

FP-GAme

Developer's Guide

This guide contains learning resources and setup needed to develop retro games for the FP-GAme platform. We will cover FP-GAme hardware capabilities, software development guidelines and setup, art/asset pipelines, and more. A table of contents is included below for convenience. This guide is written in such a way that you can jump straight to the section that most interests you. If prerequisite knowledge from another section is required, we will let you know.

Table of Contents

[FP-GAme Hardware Capabilities](#)

[FP-GAme Technical Overview](#)

[FP-GAme User Library Overview](#)

[FP-GAme Programming Best Practices](#)

[Supported Art Pipeline](#)

[Supported Audio Pipeline](#)

[Your Very First Game Project Setup](#)

[Debugging Over Minicom](#)

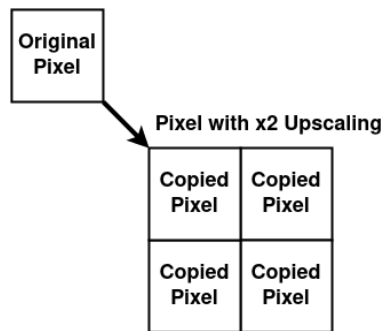
[Modifying FP-GAme](#)

FP-Game Hardware Capabilities

Video and Screen Size

FP-Game displays a 320x240 pixel screen over HDMI by upscaling the original image to 640x480. The video itself runs at 60FPS, but can drop lower depending on the speed of your game's code.

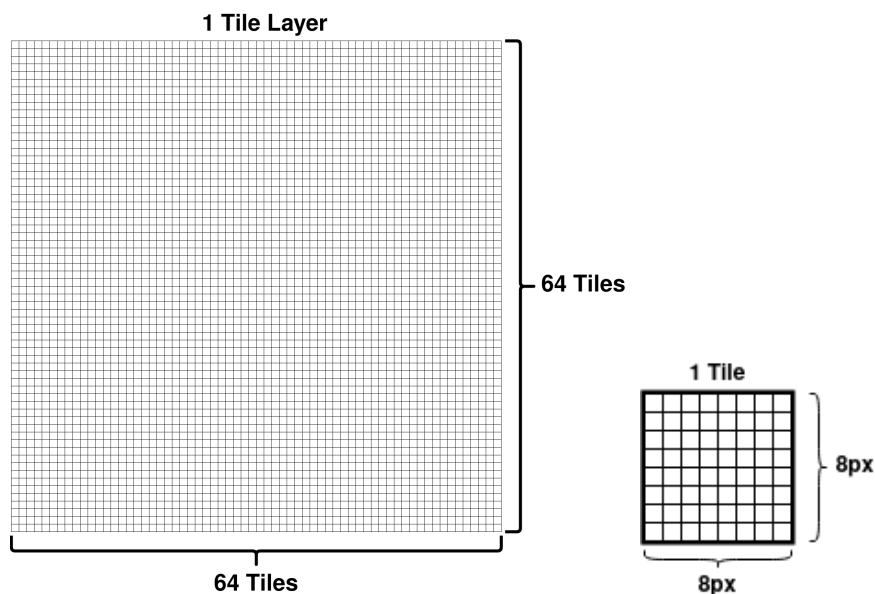
Upscaling just means that each pixel of the 320x240 image takes up more actual pixels of your screen. This upscaling is done to maintain compatibility with most modern monitors, which can only display 640x480 or higher resolution. In our case, our hardware automatically performs x2 upscaling on each pixel.



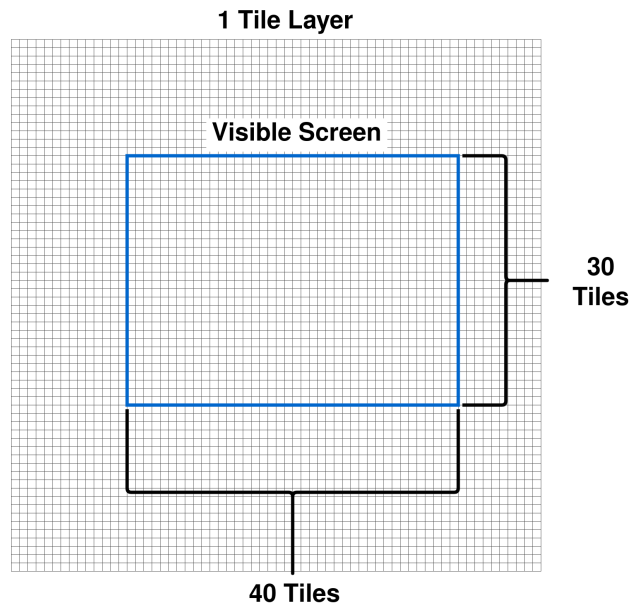
Tile Layers

FP-Game hardware features two separate tile layers to display tile graphics. These are the foreground and background tile layers, respectively. Each tile layer consists of 64x64 tiles each, with each tile made up of an 8x8 pixel image.

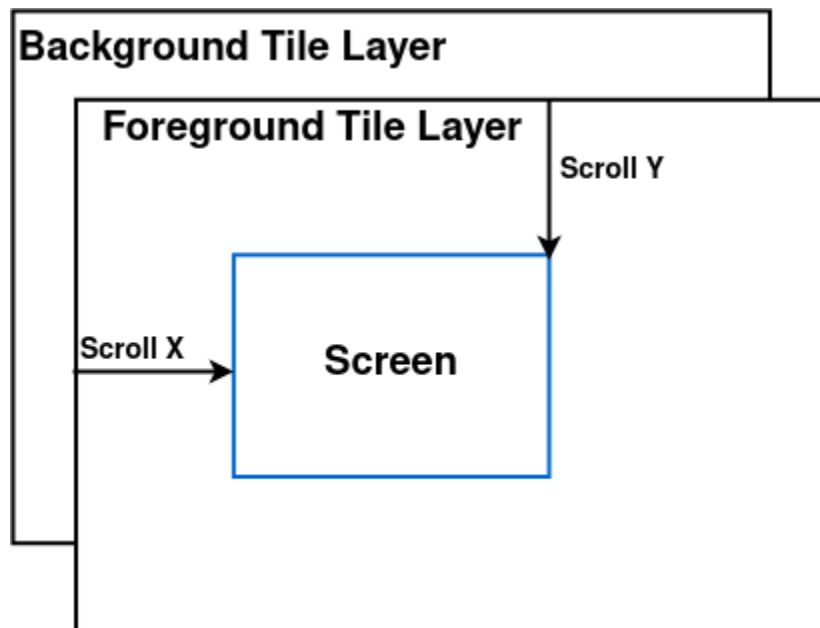
The foreground tile layer renders on top of the background tile layer. This means that the background tile layer will show on screen underneath any transparent tiles in the foreground layer, or will always show if the foreground layer is disabled through software.



Our screen size is 320x240 pixels, or 40x30 tiles worth of screen. However, this doesn't always mean that 40x30 tiles are shown on-screen at once. The exception is this: If you scroll a tile layer, a portion of an extra tile will be visible in the direction you scrolled until you have scrolled a full tile (8-pixels in said direction). So in reality, we show a maximum of 41x31 tiles and a minimum of 40x30 tiles on-screen at once.



Speaking of scrolling: Both the foreground and background tile layers can be scrolled independently. Scroll values are based on the origin being the top-left of the tile layer. You can think of scrolling as determining where your "camera" or screen view is within relation to the foreground and background layers.



Each tile layer can be enabled or disabled through software. When disabled, they act as if they are transparent.

Each tile within a tile layer has three properties:

- The 8x8 pixel pattern to display (position in Pattern-RAM).
- The color palette to use for displaying that 8x8 pixel pattern.
- The mirroring settings for the tile (Horizontal mirror, Vertical mirror, both, or none).

The tile layers themselves have the following properties:

- (x, y) scroll.
- Enable/Disable layer.

Sprite Layer

FP-Game hardware supports up to 64 sprites, with up to 16 sprites per scanline.

If more than 16 sprites would be visible on a particular scanline, only the 16 sprites with the highest render priority will be displayed (leaving the remaining sprites invisible). The render priority between sprites is determined by two factors:

1. Layer Render Priority. This determines whether the sprite:
 - a. Is rendered behind all layers.
 - b. Is rendered in the middle of background and foreground tile layers.
 - c. Is rendered on top of all layers.
2. Sprite-ID. This is the number, [0, 63] inclusive, identifying the sprite. This is also its position in Sprite-RAM (but more on that in [FP-Game Technical Overview](#) if you are interested).

Sprites that render on top of all layers are prioritized over sprites that do not. Sprites with a lower Sprite-ID are prioritized over sprites with higher Sprite-IDs.

Sprites can have variable widths and heights. FP-Game supports lengths of 8px, 16px, 24px, or 32px in both directions. For instance, a sprite can be 8x8 px, or 8x16 px, or 32x16 px, ... etc.

Sprites can be positioned relative to the screen using (0,0), the origin, at the top-left corner. A sprite's position controls the location of the top-left pixel of its image.

Each sprite within the sprite layer has the following properties:

- Sprite-ID (0 through 63).
- Rendering order (behind all, middle, in front of all).
- Position (x, y) of the top-left of the sprite.
- The pattern to display (initial pattern's position in Pattern-RAM, width, and height).
- Color palette to use.
- Mirroring settings (Horizontal mirror, vertical mirror, both, or none).

Palettes

FP-Game uses 15-color palettes (16, but with the first color marking transparency). Each color is in 24-bit RRGGBB format.

Every layer has its own independent set of color palettes:

- Background Tile Layer: 16 Palettes
- Foreground Tile Layer: 16 Palettes
- Sprite Layer: 32 Palettes

Universal Background Color

If it turns out that all layers are transparent in a given area (or are disabled), then that area will be rendered with the universal background color. This universal background color can be programmed using the PPU User Library.

Audio Player

FP-Game supports playback of 32KHz 8-bit PCM audio. This is much lower quality than modern consoles, and is meant to emulate that “Retro” sound. A bit of extra work must be done to your original music/sound files to get them to work with FP-Game. See the [Supported Audio Pipeline](#) section for more information.

GPIO-Based SNES Controller Port

FP-Game uses the GPIO of the DE10-Nano board to host a SNES Controller Port. This means that you can play games using an SNES controller. This input is polled at 60Hz to enable maximum responsiveness with 60FPS video.

For more information on setup, see the fpgame_getting_started.pdf document in the docs/ subfolder of the FP-Game User Repository.

For more information on how to create the SNES controller port, see snes_controller_mod_instructions.pdf also included in this repository.

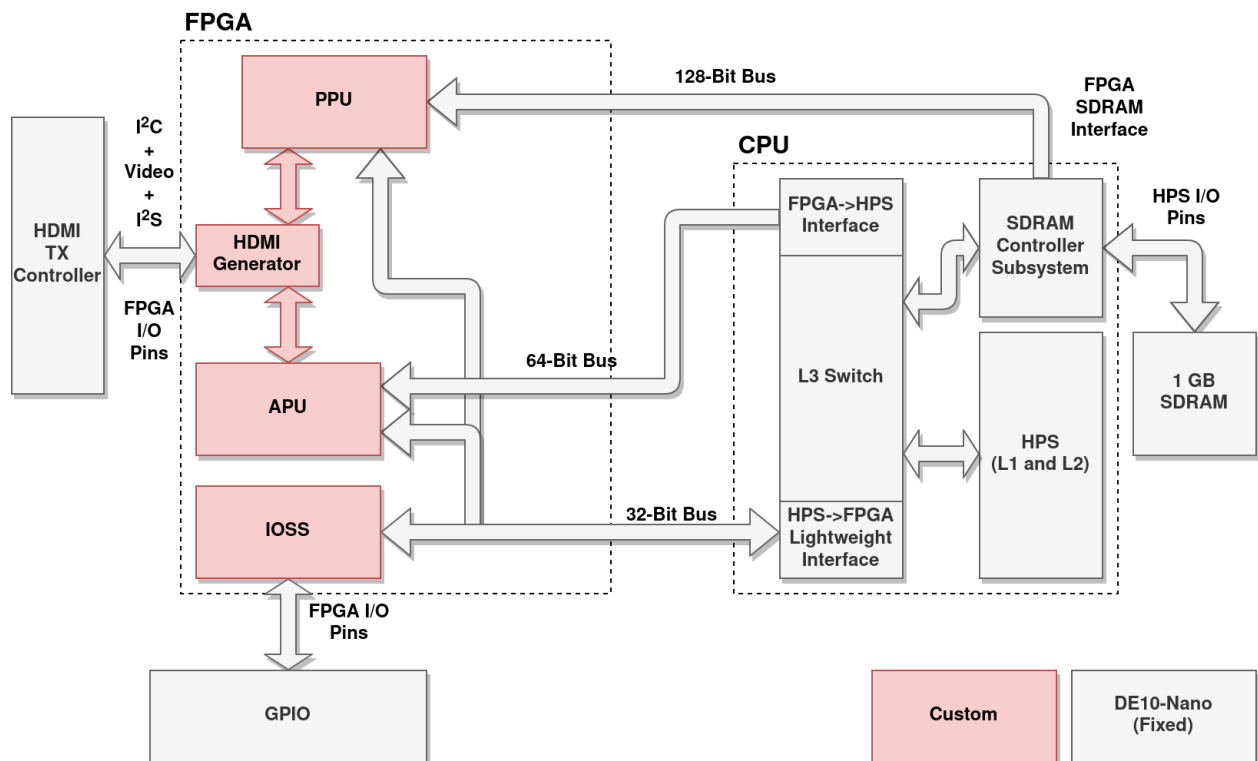
FP-Game Technical Overview

This section is not mandatory, but offers a deeper look into the FP-Game system, including hardware, kernel module drivers, user library, and the relationships between them.

It is recommended that you at least read the section on VRAM Layout, as it is helpful in understanding how tiles and sprites determine which pixel/color pattern they display.

System Overview

In this section, we will cover the crucial components of the FP-Game system by discussing our high-level system diagram.



Our system diagram is a bit busy, I will admit, but below I've listed the important points:

- The CPU hosts Linux, which your game code runs on. The CPU can talk to the Pixel Processing Unit (PPU), Audio Processing Unit (APU), and I/O Subsystem (IOSS) via various system busses.
- The PPU renders the current frame based on the latest fully-transferred copy of Video RAM (VRAM) and sends the output to HDMI.
- The APU reads in 8-bit PCM Sample Buffers from the CPU. These get sent over I2S to our HDMI Generator, which adds them to the HDMI signal going to your monitor.
- The IOSS implements an SNES controller port. It polls the SNES controller for a list of the currently pressed buttons. These are later read by the CPU via a status register accessible over MMIO.

VRAM Layout

The PPU reads from VRAM to determine what to render to the screen. There are 4 basic data types, each corresponding to one of the 4 segments of VRAM.

Firstly, tile data entries are 2B each and hold three attributes, representing a tile on screen:

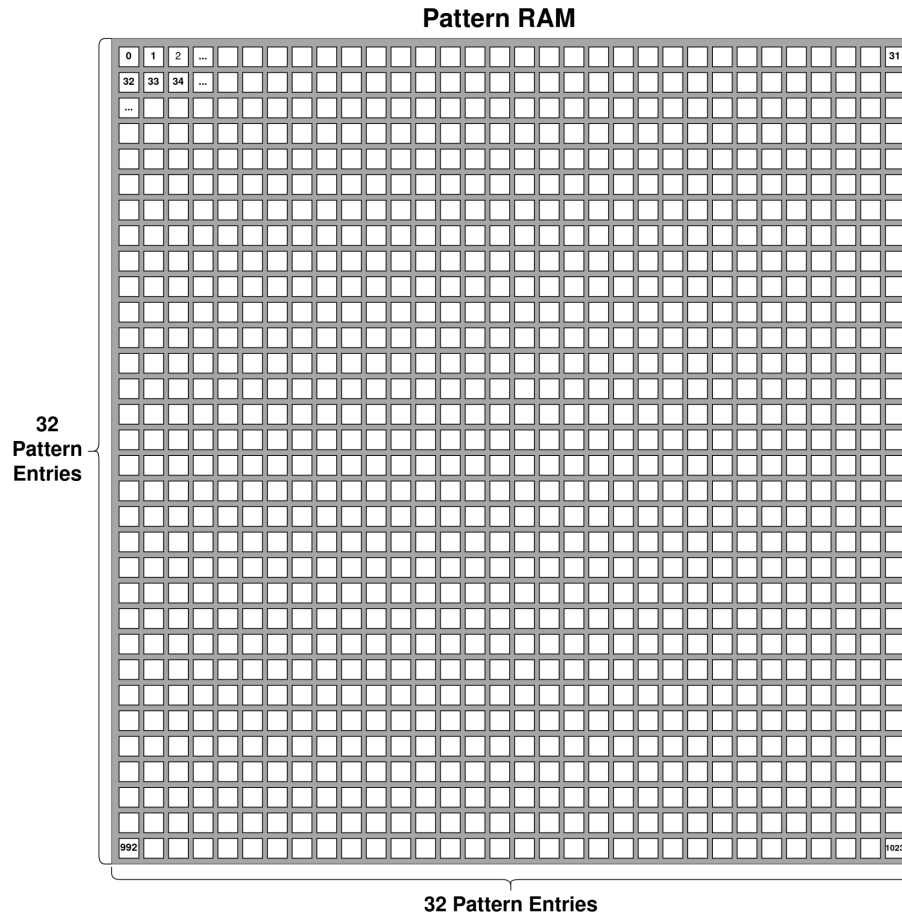
1. Pointer to an 8x8 pattern in Pattern-RAM (sometimes referred to as Pattern-ID).
2. Pointer to the color palette for this tile in Palette-RAM (sometimes referred to as Palette-ID).
3. Mirror state. This controls whether the graphics for this tile are:
 - a. Not changed
 - b. Flipped horizontally
 - c. Flipped vertically
 - d. Flipped both horizontally and vertically

Tile data entries are contained in Tile-RAM in row-major order (from top-left of the screen to bottom right). In memory, 64 tile-data entries representing the first row of tiles occur first, then the 64 tile-entries for the 2nd row occur, and so on until the 64th row of tile data entries.

There are two segments to Tile-RAM. The first segment contains 64x64 tile data entries for the background tile layer, the second segment contains another 64x64 tile data entries for the foreground tile layer.

Next, pattern data entries are 8x8 pixel tiles, with each pixel represented as a 4-bit value (corresponding to one of 16 colors in a palette) for a total of 32B per pattern entry.

Pattern-RAM is organized as a 32x32 grid of pattern data entries. This, like the Tile-RAM, follows row-major order: The first row of 32 8x8 patterns occur first, then the second row of 32, and so on, until the 32nd row of 32. More visual representations can be found in the PPU User Library documentation (ppu.h) in examples/techdemo/usr/docs/html/index.html. Another is provided here for reference.



Each square is a pattern-data entry in the above diagram.

- A pattern address of 0 indicates the top left tile.
- A pattern address of 1 indicates the pattern just to the right of that.
- A pattern address of 31 indicates the last tile in the first row.
- A pattern address of 1023 indicates the last tile in the last row.
- We can store a total of 1024 pattern entries (8x8 pixel images).

The next data type to discuss are palette entries. Palette entries contain 16 RRGGBB format colors (and take up $16 \times 4B = 64B$ each). The first color in every palette is considered transparent. It does not matter what this color is to the PPU. Note, however, it is important to define this color to be the color on our sprite/tile image that we want transparent in our export scripts (see [Supported Art Pipeline](#)).

There are 3 sections of Palette-RAM, each dedicated to one of the three layers (Background Tile, Foreground Tile, and Sprite).

- 16 Palettes for Background Tiles
- 16 Palettes for Foreground Tiles
- 32 Palettes for Sprites

Tile data will automatically reference the background palettes section or foreground palettes section depending on their location in Tile-RAM.

The last segment of VRAM is Sprite-RAM. This segment is actually split into 2 sections:

1. 64 Sprite Data entries
2. 64 Extra Data entries

Sprite data entries contain the following features:

- Pointer to an 8x8 pattern in Pattern RAM. This defines where the start of the sprite graphics occur.
- Pointer to a palette in Palette RAM. This is what the sprite uses for its color palette.
- Y position of the top-left corner of the sprite on-screen.
- X position of the top-left corner of the sprite on-screen.

Each sprite data entry corresponds to another “extra data” entry. These contain additional information for that sprite, including:

- Mirror state (Same as for tile-data, but these apply to the whole sprite, even if the sprite is bigger than 8x8).
- Height. This value is in terms of 8x8 patterns. A sprite can be 1, 2, 3, or 4 patterns tall.
- Width. This value is in terms of 8x8 patterns. A sprite can be 1, 2, 3, or 4 patterns wide.
- Priority. This value determines whether the sprite displays behind both tile layers, in the middle of both tile layers, or above both tile layers.

An important concept in sprites is how patterns are addressed in Pattern RAM. The sprite uses patterns from Pattern-RAM by the following method:

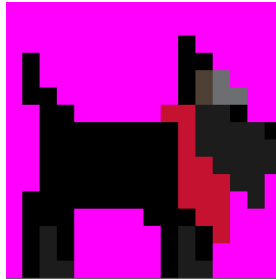
1. Define a rectangular region in Pattern-RAM with the top-left of the rectangle starting at the pattern pointed to by the pattern-data pointer in the sprite-data entry.
2. The width and height of this rectangle are determined by the extra-data for this sprite.
3. The pattern-data enclosed by this rectangle is what the sprite displays.

An example is included here using our tech demo sprite in a smaller (16x16) version of Pattern RAM (for readability).

Pattern RAM

[illegible]

In the above example, we should define our sprite with pattern address 33, width = 2, and height = 2. If we do this, our final sprite will look like this on-screen:



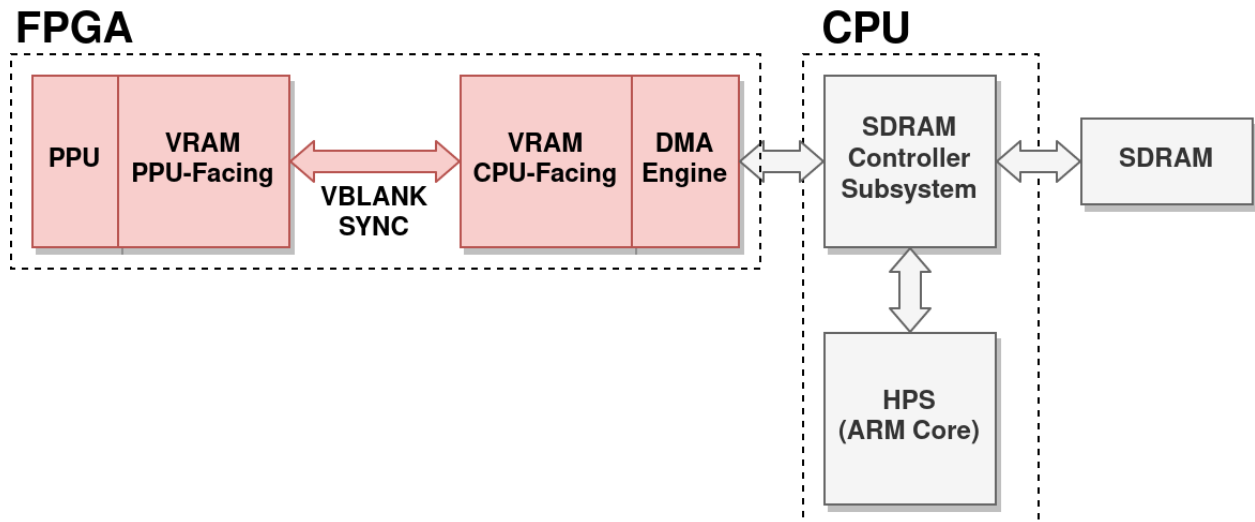
To summarize, VRAM has four segments. These are repeated below with included byte-addresses:

- **0x0000: Tile RAM**
 - Contains tile data for use with the two tile layers.
 - Background Tile Layer starts at 0x0000 and ends at 0x1FFF
 - Foreground Tile Layer starts at 0x2000 and ends at 0x3FFF
- **0x4000: Pattern RAM**
 - Contains a 32x32 grid of pattern data
- **0xC000: Palette RAM**
 - Contains a total of 64 palette data entries. 16 for each tile layer (32 total) and 32 for the sprite layer.
- **0xD000: Sprite RAM**
 - Contains 64 sprite-data entries.
 - Contains a segment with extra data for each sprite starting from 0xD100.

PPU-CPU Communication

High-level understanding of the PPU and CPU communication is important for programming performant games on FP-Game. In this section, we provide a high-level overview of how the PPU receives data from the CPU.

The PPU contains two internal copies of Video RAM (referred to as a Double-Buffered VRAM).



When rendering a frame to send out to HDMI, the PPU reads the “PPU-Facing VRAM” directly. The CPU-Facing VRAM contains either:

- The most recent completed transfer of video data from the CPU.
- Or, a transfer-in-progress. This happens when the DMA-Engine is busy reading from the CPU’s updated VRAM copy. This VRAM copy is the same one your code writes to through the User Library.

During the downtime between rendering frames (known as VBLANK), the CPU-Facing and PPU-Facing VRAMs will synchronize, updating the PPU-Facing VRAM with the latest video data from the CPU.

If, however, VBLANK occurs and the CPU-Facing VRAM contains a transfer in progress (DMA-Engine is busy), no synchronization will occur, and the PPU will reuse the old copy of VRAM for rendering the next frame. This is called “Dropping Frames”, and is similar to what modern video games will do when frame-rate drops due to heavy CPU/GPU load. The result will be that the image stays on screen for longer than normal (usually an extra frame), temporarily reducing your effective frame-rate to < 60FPS.

How can you prevent “Dropping Frames”? We have written our user library and kernel modules so that synchronizing your game logic and rendering to the PPU’s VBLANK timing is easy to do. See the section about Game Loops and Rendering in [FP-Game Programming Best Practices](#), for more information on this.

So far we have covered how the PPU does its internal synchronization and rendering, but how does new video data get from the CPU to the CPU-Facing VRAM? When the CPU is ready to transfer its virtual VRAM copy (located in SDRAM), the CPU writes the physical address of this VRAM copy to a PPU status register. The DMA-Engine will immediately start copying the entire VRAM copy from SDRAM into the CPU-Facing VRAM. Most importantly, the CPU is not allowed to write new changes to its VRAM copy in SDRAM while the data transfer is taking place. This is to prevent corruption of the data being transferred. This is also why many of the PPU User Library `ppu_write_x` functions can fail: If the DMA-Engine is currently transferring the CPU's copy of VRAM, you are not allowed to write to the CPU's copy of VRAM.

To summarize and put things into perspective, below is a step by step process of what a typical game will do to render a frame, starting from the software abstraction, and finishing with the hardware:

1. The game software will read inputs from the controller and update game logic. This will result in changes in the player's character, enemy AI, the background or foreground graphics, etc.
2. The game software will call user library functions `ppu_write_x` (tiles, sprites, etc.) to write changes to the CPU's copy of VRAM.
3. When finished making changes to the CPU's copy of VRAM, the game software will call `ppu_update` to send the physical address of the CPU's VRAM to the PPU, and start the DMA-Engine.
4. The DMA-Engine transfers all of the CPU's VRAM copy to the CPU-Facing VRAM.
5. VBLANK occurs at some point after. The PPU-Facing VRAM is updated with the CPU-Facing VRAM, containing the new changes.
6. The PPU uses the PPU-Facing VRAM to render the frame and send it out over HDMI.

Linux and Kernel Module Drivers

Your game code runs on Linux and interacts with the FP-GAME hardware via a user library. Behind the scenes, this user library writes to three devices in `/dev/`. Writes to these devices are handled by our Linux kernel modules, which do the communication with the FP-GAME hardware on your behalf.

FP-Game User Library Overview

The FP-Game User Library is pre-built and included alongside the tech demo. The library folder can be found at `examples/techdemo/usr` in this user repository. As you will see in [Your Very First Game Project Setup](#), this folder should be copied into your project folder in order for you to use the prebuilt library.

In your code, you should include 3 header files:

- `#include <fp-game/apu.h>`
 - Include this for audio playback functionality.
- `#include <fp-game/ppu.h>`
 - Include this for graphics functionality.
- `#include <fp-game/con.h>`
 - Include this for reading the SNES controller state.

Also included with this library (in the `usr` folder) is a `html` folder full of Doxygen-generated documentation. To access this documentation, open the `index.html` file contained within.

FP-Game Programming Best Practices

Included here are best practices for programming your retro game. The User Libraries do their best to abstract away some of the minute hardware details. Occasionally, knowledge of the underlying system will assist you in programming more performant games.

Specifying Unused Sprites

You should specify any sprites not being used in OAM as off-screen or else they will affect rendering!

- You can do this by setting $y = 240$ for all unused sprites.
- Recall only 16 sprites are supported per scan-line. The default Sprite-RAM values are zeroed out, so if pattern 0 happens to be a non-transparent pattern, then all unused sprites will have pattern 0 and be visible at (0,0).
- You can also get around this by making pattern 0 transparent (according to palette 0).

Game Loop and Rendering

FP-Game's PPU and CPU communicate video data via a large DMA transfer. This transfer is started whenever `ppu_update` is called and returns 0. If `ppu_update` instead returns 1, then you know there is a DMA transfer in progress. What does this mean for a game programmer?

- In order to maximize alignment with the PPU's internal timing (meaning that your game logic will cause less frame-skips if it takes too long), use a game loop.

What is a game loop? It is a structure or organization for your code:

- Read input
- Do your game logic for this frame:
 - Move the player 1 step based on the controller input.
 - Update the state or positions of the AI characters.
 - Is the player currently touching a powerup?
 - ... etc.
- Commit all changes to memory
- Update screen

Most importantly, we recommend organizing your code by putting "Commit all changes to memory" and "Update screen" last, in that order.

In terms of real FP-Game user library functions, this means calling the "`ppu_write_x`" functions before calling "`ppu_update`". The "`ppu_write_x`" functions return 1 if the PPU is currently busy and return 0 if the PPU is free. Since the PPU may be busy when you are trying to call "`ppu_write_x`" to commit your changes, you will need to poll this function until the PPU is no longer busy and returns 0, meaning your changes have successfully been committed and will be displayed on the next `ppu_update` call.

To summarize, poll the `ppu_write_x` functions when you are finished with your game logic. This ensures you are not blocking logic during the time when the PPU is busy.

For an example of this game loop in practice, see the included “techdemo” example under `examples/techdemo/main.c`.

Supported Art Pipeline

FP-Game’s PPU User Library use the following text-based formats:

- `.pattern`
 - This contains several rows of pixel values separated by newlines.
 - Each row contains a number of hex characters (0-F), indicating which out of the 16 colors (with 0 as transparency) are to be shown for pixels in that row.
- `.tilemap`
 - This contains several rows of (PPP, C, M) entries, where
 - PPP indicates three hex character (0-F each), which determine the pattern address to use for the given tile.
 - C indicates the color palette ID to use for the given tile.
 - M indicates the mirror setting
 - 0 for no mirror
 - 1 for horizontal mirror
 - 2 for vertical mirror
 - 3 for both
- `.palette`
 - This contains 16 rows of RRGGBB format colors. Each row is a color in the color palette and the first row is the color which will be “transparent”.

These file formats can be loaded in with the PPU User Library functions, and then written to their respective section in VRAM for usage in your game.

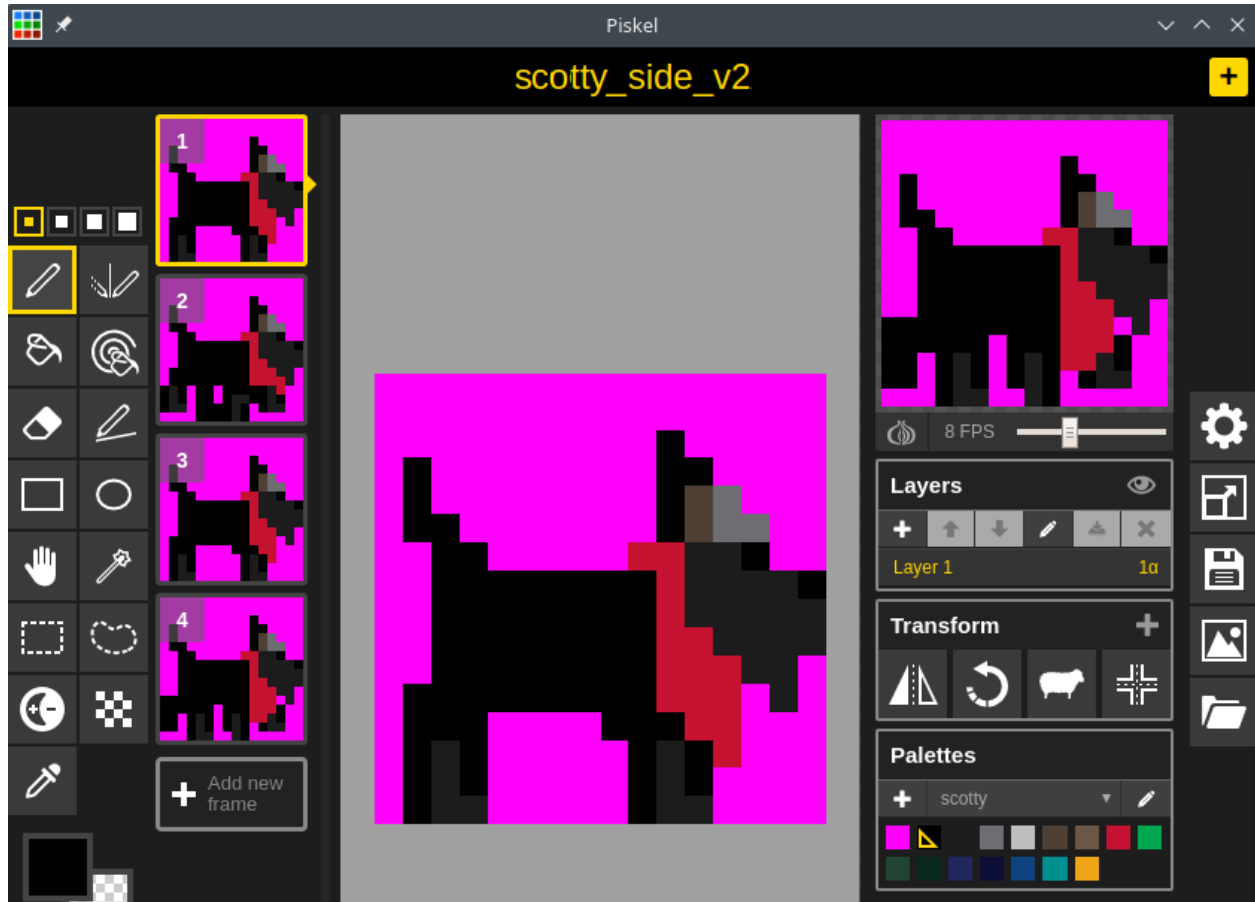
Pattern and Palette Export

The techdemo `.pattern` and `.palette` art assets were created using Piskel:

<https://www.piskelapp.com/>

We decided to implement some scripts to convert the export formats of Piskel to those used by FP-Game’s user libraries. These scripts and a README file about their usage can be found in the user repository under the `user_tools/` subdirectory.

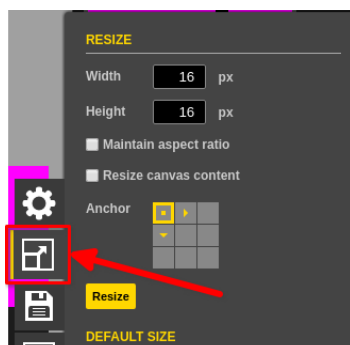
Included in this guide is a quick demonstration of the Piskel export process for usage with our Scotty Dog sprite from the techdemo game.



In the Piskel view above, I have drawn 4 frames of the 16x16 pattern we use with the Scotty Dog sprite in our techdemo game. After exporting from Piskel, our converter scripts will automatically separate out these frames into individual .pattern files. Individual frames must be kept as separate .pattern files due to the way patterns are stored in memory. You can do sprite animation by modifying which pattern a sprite is currently displaying at a given time (much like the idea of a sprite-sheet).

Also notice that our “scotty” palette has 16 colors. The first color of a palette must be the color to use as “transparent”. In our case, this is the pink color surrounding the Scotty Dog.

To set up this view at 16x16 pixels, you need to click on the icon shown in the image below.



This icon opens a menu which allows you to resize the drawing view.

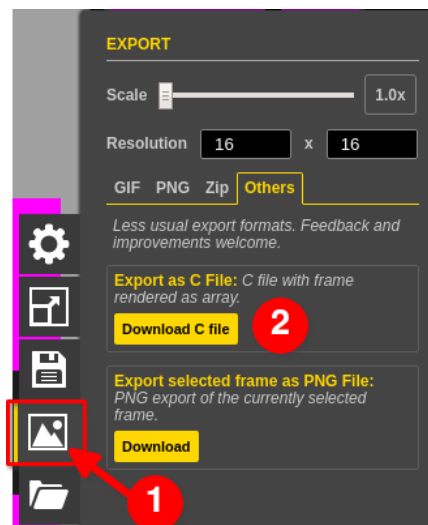
FP-Game supports 8x8 patterns for tiles and any combination of the following widths and heights for patterns used for sprites:

Width: 8, 16, 24, 32

Height: 8, 16, 24, 32

For example, you can do 8x16, 24x8, 32x32, ...etc. sprites.

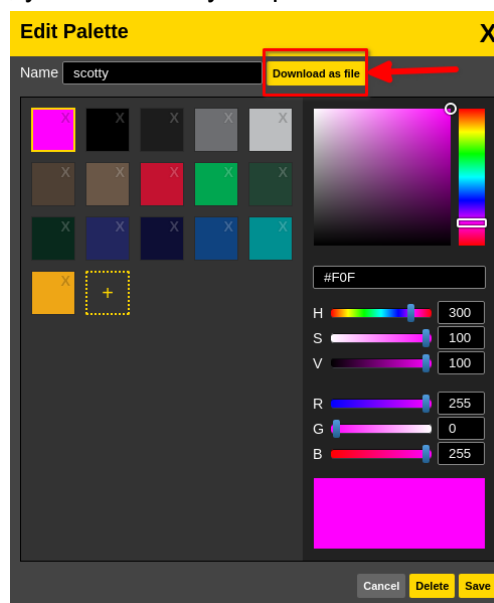
Finally, to export your completed pattern from Piskel, you must click on the photo frame icon as shown in the image below (1).



The menu as shown above will open. From here, “Export as C File” (2). Our `c_to_pattern.py` script converts these files into `.pattern` format for use with FP-Game.

Another step that must be taken to use your pixel art within FP-Game is to export the color palette you used to create that pixel art.

To access the export window, you must edit your palette:



“Download as file” will allow you to save the palette as `.gpl`. In order to be compatible with our `gpl_to_palette.py` converter script, you must ensure that you have exactly 16 colors in your palette. It is okay if some colors in the palette are unused by your sprite; It is not okay to have colors in the aforementioned `.c` export missing from your palette, however.

With your .gpl palette and .c graphics files exported, it is time to use the converter scripts.

First, run:

```
python gpl_to_palette.py <Path to .gpl File> <Path to store .palette>
```

This will copy your .gpl file and convert it to a .palette file, writing the result to <Path to store .palette>.palette. You should avoid adding an extension to that variable, as it is appended for you.

With your .palette file generated, it is time to convert your .c to one or more .pattern files.

Run the following command:

```
python c_to_pattern.py <.palette file> <.c file> <Path to store .pattern(s)>
```

Replacing:

- <.palette file> with the path to your recently exported .palette file
- <.c file> with the recently exported .c file from Piskel
- <Path to store .pattern(s)> with the path and filename (without file extension) you want to store the converted output to. If multiple frames are exported in the .c file from Piskel, the output will be split into several files using your filename and appending "-1", "-2", ... etc.

Congratulations, you now have a .pattern and a .palette file you can use with the user library functions ppu_load_pattern.

Tilemap Export

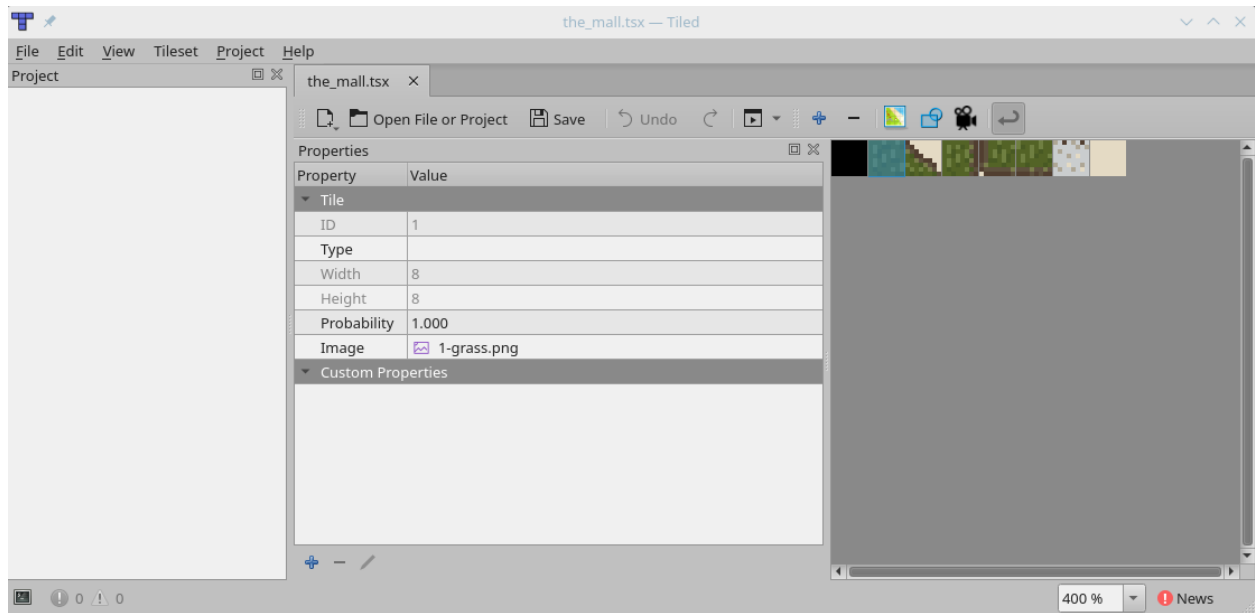
The .tilemap file used for the techdemo game included in this repository was generated using a free tool called Tiled:

<https://www.mapeditor.org/>

Tiled is great for planning your tile layouts. It isn't mandatory for your games, but it greatly reduces your workload in games that utilize room-based worlds where you load full rooms of tiles all at once.

To use Tiled, you must first export any tile images you want to .png format from Piskel. You do not need to worry about palettes for Tiled.

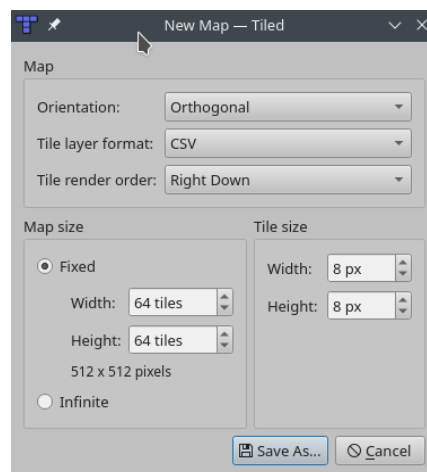
When opening Tiled, create a new Tileset file. This will open a window with the following appearance:



You can drag your .png file into the right section and they will automatically be added to the Tileset file. NOTE: Please do this in the order you wish to have the tile patterns organized in Pattern RAM. This is crucial, because the .csv file we export later will tell the PPU user library to link the tiles to the “ID” of the 8x8 tile in Pattern RAM.

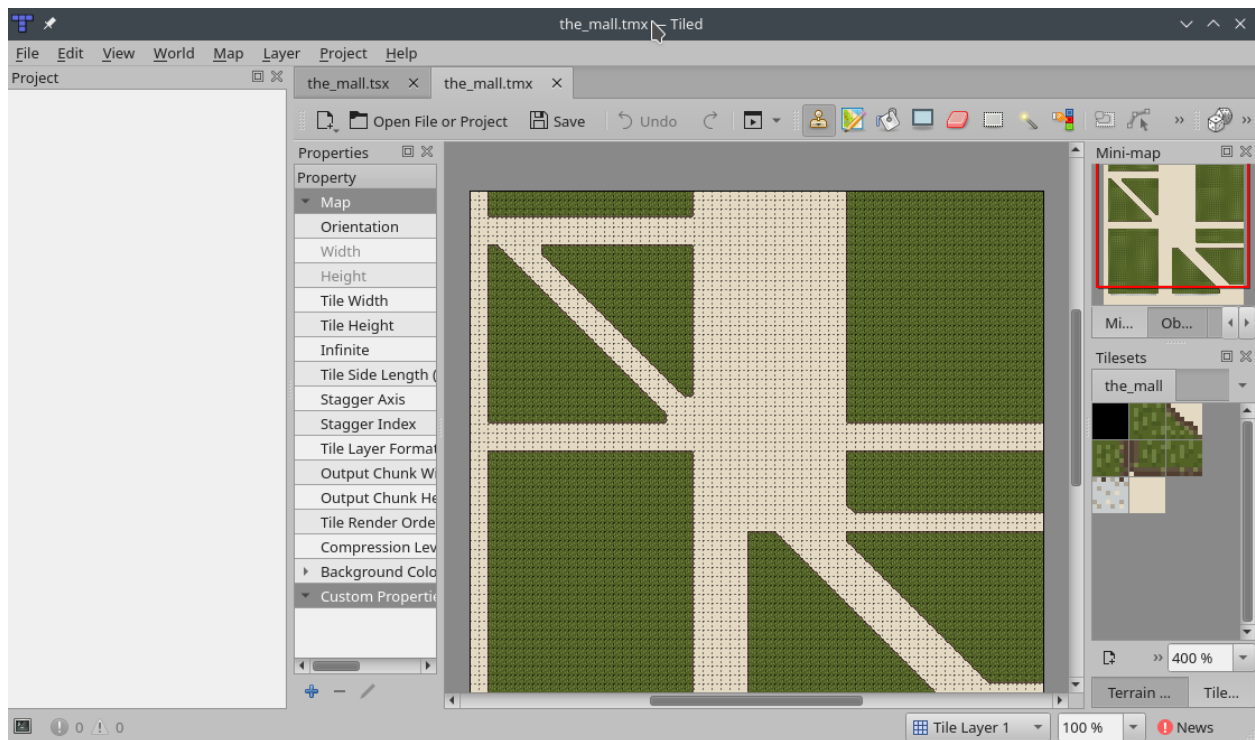
A trick/best-practice is to make tile ID 0 a completely transparent tile. To do this, fill an 8x8 pattern with the first (transparent) color of palette 0. This ensures that any tiles you do explicitly set in your game default to being invisible.

With your Tileset defined and saved. Open a “New Map”. A popup window will ask you for some settings. Use the settings defined below to be compatible with FP-GAME’s tile layer. If you want, you can reduce the “Map size” from 64x64 to whatever room size you want below that size. Map sizes greater than 64x64 cannot be loaded in all at once in FP-GAME, as the actual Tile layer is only 64x64 tiles in size.



(For advanced users): If you do wish to store a larger tilemap, you can, and the PPU User Library functions will let you load them, but you will be unable to write sections larger than 64x64 sections to the Tile RAM at a time. You can instead use this to create scrolling games with large worlds, but you must implement the tile loading logic yourself (by, for example, using `ppu_write_tiles_horizontal/vertical` to write new tiles from your large world to the region just outside of your screen's view before those tiles are visible). The User Library currently does not provide scrolling support for worlds larger than 64x64, but this is totally possible with FP-Game using programming tricks (like the method hinted at above).

Once you “Save As” in the settings window, a new window will open where you can paint tiles from your tileset onto a grid:



Importantly, you can press “x” to horizontally flip tiles, or “y” to vertically flip tiles. DO NOT PRESS “z” TO ROTATE TILES. Tile rotations are not supported in FP-Game hardware, but the above horizontal/vertical flip operations are. The converter script we will use shortly warns you of which tiles are rotated, and these are automatically ignored by FP-Game, so no need to painstakingly check for mistakes.

Once you are done, you can head to “File/Export As” and select “CSV Files” as the extension to use.

Now that the .csv file has been exported, we can run our converter script:

```
python tiled_csv_to_tilemap.py <src .csv file> <dest .tilemap> <palette ID>
```

Where:

- <src .csv file> is a file path to your recently exported .csv map.
- <dest .tilemap> is a file path and name for the .tilemap output. Do not include the file extension as it will be automatically added.
- <palette ID> refers to the palette to use with this .tilemap. Currently this tool only supports a single palette, so if you want to use multiple palettes in your tilemap, you will have to edit the .tilemap file manually, replacing the C value with your desired palette ID in the tile format: (PPP, C, M).

Congratulations, you now have a .tilemap format you can use in your game via the PPU User Library.

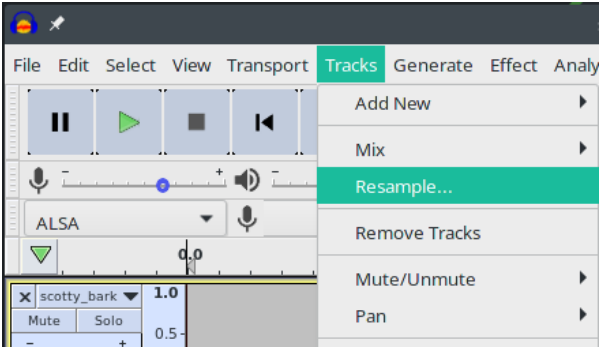
For examples on how to load these file formats into your game, see main.c under examples/techdemo/src in the FP-GAME user repository.

Supported Audio Pipeline

Audio export to FP-Game compatible formats is made easy by the free program Audacity:
<https://www.audacityteam.org/>

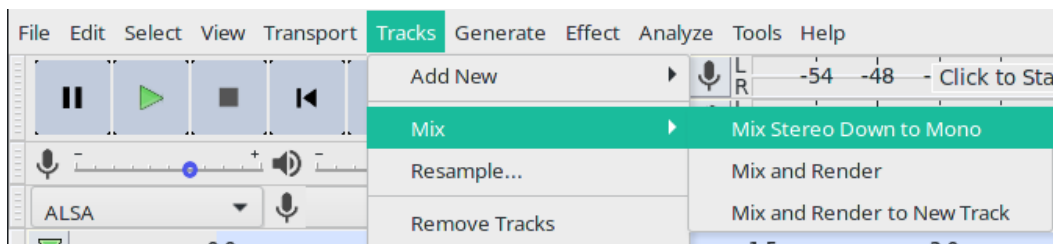
You can open your sound file in Audacity. There are a few simple steps to exporting to the FP-Game compatible format:

First, set your sound's sampling rate to 32000.

A screenshot of the Audacity application window. The 'Tracks' menu is open, showing options: 'Add New', 'Mix', 'Resample...', 'Remove Tracks', 'Mute/Unmute', and 'Pan'. The 'Resample...' option is highlighted in green. The background shows a track named 'scotty_bark' with a volume of 1.0 and a solo button.

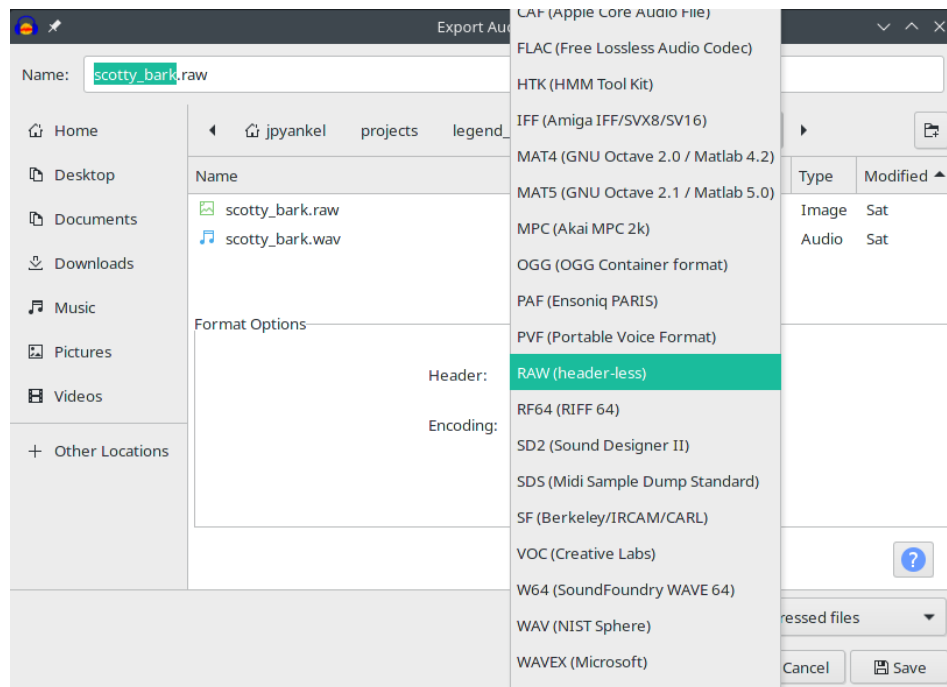
The end result will be that your track will (likely) be downsampled to 32KHz sampling rate. This is what FP-Game uses for its audio playback.

Next, if you have stereo audio, you must convert your audio down to mono using
“Tracks/Mix/Mix Stereo Down to Mono”

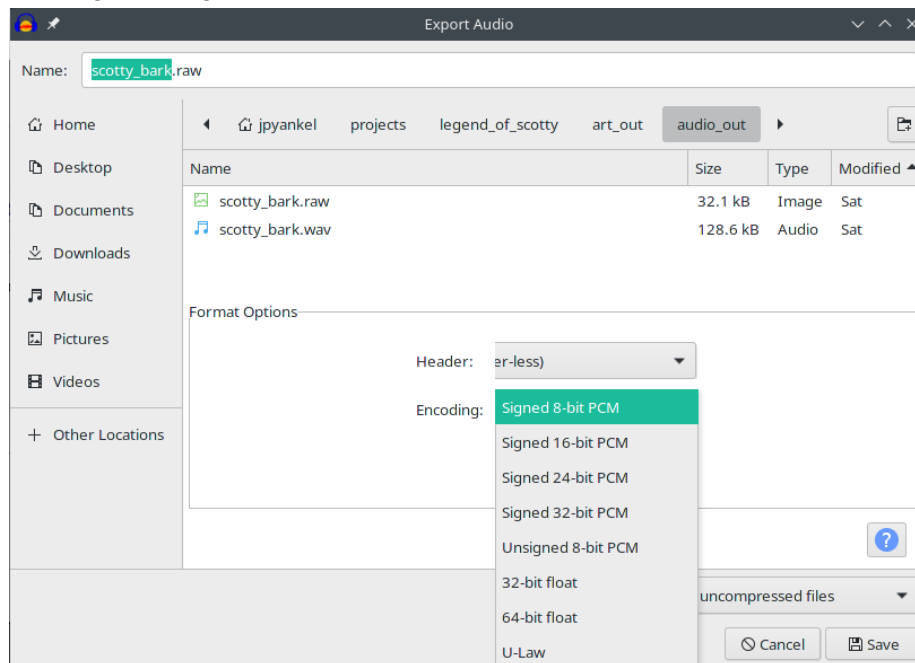


Next, head to “File/Export/Export Audio”.

First, you will need to set the “Header: “ as “RAW (header-less)”.



Then, set “Encoding” as “Signed 8-bit PCM”.



Save your exported file.

Rename the extension to .bin.

Congratulations, all you need to do to add the sound into your game is place your audio file in the bin/ folder and add <your sound>.o to “BINS =” section in config.mk. To access this sound in your game code, you must define two variables:

```
extern const int8_t _binary_bins_<your sound>_bin_start[];  
extern const int8_t _binary_bins_<your sound>_bin_end[];
```

It is important that the variables have that specific naming convention when used with our provided Makefiles. For the correct setup, view config.mk, bin/, and main.c in examples/techdemo/.

Your Very First Game Project Setup

In this section, we will walk you through compiling the included techdemo program. Once you complete these instructions, you can simply copy the examples/techdemo folder and repurpose it for your game. The most crucial step is installing the C toolchain, which will allow you to use our provided Makefile and compile your game for ARM. The steps are included below:

Toolchain Installation

FP-Game uses a prebuilt arm-linux-none-gnueabihf toolchain from ARM. The toolchain can be found at the following link:

<https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-a/downloads>

Download the “AArch32 target with hard float (arm-linux-none-gnueabihf)” .tar.xz file. Our download was titled: “gcc-arm-10.2-2020.11-x86_64-arm-none-linux-gnueabihf.tar.xz”. However, you may be presented with a different version depending on the time of reading. Download the version available to you. Note, the remainder of these instructions will assume the version listed above, so be sure to change any references to the toolchain to your specific version in the commands you run.

We recommend keeping a folder to contain your separate toolchains. For the remainder of these instructions, we will assume that you have a folder `~/toolchains` to store the extracted toolchain folders. If you want to create one, run `mkdir ~/toolchains` and follow the instructions below to extract your newly downloaded toolchain:

```
cd <path to your downloads folder>
tar -xvf gcc-arm-10.2-2020.11-x86_64-arm-none-linux-gnueabihf.tar.xz \
-C ~/toolchains
```

To enable Makefiles to see your new toolchain, you should add the following to your `.bashrc` (or `.zshrc`, or whatever you use):

```
# Toolchain for Cyclone V SoC (FP-Game)
export FPGAME_TC="$HOME/toolchains/\
gcc-arm-10.2-2020.11-x86_64-arm-none-linux-gnueabihf"
export PATH="$PATH:$FPGAME_TC/bin"
```

Ensure that you `source ~/.bashrc` (or `.zshrc`) or restart your terminal session so that you get the path changes and environment variables.

With this done, you are ready to run the provided Makefile.

Compiling techdemo

With the user repository (<https://github.com/FP-Game/fpgame-usr>) downloaded, “cd” into examples/techdemo/ and run `make clean` and then `make`. If you installed the toolchain as described in the section above, then this should work without errors. The result is the techdemo binary should be output into the current directory.

Setup Your Own Game Folder

This step is a matter of copying the suggested files from examples/techdemo. These files are:

Makefile	Our provided Makefile. It is a general Makefile.
config.mk	Configuration for Makefile. Tells makefile to target ARM and include our FP-Game user libraries.
sdk.mk	Tells the Makefile where the toolchain is installed.
usr/	This folder contains the FP-Game library, which you will want to use in your game. It also contains the library documentation (in index.html) which you can open in your web browser.

Note, when you copy config.mk over, be sure to replace

```
BINS = bins/scottybark.o
```

with

```
BINS =
```

When you build your project, use the recommended folder structure:

src/	Any .c source files for your game.
src/inc/	Any custom header files you want included.
bins/	Add your audio .bin files here. Be sure to explicitly add them to the line <code>BINS = <PATHS TO YOUR BINS></code> in config.mk afterwards.

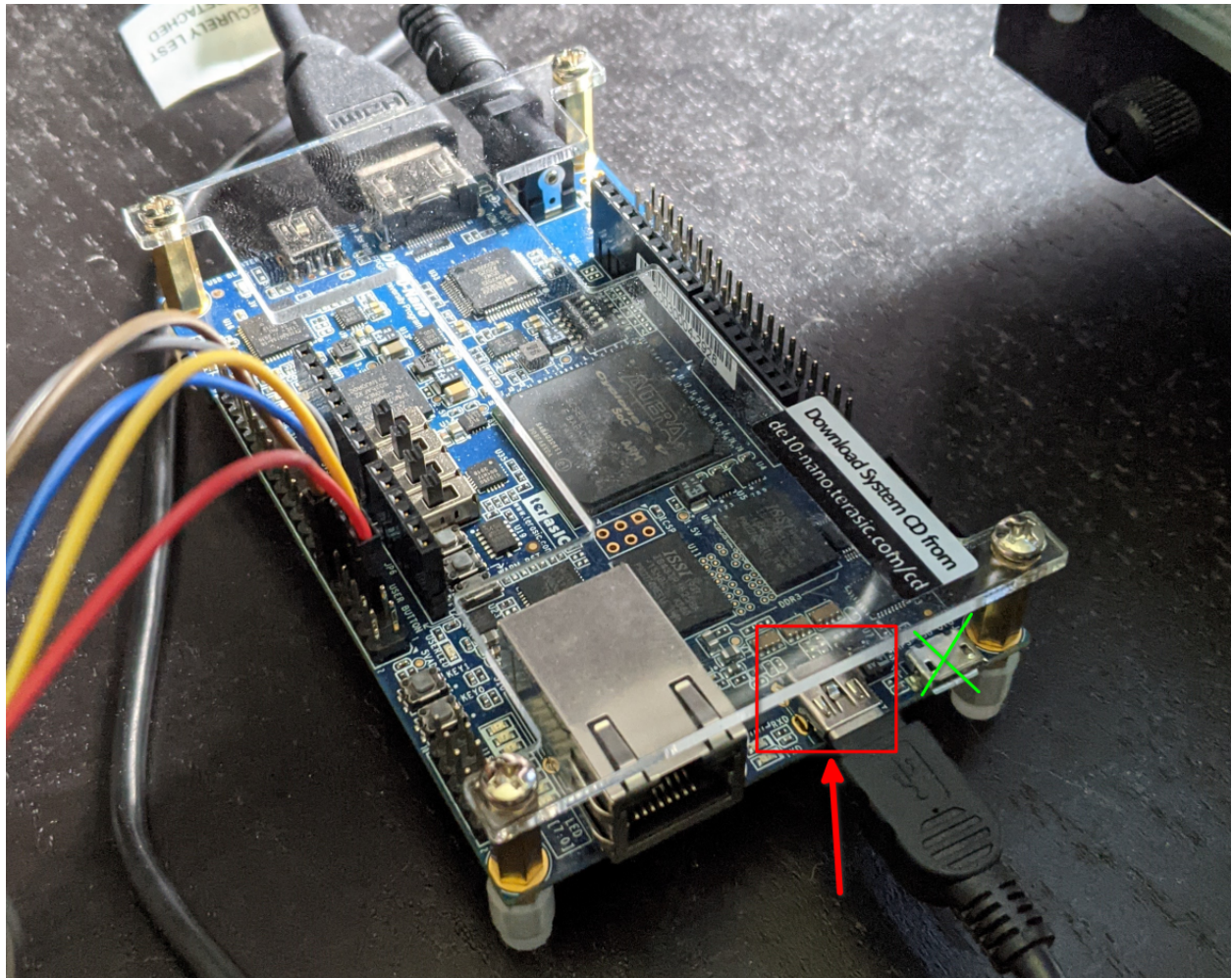
Auto-loading Your Game on FP-GAME

FP-GAME uses a systemd service to autorun your game on boot. Copy your game's executable file to /home/root in the SD Card. Rename the executable as "game". Also remember to copy over any assets or asset folders your game requires into the home/root directory.

You may wish to delete or move the pre-included "techdemo" game files first to avoid confusing them with your own.

Debugging Over Minicom

You can debug your programs and access Linux on FP-GAME using a serial connection over USB.



Above: You must use the USB-Mini B port when connecting the DE10-Nano to your computer.

The instructions included below are for running minicom on a Linux PC to talk to the DE10-Nano device, but other applications can be used, such as screen or picocom. Download Minicom using whatever package manager or method is available to you.

Then, run `sudo minicom -s` to begin configuration.

Head into Serial port setup and configure it to use the newly found ttyUSB device in /dev/. Mine was ttyUSB0:

```
A - Serial Device      : /dev/ttyUSB0
B - Lockfile Location  : /var/run
C - Callin Program     :
D - Callout Program    :
E - Bps/Par/Bits       : 115200 8N1
F - Hardware Flow Control : No
G - Software Flow Control : No
H - RS485 Enable       : No
I - RS485 Rts On Send  : No
J - RS485 Rts After Send : No
K - RS485 Rx During Tx  : No
L - RS485 Terminate Bus : No
M - RS485 Delay Rts Before: 0
N - RS485 Delay Rts After : 0

Change which setting? 0
```

Also be sure that Bps/Par/Bits is set to 115200 8N1.

Lastly, be sure to “Save setup as dfl” before exiting.

Now, whenever you run `sudo minicom`, you should be able to see the serial console output of the DE10-Nano.

You will be met with a login prompt. Type root as the username to gain access:

[illegible]

From here, you can run gdb on your program and/or run your program with printf debugging and view the console outputs. Either way, good luck with your debugging!

Modifying FP-GAME

If you wish to make modifications to FP-GAME or even just build it from source, visit the FP-GAME Source Repository: <https://github.com/FP-GAME/fpgame-src>

A guide is included there for building from source. In addition, HDL, Kernel Modules, and Library sources are included for your learning and or experimentation.