IMP Project 2024 - Tetris

Nguyen Le Duy xnguye27

10. December 2024

1 Introduction

This project is a Tetris game[1] implemented on the Raspberry Pi Pico 2[2] using the Rust programming language[3] and the `rp235x-hal`[4] library. The game is displayed on a 128x64 ssd1306 OLED I2C display and is controlled by a joystick and optional buttons.

2 Installation

2.1 Physical requirements

- Raspberry Pi Pico 2
- Micro USB cable
- 128x64 ssd1306 I2C display
- Joystick with integrated button (e.g. Keyes KY-023)
- (Optional) Buzzer

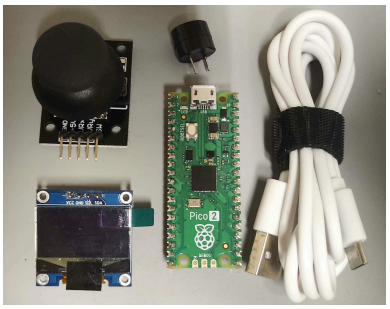

Abb. 1: Required components for the project

2.2 Wiring

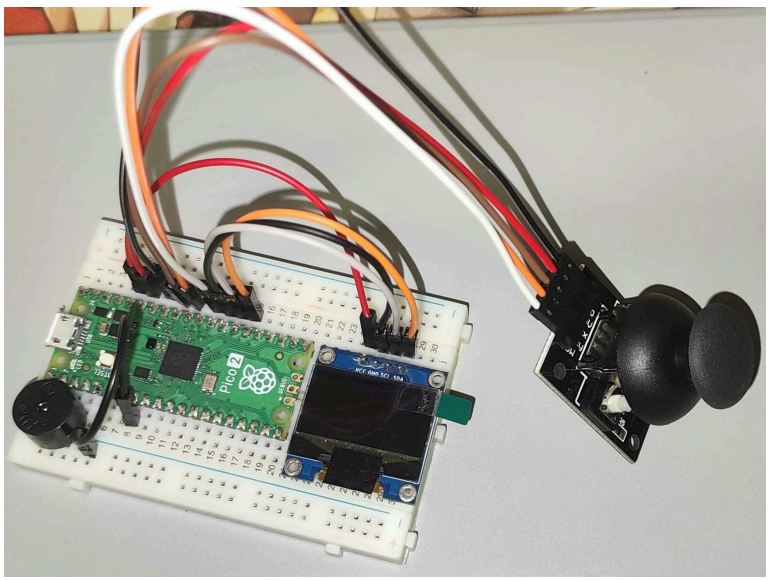

Abb. 2: Wired components

The components are wired as follows:

Tab. 1: Wiring

Pin	Name	Wired To
	Joystick	
35	ADC_VREF	VCC of the joystick
33	AGND	GND of the joystick
32	ADC1	x-axis of the joystick
31	ADC0	y-axis of the joystick
29	GP22	Button of the joystick
	Display	
36	3v3(OUT)	VCC of the display
38	GND	GND of the display
27	I2C0 SCL	SLC of the display
26	I2C0 SDA	SDA of the display
	Buzzer	
2	GP1	Buzzer +
8	GND	Buzzer -

Note that other than the joystick, the GND of other components can be connected to any GND pin. The joystick needs AGND and ADC_VREF for accurate sampling[1].

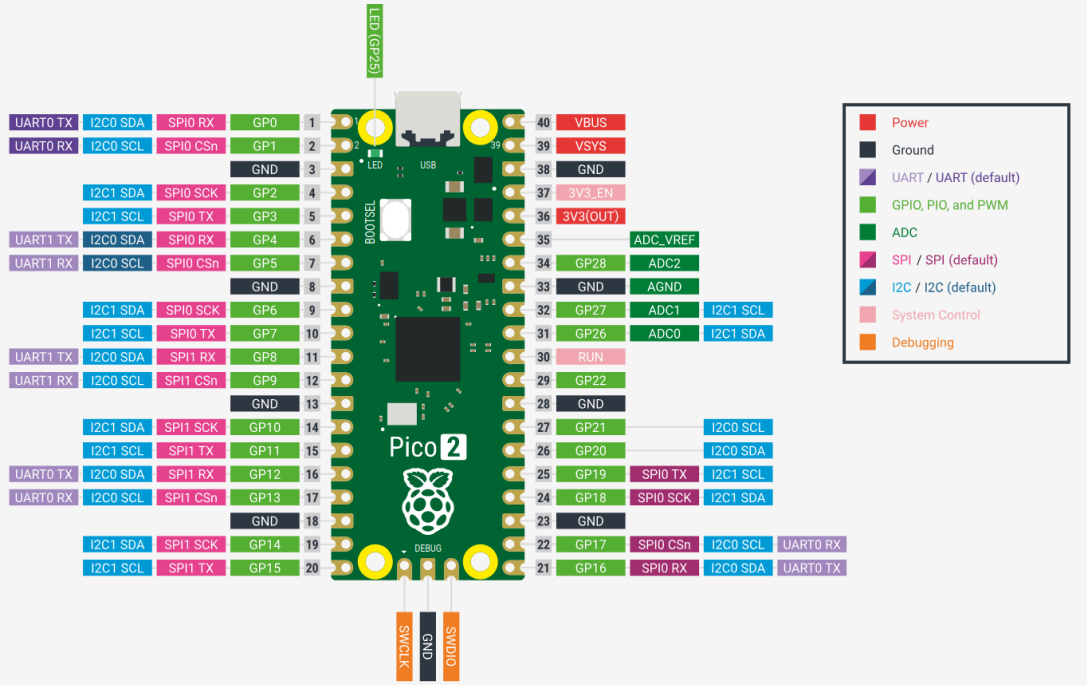

Abb. 3: Raspberry Pi Pico 2 pinout for reference[2]

[1]Section 3.1 in the Raspberry Pi Pico 2 Datasheet[5]

2.3 Pre-requisites

In order to build and flash the project for the Raspberry Pi Pico 2, the following tools need to be installed:

- Rust compiler - for compiling the Rust code
- picotool - for flashing the compiled binary to the Pico 2

Then grab the compilation target for the Raspberry Pi Pico 2

```
$ rustup target add thumbv8m.main-none-eabihf
# or
$ make add-target
```

Note that the `riscv32imac-unknown-none-elf` can also be used as the RP2350 chip used by the Pico 2 support both ARM and RISC-V architectures. However, it will not work for this project as it uses a specific ARM Cortex-M instruction.

2.4 Building the project

Navigate to the root folder of the project (where the `Cargo.toml` file is located) and run the following command:

```
$ cargo build --release --target=thumbv8m.main-none-eabihf
# or
$ make build
```

This will compile the project into a binary file located at `target/thumbv8m.main-none-eabihf/release/tetris`.

2.5 Flashing the binary to the Pico 2

Now that the binary is compiled, it needs to be flashed to the Pico 2. This can be done using the `picotool` utility.

Connect the Pico 2 to the computer with a micro-USB cable and put it into bootloader mode by holding down the `BOOTSEL` button while plugging in the USB cable.

Then run the following command to flash the binary to the Pico 2:

```
$ sudo picotool load -u -v -x -t elf target/thumbv8m.main-none-eabihf/release/tetris
\# or
$ make flash
```

Note that the `sudo` command is required to access the USB device.

The Tetris start screen should now be displayed.

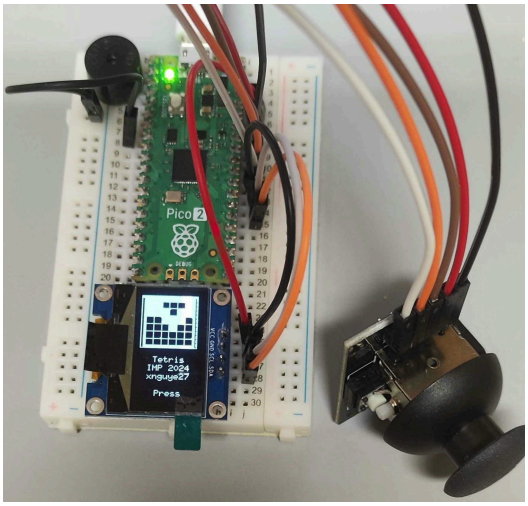

Abb. 4: Startup screen of the Tetris game

3 Usage

On the startup screen, the game has not yet started. Press the joystick button to start the game.

The current score and the next block are displayed at the top of the screen. The current score is increased by 1 for each line cleared.

Blocks will start spawning from the top of the screen and the player must move the blocks to create a line without gaps. When a line is created, it is cleared and all blocks above it fall down.

The game is controlled by the joystick and its button:

Left	Move the block to the left.
Right	Move the block to the right.
Up	Rotate the block.
Down	Move the block down.
Press the joystick button	Drop the block to the bottom.

The game ends when the block reaches the top of the screen. You will now be taken to the game-over screen, which displays the final score. Press the joystick button to restart the game.

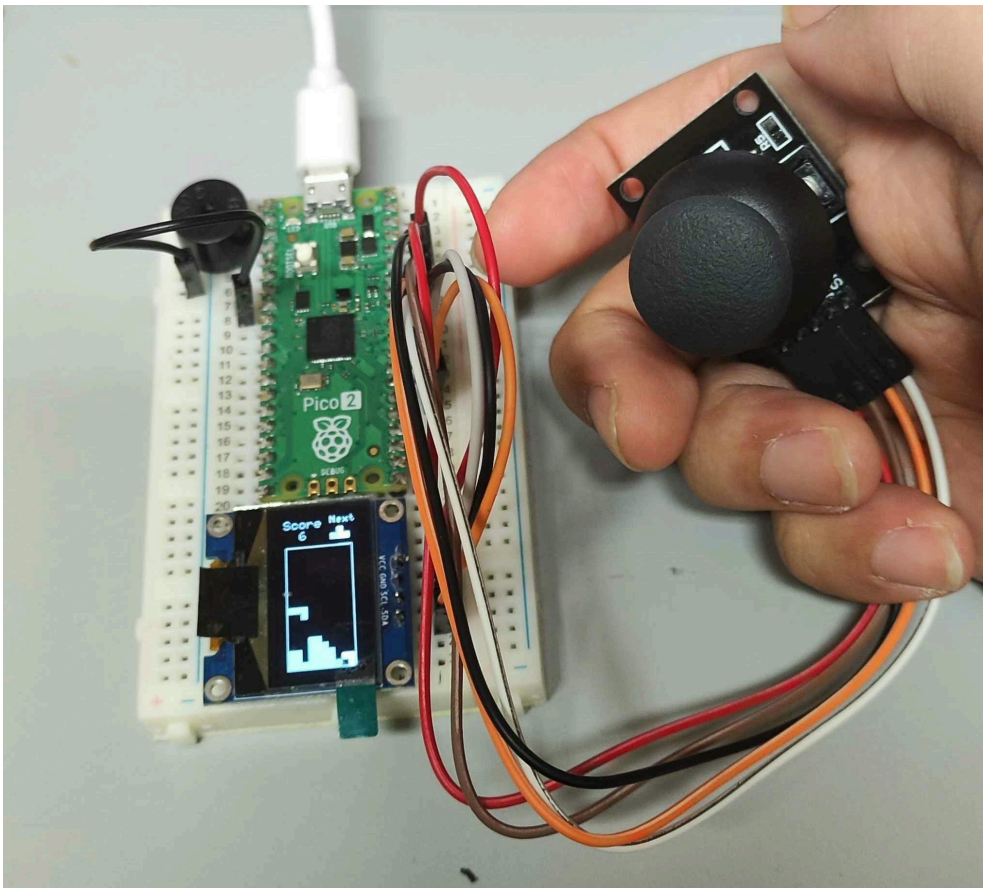

Abb. 5: Playing the Tetris game

4 Implementation

The project is implemented based on several open-source libraries, including:

- `rp235x-hal` - A Rust library for the Raspberry Pi Pico 2.
- `ssd1306` - Rust driver for the ssd1306 OLED display.
- `embedded-hal` - Provides an abstraction layer for embedded devices.
- `embedded-graphics` - Provides API for drawing graphics on embedded systems that is wrapped by the `embedded-hal`.
- `heapless` - Provides heap-like data structures that can be used in the embedded environment, since heap allocation is not available for most of embedded systems.
- `critical-section` - Provides a way to execute code in a critical section.
- `cortex-m` - Provides low-level access to the ARM Cortex-M processor.
- `rand` - Provides generic trait for a random number generation.

Also, in Rust terminology, a `library` is called a `crate`, and I may interchange the two terms in the following sections.

The code is organized into 5 main modules:

- `main`: The heart of the program, handles microcontroller initialization, interrupts, timing and the game loop.
- `display`: A wrapper around the `ssd1306` crate to handle the display. It provides a simple API to draw the tetris game on the screen.
- `input`: Handles input from the joystick and the button.
- `tetris`: Handles the game logic. It is written as generically as possible for easy integration with other systems.
- `bgm`: Handles the background music.

4.1 Game logic

The game is implemented based on the classic Tetris game with support for hard drop, additional features like hold piece, T-spin, ghost piece, etc. are not implemented.

4.1.1 Random Number Generator

> Random Generator generates a sequence of all seven one-sided tetrominoes (I, J, L, O, S, T, Z) permuted randomly, as if they were drawn from a bag. Then it deals all seven tetrominoes to the piece sequence before generating another bag. There are 7!, or 5,040, permutations of seven elements, and it is believed that Tetris assigns a nearly equal probability to each of these, making it much less likely that the player will get an obscenely long run without a desired tetromino.
>
> — TetrisWiki[6]

Raspberry Pi Pico 2 is equipped with a TRNG[2] module via its Ring Oscillator[3]. The `rp235x-hal` implemented the `RngCore` trait from the `rand` crate for it. So it can use all the functionality of the `rand` crate, in this project the shuffle function[4].

4.2 Display

Thanks to the abstraction layer of the `embedded-hal` crate, there is a crate to handle the `ssd1306` display using the `I2c` interface, which is also implemented by the `rp235x-hal`, so the integration is flawless.

The `ssd1306` crate also implements `Drawable` trait from the `embedded-graphcis` crate, so it has all the functionality that the `embedded-graphics` provides for drawing on the screen, from shapes to text.

[2] True Random Number Generator

[3] Section 8.3 in the rp2350 datasheet[7]

[4] https://docs.rs/rand/latest/rand/seq/trait.SliceRandom.html#tymethod.shuffle

4.3 Input

The input is handled by the `input` module, which is responsible for reading the input from the joystick and the button.

4.3.1 Button

The game has a button that is connected to the GPIO pins of the Raspberry Pi Pico 2 with a pull-up resistor. This GPIO pin is then opened for interrupt handling, which is handled by the interrupt handler (function `IO_IRQ_BANK0` in the `main` module). It responds to the low edge of the signal, i.e. when the button is pressed, the signal is pulled down to ground.

The interrupt is enabled by the following code in the `main` module:

```
unsafe {
    cortex_m::peripheral::NVIC::unmask(hal::pac::Interrupt::IO_IRQ_BANK0);
}
```

This function is a handy way to write interrupt number to the ISER register of the processor.

Safety: it is called only once at program initialization, so it is safe to use the `unsafe` block.

4.3.1.1 Debouncing

When a button is pressed, it is not guaranteed that the signal will be stable, it may have some *noise* that causes the signal to *fluctuate*, making the system think that the button has been pressed several times. This is called bouncing.

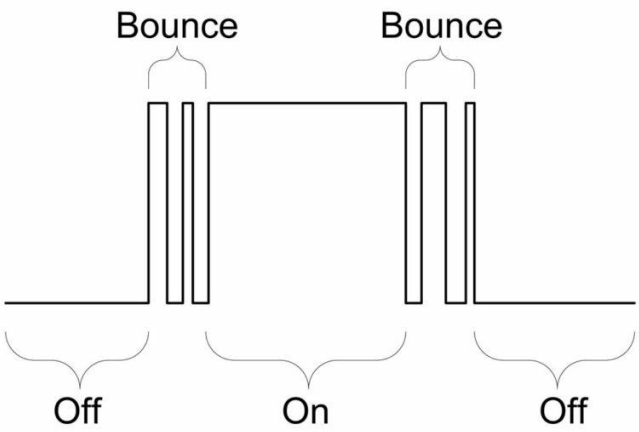

Abb. 6: Button bouncing[8]

To prevent this, a simple debouncing algorithm is implemented in the `input` module, allowing the button to be pressed only once every 130 milliseconds.

4.3.2 Joystick

The joystick works by having 2 internal potentiometers for the X and Y axis, the voltage of the output pin is proportional to the position of the joystick. The Raspberry Pi Pico 2 has 3 ADC modules that can be used to convert the analog signal to a digital value.

They work by reading the value of the ADC pin of the X and Y axis of the joystick and then mapping those values to an action in the game.

However, the joystick struct defined in the `input` module only handles the logic of the joystick like the center position, the dead zone and the last input to prevent the game from being too sensitive to the joystick movement, It does not directly interact with the ADC modules due to the complex abstraction layer of the `rp235x-hal` crate and the strictness of the Rust compiler.

The main loop will read the joystick values at the beginning of each loop, then use the `Joystick` struct to determine the action based on those values.

To make sure that the joystick does not move by itself when it is in the center position, a `DEADZONE` is defined as the radius in the center position, if the `X` and `Y` values are within the `DEADZONE`, the joystick is considered to be in the center position. The calculation to check if the joystick is in the dead zone or not is done by the Euclidean distance between the current position and the center position of the joystick.

$$(x - x_{\text{center}})^2 + (y - y_{\text{center}})^2 \leq \text{DEADZONE}^2$$

The deadzone is squared to avoid the computationally expensive square root calculation.

The pico is set to read the ADC value in 12-bit resolution, which means the value will be between 0 and 4095, so the radius of the `DEADZONE` is set to 1000, which is about 50% of the maximum value.

4.4 Main loop

There is a constant `REFRESH_RATE_NS` in the main module that is set to $\frac{1000000000}{60} - 4000$ nanoseconds, This is used for timing between two loops in the main loop. In each loop it will do 4 things:

1. Read the inputs from the joystick and process them if necessary.
   - The ADC sampling is done in 96 ADC clock cycles (48Mhz), which is 2 microseconds, and it is done 2 times (for X and Y axis), so the total time is 4 microseconds.
2. Check if enough time has elapsed to cause the current tetromino to fall.
3. Draw the screen based on the changes of the game state in between loop by interrupt handler or pre-draw code, it does not completely refresh the screen unless needed.
4. Sleep for the remaining time to keep the refresh rate at 60 FPS

4.5 Critical Section

The MCU must take action on the game state based on the pressed button in the interrupt handler, or after some amount of time has passed to drop a block down by 1 through the main loop. Thus, there are more than 1 place of code that modify the same piece of memory and is not executed in a known order.

By the design of the Rust language, this is not allowed due to its memory safety policy, so the code will not compile if it is not handled properly.

The `critical-section` crate is used to handle this problem. It uses the processor's spinlocks[5] to ensure that the critical section is safely executed, This means that no other code can use the game state while it is in the critical section, thus preventing concurrency problems and keeping the Rust compiler happy.

4.6 BGM

The background music is implemented using the Raspberry Pi Pico 2's PWM module[6]. The PWM module is used to generate a square wave signal with a specific frequency.

The Tetris theme song is played only when the game is started and stopped when the game is finished. The song is played in a loop and the frequency of the notes is defined in the `bgm` module.

The song requires 9 musical notes to be played, each note can be represented as a signal frequency in Hz, they are:

- A4 -> 440 Hz
- B4 -> 493.88 Hz
- G#4 -> 415.30 Hz
- C5 -> 523.25 Hz
- D5 -> 587.33 Hz
- E5 -> 659.25 Hz
- F5 -> 698.46 Hz
- G5 -> 783.99 Hz
- G#5 -> 830.61 Hz
- A5 -> 880 Hz

[5]Section 3.1.4 in the RP2350 Datasheet[7]
[6]Section 12.5 in the RP2350 Datasheet[7]

However, changing from one frequency to another without a pause will result in a very high pitch due to the sudden change in frequency. So a small pause is added between each note to make the transition smoother. For this project, the pause duration is set to $\frac{1}{64}$ of the total note duration, where the total note duration is 1 minute $* \frac{4}{\text{BMP}} \Rightarrow 60000 * \frac{4}{144}$ milliseconds.

To generate these frequencies using the Raspberry Pi Pico 2′s PWM module, there are 3 factors that affect the frequency of the PWM signal:

Abbr	**Name**	**Value**
CLK	The clock frequency of the Raspberry Pi Pico 2 itself.	Fixed at 150 MHz
CLK_DIV	The clock divider of the PWM module.	Variabled from 0 to 255, where 0 is interpreted as 256.
CNT	The maximum value of the PWM counter.	Variabled from 1 to 65535

They work by counting from 0 to (and including) `CNT` and then resetting to 0. The counting speed is based on the `CLK_DIV` values with the formula $\frac{\text{CLK}}{\text{CLK\_DIV}}$. By default (`CLK_DIV` $= 1$) the counter increments by 1 every clock cycle, if set to 2 it increments by 1 every 2 clock cycles, and so on. So the final formula to calculate the frequency of the PWM signal is

$$\text{Frequency} = \frac{\text{CLK}}{\text{CLK\_DIV} * \text{CNT}} = \frac{150000000}{\text{CLK\_DIV} * \text{CNT}}$$

By brute forcing the values of `CLK_DIV` from 1 to 256 and `CNT` from 1 to 65535, we can find the values that are closest to the desired frequency. These values are then hardcoded into the `bgm` module to generate the notes.

Since the RP2350 chip used by the Pico 2 has 2 cores, the PWM module can be run on the second core while the main game loop is running on the first core. This allows the music timing not to interfere with the game timing.

The RP2350 also supports `FIFO mailbox`[7] system to communicate between the 2 cores, which is used to tell the second core to start and stop the music based on the game state.

[7]Section 3.1.5 in the RP2350 Datasheet [7]

Tab. 2: Calculated values for the PWM module to generate the musical notes.
The error is the difference between the desired frequency and the calculated frequency.
The program is included in the appendix Section 7.2.

Note	**Frequency**	**CLK_DIV**	**CNT**	**Error**
A4	440.00 Hz	10	34091	0.001173 Hz
B4	493.88 Hz	181	1678	0.000809 Hz
Gs4	415.30 Hz	11	32835	0.000361 Hz
C5	523.25 Hz	5	57334	0.000270 Hz
D5	587.33 Hz	9	28377	0.000115 Hz
E5	659.25 Hz	4	56883	0.002070 Hz
F5	698.46 Hz	6	35793	0.000593 Hz
G5	783.99 Hz	3	63776	0.003979 Hz
Gs5	830.61 Hz	5	36118	0.000776 Hz
A5	880.00 Hz	5	34091	0.002347 Hz
REST	60000.00 Hz	1	2500	0.000000 Hz

5 Compile for Raspbeery Pi Pico (1st version)

Since this project was written in barebone programming specifically for the Raspberry Pi Pico 2, it is not as portable as a program written in C/C++ or Python using the official SDK.

However, it is still possible to compile the project for the Raspberry Pi Pico, but it will require some changes to the code.

1. Change the `rp235x-hal` dependency in the `Cargo.toml` file to `rp2040-hal` (the git repository is the same).
2. Change the line `use rp235x_hal as hal;` to `use rp2040_hal as hal;` in the `main.rs` file.
3. Recalculate the PWM values for the RP2040 chip, since its clock frequency is 125 MHz instead of 150 MHz.

The code to replace the pmw values for the Raspberry Pi Pico can be found in the appendix Section 7.2.1.

This should be compatible with the Raspberry Pi Pico 1, but is untested as I do not have the hardware to test it.

The process of compiling and flashing is the same as for the Raspberry Pi Pico 2 in the Section 2, but the target architecture is different. Install the `thumbv6m-none-eabi` architecture target for the Rust compiler first, if not already done:

```
$ rustup target add thumbv6m-none-eabi
```

Then compile the project using the following command:

```
$ cargo build --release --target=thumbv6m-none-eabi
```

And flash the binary to the Raspberry Pi Pico 1 with the following command:

```
$ sudo picotool load -v -x -t elf target/thumbv6m-none-eabi/release/tetris
```

6 Conclusion

This project is a fun and challenging experience. It combines the knowledge of Rust programming language, embedded systems, and music theory. Learning how to work with the Raspberry Pi Pico 2, pin configurations, ADC and PWM modules, all of them combine with the strictness of the Rust language, make this project a great learning experience as they are very practical and useful in the real world.

Bibliography

[1] "Tetris." 2024.

[2] R. P. Foundation, "Raspberry Pi Pico 2." 2024.

[3] T. R. P. Developers, "The Rust Programming Language." 2024.

[4] T. rp-rs team, "rp-hal: Rust support for the RP2040 chip." 2024.

[5] R. P. Foundation, "Raspberry Pi Pico 2 Datasheet." 2024. [Online]. Available: https://datasheets.raspberrypi.com/pico/pico-2-datasheet.pdf

[6] "Random Generator." 2024.

[7] R. P. Foundation, "RP2350 Datasheet." 2024. [Online]. Available: https://datasheets.raspberrypi.com/rp2350/rp2350-datasheet.pdf

[8] "GPIO Basics." 2024.

7 Appendix

7.1 Glosarry

Terminology	Meaning
RNG	Random Number Generator
Crate	A Rust library
Trait	Can be thought at Interface in other programming language terminology
BGM	BackGround Music
ADC	Analog-to-Digital Converter
PWM	Pulse-width modulation
GPIO	General-purpose Input/Output

7.2 Rust program to calculate the PWM values

```rust
const BASE_CLK: u32 = 150_000_000; // Default by the Pico 2
const CLK_DIV_MIN: u32 = 1;
const CLK_DIV_MAX: u32 = 256;
const CNT_MIN: u32 = 1;
const CNT_MAX: u32 = 65536;

struct Note {
    name: &'static str,
    frequency: f64,
    cnt: u32,
    clk_div: u32,
    error: f64,
}

impl Note {
    fn new(name: &'static str, frequency: f64) -> Self {
        Self {
            name,
            frequency,
            cnt: 1,          // Default by the Pico 2
            clk_div: 65536,  // Default by the Pico 2
            error: f64::MAX, // MAX as unknown error
        }
    }
}

fn main() {
    let mut notes = [
        Note::new("A4", 440.00),
        Note::new("B4", 493.88),
        Note::new("Gs4", 415.30),
        Note::new("C5", 523.25),
        Note::new("D5", 587.33),
        Note::new("E5", 659.25),
```

```
        Note::new("F5", 698.46),
        Note::new("G5", 783.99),
        Note::new("Gs5", 830.61),
        Note::new("A5", 880.00),
        Note::new("REST", 60000.0),
    ];

    // Brute force to find the best clock divider and counter value
    for clk_div in CLK_DIV_MIN..=CLK_DIV_MAX {
        for cnt in CNT_MIN..=CNT_MAX {
            for note in notes.iter_mut() {
                let frequency = BASE_CLK as f64 / (clk_div as f64 * cnt as f64);
                let error = (frequency - note.frequency).abs();
                if error < note.error {
                    note.error = error;
                    note.cnt = cnt;
                    note.clk_div = clk_div;
                }
            }
        }
    }

    for note in notes {
        println!(
            "{} ({} Hz) -> CLK_DIV: {}, CNT: {}, Error: {:.6} Hz",
            note.name, note.frequency, note.clk_div, note.cnt, note.error
        );
    }
}
```

7.2.1 PWM values for the Raspberry Pi Pico 1st version

Tab. 3: PWM values for the Raspberry Pi Pico 1st version.
The program is provided in the appendix Section 7.2 with modified BASE_CLK to 125_000_000.

Note	Frequency	CLK_DIV	CNT	Error
A4	440	5	56818	0.0014080045056061863
B4	493.88	6	42183	0.0001589897984217714
Gs4	415.3	9	33443	0.00032858561996818025
C5	523.25	4	59723	0.0010004520871689238
D5	587.33	4	53207	0.0012650591088458896
E5	659.25	3	63203	0.0014068425022060183
F5	698.46	3	59655	0.0005928533511223577
G5	783.99	3	53147	0.0009382153900787671
Gs5	830.61	3	50164	0.001063976822706536
A5	880	5	28409	0.0028160090112123726
REST	60000	1	2083	9.601536245798343

```
impl Note {
    pub fn frequency(&self) -> Frequency {
        match self {
            Self::A4 => Frequency {
                clk_div: 5,
                cnt: 56818,
            },
            Self::B4 => Frequency {
                clk_div: 6,
                cnt: 42183,
            },
            Self::Gs4 => Frequency {
                clk_div: 9,
                cnt: 33443,
            },
            Self::C5 => Frequency {
                clk_div: 4,
                cnt: 59723,
            },
            Self::D5 => Frequency {
                clk_div: 4,
                cnt: 53207,
            },
```

```
            Self::E5 => Frequency {
                clk_div: 3,
                cnt: 63203,
            },
            Self::F5 => Frequency {
                clk_div: 3,
                cnt: 59655,
            },
            Self::G5 => Frequency {
                clk_div: 3,
                cnt: 53147,
            },
            Self::Gs5 => Frequency {
                clk_div: 3,
                cnt: 50164,
            },
            Self::A5 => Frequency {
                clk_div: 5,
                cnt: 28409,
            },
            Self::REST => Frequency {
                clk_div: 1,
                cnt: 2083,
            },
        }
    }
}
```