods are available on the `Iso` object:

```
:failed()       -- Returns true if the Iso file failed in some ways. The Iso object
is defunct if this is true.
:createReader() -- Returns an ISOReader object off the Iso file.
:open(lba[, size[, mode]]) -- Returns a File object off the specified span of
sectors.
:clearPPF()     -- Clears out all of the currently applied patches.
:savePPF()      -- Saves the currently applied patches to a PPF file named after the
ISO file.
```

The `:open` method has some magic built-in. The size argument is optional, and if missing, the code will attempt to guess the size of the underlying file within the Iso. It

will represent the size of the virtual file in bytes. The size guessing mechanism can only work on MODE2 FORM1 or FORM2 sectors, and will result in a failed File object otherwise. The mode argument is optional, and can be one of the following:

- `'GUESS'`: will attempt to guess the mode of the file. This is the default.
- `'RAW'`: the returned File object will read 2352 bytes per sector.
- `'M1'`: the returned File object will read 2048 bytes per sector.
- `'M2_RAW'`: the returned File object will read 2336 bytes per sector. This can't be guessed. This is useful for extracting STR files that require the subheaders to be present.
- `'M2_FORM1'`: the returned File object will read 2048 bytes per sector.
- `'M2_FORM2'`: the returned File object will read 2324 bytes per sector.

The resulting File object will cache a single full sector in memory, meaning that small sequential reads won't read the same sector over and over from the disk.

The resulting File object will be writable, which will temporarily patch the CD-Rom image file in memory. It is possible to flush the patches to a PPF file by calling the `:savePPF()` method of the corresponding Iso object. When writing to one of these files, the filesystem metadata information will not be updated, meaning that the size of the file on the filesystem will not change, despite it being possible to write past the end of it and overflow on the next sectors. Note that while the virtual File object will enlarge to accommodate the writes, it will not be filled with zeroes as with typical filesystem operations, but instead will be filled with the existing data from the iso image.

The ISOReader object has the following methods:

```
1   :open(filename) -- Returns a File object off the specified file within the ISO.
```

This method is basically a helper over the `:open()` method of the Iso object, and will automatically guess the mode and size of the file.

# 8.6 Webserver Lua API

When the webserver is enabled, it will expose the `/api/v1/lua/` prefix, which can be used to execute Lua code on the emulator. When an endpoint with this prefix is called, the Lua table `PCSX.WebServer.Handlers` will be inspected to find a handler for the rest of the path in the endpoint. If a handler is found, it will be called with a request object representing the query, and it has to return a string, which will be sent back to the client as the response. If no handler is found, a 404 error will be returned. If an error occurs while executing the handler, a 500 error will be returned.

The request object has the following fields:

- `form` is a table of the form data in the request. This is only available if the request is a POST request, and the content type is `application/x-www-form-urlencoded`.
- `headers` is a table of the headers in the request.
- `method` is the HTTP method of the request.
- `urlData` is a table with more information about the URL. It has the following string fields:
- `fragment`
- `host`
- `path`
- `port`
- `query`
- `schema`
- `userInfo`

If the returned string starts with the characters "HTTP/", then the web server will consider the response string is a full HTTP response with headers, and will send it as-is to the client. Otherwise, the response string will be sent as the body of a normal 200 response.

# 8.7 Memory and registers

## 8.7.1 FFI access

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.getParPtr()` will return a `cdata[uint8_t*]` representing up to 512kB of the EXP1/Parallel port memory space. This can be 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:
- `PCSX.getReadLUT()` will return a `cdata[uint8_t**]` representing the read LUT for the CPU.
- `PCSX.getWriteLUT()` will return a `cdata[uint8_t**]` representing the write LUT for 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;
```

## 8.7.2 Safer access

The above methods will return direct pointers into the emulated memory, so it's easy to crash the emulator if you're not careful. The `getMemoryAsFile()` method is safer, but will be slower:

- `PCSX.getMemoryAsFile()` will return a `File` object representing the full 4GB of accessible memory. All operations on this file will be translated to the emulated memory space. This is slower than the direct access methods, but safer. Any read or write operation will be clamped to the emulated memory space, and will not crash the emulator.

## 8.7.3 Memory mapping

PCSX-Redux will attempt to forward reads and writes for memory not mapped in the LUTs. This is useful for debugging, but will be slower than the direct access methods.

- `UnknownMemoryRead(address, size)` will be called when a read is attempted to an unmapped memory address. The function should return an 8, 16, or 32-bit value to be returned to the CPU. - `UnknownMemoryWrite(address, size, value)` will be called when a write is attempted to an unmapped memory address. The function should return `true` or `false` indicating whether the write was handled.

# 8.8 Events

The Lua code can listen for events broadcasted from within the emulator. The following function is available to register a callback to be called when certain events happen:

```
PCSX.Events.createEventListener(eventName, callback)
```

**Important**: the return value of this function will be an object that represents the listener itself. If this object gets garbage collected, the corresponding listener will be removed. Thus it is important to store it somewhere that won't get garbage collected right away. The listener object has a `:remove` method to stop the listener before its garbage collection time.

The callback function will be called from an unsecured environment, and it is advised to delegate anything complex or risky enough to `PCSX.nextTick`.

The `eventName` argument is a string that can have the following values:

- `Quitting`: The emulator is about to quit. The callback will be called with no arguments. This is where you'd need to close libuv objects held by Lua through luv in order to allow the emulator to quit gracefully. Otherwise you may soft lock the application where it'll wait for libuv objects to close.
- `IsoMounted`: A new ISO file has been mounted into the virtual CDRom drive. The callback will be called with no arguments.
- `GPU::Vsync`: The emulated GPU has just completed a vertical blanking interval. The callback will be called with no arguments.
- `ExecutionFlow::ShellReached`: The emulation execution has reached the beginning of the BIOS' shell. The callback will be called with no arguments. This is the moment where the kernel is properly set up and ready to execute any arbitrary binary. The emulator may use this event to side load binaries, or signal gdb that the kernel is ready.
- `ExecutionFlow::Run`: The emulator resumed execution. The callback will be called with no arguments. This event will fire when calling `PCSX.resumeEmulator()`, when the user presses Start, or other potential interactions.
- `ExecutionFlow::Pause`: The emulator paused execution. The callback will be called with a table that contains a boolean named `exception`, indicating if the pause is the result of an execution exception within the emulated CPU. This event will fire on breakpoints too, so if breakpoints have Lua callbacks attached on them, they will be executed too.
- `ExecutionFlow::Reset`: The emulator is resetting the emulated machine. The callback will be called with a table that contains a boolean named `hard`, indicating if the reset is a hard reset or a soft reset. This event will fire when calling `PCSX.resetEmulator()`, when the user presses Reset, or other potential interactions.
- `ExecutionFlow::SaveStateLoaded`: The emulator just loaded a savestate. The callback will be called with no arguments. This event will fire when calling `PCSX.loadSaveState()`, when the user loads a savestate, or other potential interactions. This is useful to listen to in case some internal state needs to be reset within the Lua logic.
- `GUI::JumpToPC`: The UI is being asked to move the assembly view cursor to the specified address. The callback will be called with a table that contains a number named `pc`, indicating the address to jump to.

- `GUI::JumpToMemory`: The UI is being asked to move the memory view cursor to the specified address. The callback will be called with a table that contains a number named `address`, indicating the address to jump to, and `size`, indicating the number of bytes to highlight.
- `Keyboard`: The emulator is dispatching keyboard events. The callback will be called with a table containing four numbers: `key`, `scancode`, `action`, and `mods`. They are the same values as the glfw callback set by `glfwSetKeyCallback`.
- `Memory::SetLuts`: The emulator has updated the memory LUTs. The callback will be called with no arguments.

# 8.9 Breakpoints

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

```
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 signature of the invoker callback is:

```
function(address, width, cause)
    -- body
end
```

The `address` parameter will contain the address that triggered the breakpoint. For `'Exec'` breakpoints, this is going to be the same as the current `pc`, but for `'Read'` and `'Write'`, it's going to be the actual accessed address. The `width` parameter will contain the width of the access that triggered the breakpoint, which can be different from what the breakpoint is monitoring. And the `cause` parameter will contain a string describing the reason for the breakpoint; the latter may or may not be the same as what was passed to the `addBreakpoint` function. Note that you don't need to strictly

adhere to the signature, and have zero, one, two, or three arguments for your invoker callback. The return value of the invoker callback is also optional.

For example, these two examples are well formed and perfectly valid:

```
bp1 = PCSX.addBreakpoint(0x80000000, 'Write', 0x80000, 'Write tracing',
function(address, width, cause)
    local regs = PCSX.getRegisters()
    local pc = regs.pc
    print('Writing at ' .. address .. ' from ' .. pc .. ' with width ' .. width ..
' and cause ' .. cause)
end)

bp2 = PCSX.addBreakpoint(0x80030000, 'Exec', 4, 'Shell reached - pausing',
function()
    PCSX.pauseEmulator()
    return false
end)
```

The returned breakpoint 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.

Note that the breakpoint will run outside of any safe Lua environment, so it's possible to crash the emulator by doing something wrong which would normally be caught by the safe environment of the main thread. This is to ensure that the breakpoint can run as

fast as possible. In order to avoid this, it's possible to wrap the invoker callback in a `pcall` call, which will catch any error and display it in the logs. For example:

```
local someActualFunction = function(address, width, cause)
    -- body
end
bp = PCSX.addBreakpoint(0x80030000, 'Write', 4, 'Shell write tracing',
function(address, width, cause)
    local success, msg = pcall(function()
        someActualFunction(address, width, cause)
    end)
    if not success then
        print('Error while running Lua breakpoint callback: ' .. msg)
    end
end)
```

This will ensure that the breakpoint will never crash the emulator, and will instead display the error in the logs, but it will also slow down the execution of the breakpoint. It's up to the user to decide whether or not this is acceptable.

# 8.10 Inline assembler

There is a Lua API for an inline MIPS assembler.

One can instantiate an assembler with `PCSX.Assembler.New()`, which will keep all the state of the assembler. The assembler can be used to assemble a string of MIPS code, and then compile it to memory or a file.

The object has the following methods:

- `:parse(code)` will parse the string `code` and assemble it. It will return the assembler object itself, so it can be chained with the compile methods. The parser is fairly simple, but it should be enough for most cases. The parser should handle all of the basic MIPS instructions, all of the PS1's GTE opcodes, and many pseudo-instructions. It will also handle labels. The parser is more lenient than normal MIPS assemblers, and will accept some invalid syntax, but it will throw an error if it can't parse the code.
- `:compileToUint32Table(baseAddress)` will compile the assembled code to a table of `uint32_t` values. This is useful for debugging, but not very useful for actually running the code. The `baseAddress` is the address that the code will be loaded at, in order to handle relative jumps.
- `:compileToMemory(memory, baseAddress, memoryStartAddress)` will compile the assembled code to an indexable memory object, such as an ffi array. The memory object must be at least as large as the assembled code. The memory object will be modified in-place. The `baseAddress` is the address that the code will be loaded at, in order to handle relative jumps. The `memoryStartAddress` is the address that the memory object starts at.
- `:compileToFile(file, baseAddress, fileStartAddress)` will compile the assembled code to a file object. The file object must be at least as large as the assembled code. The file object will be modified in-place. The `baseAddress` is the address that the code will be loaded at, in order to handle relative jumps. The `fileStartAddress` is an optional argument which defaults to 0, and is the address that the file object starts at. Using a 0-based file address is relevant when using with the `PCSX.getMemoryAsFile()` function, or when using a `Support.mem4g()` File object.

# 8.11 Handling of PSX binaries

There is some support for handling PSX binaries in the Lua API. The `PCSX.Binary` module has the following functions:

- `PCSX.Binary.load(input, output)`: loads an input `File` object into an output `File` object. The input file must be a valid PSX binary, which can be in the formats CPE, PS-EXE, PSF, or ELF, and the output file must be at least 4GB large, which means it's really only suitable with the `mem4g` object, or the object returned by `getMemoryAsFile()`. The output file will be modified in-place. The output file will be loaded at the address specified in the binary header. If successful, the function will return an info structure with the following optional fields:
- `pc`: the entry point of the binary
- `gp`: the global pointer of the binary
- `sp`: the stack pointer of the binary
- `region`: the region of the binary, which can be one of the following:
- `'NTSC'`: NTSC region
- `'PAL'`: PAL region
- `PCSX.Binary.pack(src, dest, addr, pc, gp, sp, options)`: compresses the input binary stream into a self-decompressing stream. The input must be a `File` object, and the output must be a `File` object. The `addr` is the address that the binary will be loaded at. The `pc`, `gp`, and `sp` are the entry point, global pointer, and stack pointer of the binary. The `options` is an optional table with the following optional fields:
- `tload`: the address that the compressed binary will be loaded at. If not specified, it will be set to a suitable address. Not specifying this will generate an in-place decompression binary, which doesn't require much extra memory. When specifying this, the whole output stream needs to be loaded at this specific address, and the decompression code will be located at its beginning, meaning both the entry point and the loading addresses will be the same.
- `booty`: a boolean specifying that the output stream will be suitable to boot as a PIO bytestream. Incompatible with `tload` or `raw`.
- `shell`: a boolean specifying that the output stream will attempt to reboot the machine and load the binary, which can be useful when resetting the kernel.
- `raw`: a boolean specifying that the output stream will be a raw binary, without a PS-EXE header. It doesn't make sense to use this without `tload`.

- `PCSX.Binary.createExe(src, dest, addr, pc, gp, sp)`: creates a PS-EXE binary from the input binary stream. The input must be a `File` object, and the output must be a `File` object. The `addr` is the address that the binary will be loaded at. The `pc`, `gp`, and `sp` are the entry point, global pointer, and stack pointer of the binary.

The above methods can be used for example the following way:

```
local src = PCSX.getCurrentIso():createReader():open('SLUS_012.34;1')

local m4g = Support.File.mem4g()
local info = PCSX.Binary.load(src, m4g)
local asm = PCSX.Assembler.New()
asm:parse [[
    lui   $a0, 0x8001
    addiu $a0, 0x1234
]]:compileToFile(m4g, 0x80045678)
local bytes = m4g:subFile(m4g:lowestAddress(), m4g:actualSize())

local dst = Support.File.open('compressed-from-lua.ps-exe', 'TRUNCATE')

PCSX.Binary.pack(bytes, dst, m4g:lowestAddress(), info.pc, info.gp, info.sp)
```

Additionally, the `PCSX.Misc` module has the following functions:

- `PCSX.Misc.uclPack(src, dest)`: compresses the input binary stream into a ucl-compressed stream. Both the input and output arguments must be `File` objects. The output stream will be written at its current write pointer, and will be compressed using the UCL-NRV2E compression algorithm, which is a variant of the UCL compression algorithm. The output stream can be decompressed in-place with very little memory overhead. Simply place the compressed data at the end of the decompression buffer + 16 bytes. The stream doesn't require to be aligned in any particular way.
- `PCSX.Misc.writeUclDecomp(dest)`: writes a MIPS UCL-NRV2E decompression routine to the output `File` object, at its current write pointer. The function returns the number of bytes written, which at the moment is 340 bytes. The code is position independent, and has the following function signature:
- `void decompress(const uint8_t* src, uint8_t* dest);`

# 8.12 Case studies

## 8.12.1 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.

## 8.12.2 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
```

## 8.12.3 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.

# 9. Openbios

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

## 9.1 Purposes of Openbios

- Educational
- Ease of distribution
- Automated testing

See this page for more details.

## 9.2 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.

## 9.3 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.

## 9.4 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.

## 9.5 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.

## 9.6 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.

## 9.7 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.