从零开始实现GameBoy模拟器 #13 脉冲音频通道

銀葉吉祥
浙江大学 软件工程硕士

目录 收起

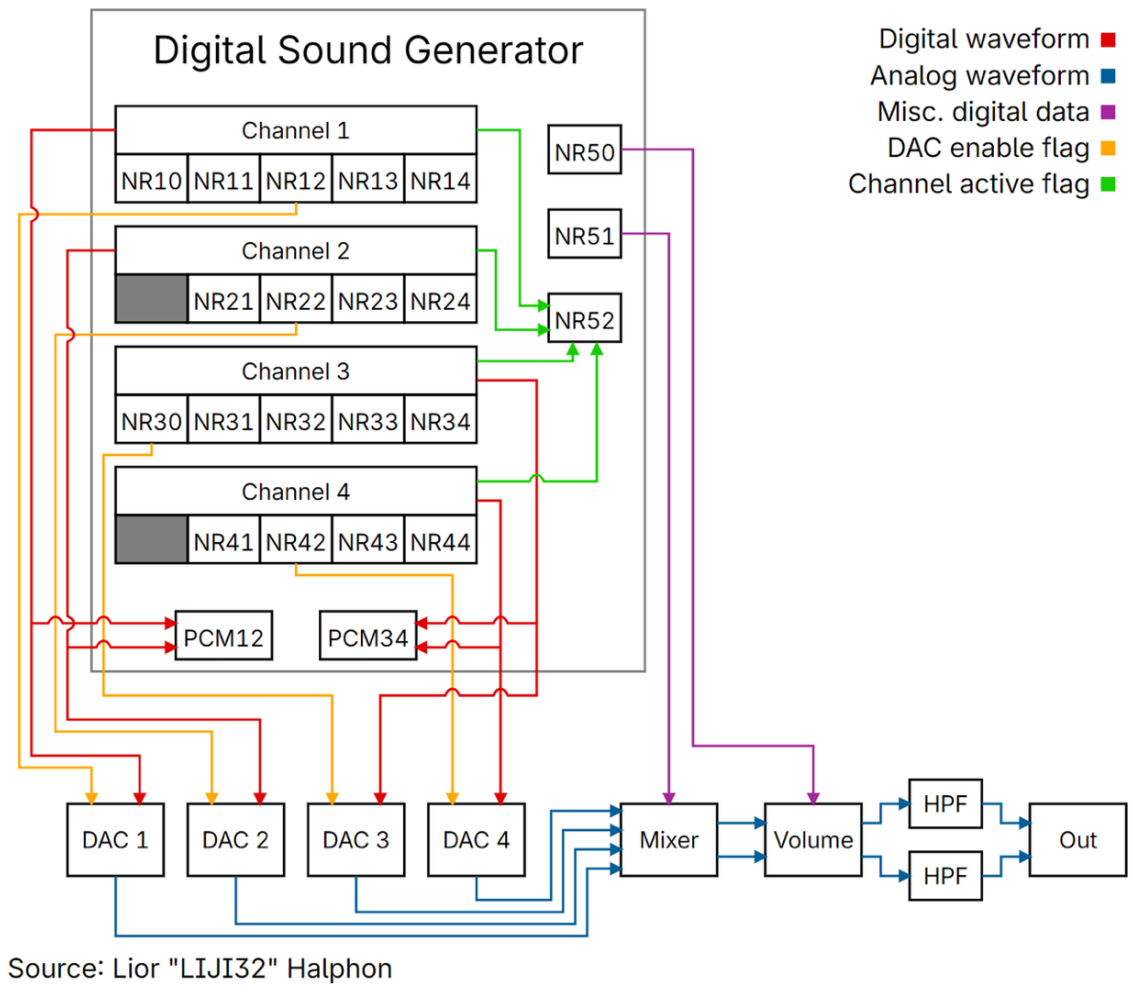
- APU架构
  - 脉冲波形通道原理
  - 脉冲波形信号生成
  - sweep
  - envelope
  - length timer
  - DAC
  - 混音器
- 实现APU
  - 实现DIV-APU
  - 实现通道1的脉冲波形生成
  - 实现通道1的sweep功能
  - 实现通道1的envelope功能
  - 实现通道1的length timer
  - 实现mixer
  - 输出音频
  - 实现通道2的音频输出

欢迎来到从零开始实现GameBoy模拟器第十三章。在本章中，我们将开始APU部分的实际编程。我们将首先讲解APU的硬件架构，给出APU相关的控制寄存器，然后编程实现通道1和通道2的信号输出，这两个通道使用脉冲信号产生器电路输出方波信号，产生8bit音乐的独特听感。那么，让我们开始吧！

> 本章节代码已经上传至项目仓库，名称为LunaGB-13，已经下载项目代码的读者可以更新代码库来查阅最新代码。同时，欢迎大家加入GameBoy模拟器开发交流群：33854734
> 2024.3.18更新：APU::tick混音时添加了对DAC状态的判断。

APU架构

下图展示了GameBoy APU的硬件架构：



Source: Lior "LIJI32" Halphon

来源：https://gbdev.io/pandocs/Audio_details.html

APU主要分为五个组件构成：四个硬件波形生成器负责生成对应的音频波形信号，一个混音器负责将四个通道的声音混合，并处理成最终的音频输出信号。其中，四个硬件波形生成器生成的均为数字信号，每一个生成器在输出信号前都会通过一个DAC（数模转换器）转换成模拟信号，然后才输入混音器，因此混音器处理的完全是模拟信号。五个组件由不同的寄存器控制，其在任天堂官方文档中以NRXX的方式命名，这些寄存器包括：

1. NR10~NR14：控制通道1的信号生成器的行为。
2. NR21~NR24：控制通道2的信号生成器的行为。
3. NR30~NR34：控制通道3的信号生成器的行为。
4. NR41~NR44：通知通道4的信号生成器的行为。
5. NR50~NR52：控制APU的混音器的行为。

其中，图中的PCM12和PCM34寄存器用于监听APU的数字信号输出，其只在GameBoy Color以及后续机型中存在，且从未在官方文档中提及，因此我们在此不予考虑。

脉冲波形通道原理

通道1和通道2均为脉冲信号通道，其硬件电路布局与寄存器用法基本一致，唯一的区别在于通道2阉割了通道1的sweep功能，因此也没有NR10对应的NR20寄存器。在下文中，我们将以通道1为例讲解脉冲波形通道的硬件原理。

我们首先来看一下NR10~NR14寄存器的含义。NR10（0xFF10）寄存器的含义如下表所示：

7	4~6	3	0~2
-	sweep时间间隔	sweep方向	sweep步长

NR11（0xFF11）寄存器的含义如下表所示：

6~7	0~5
波形选择	length timer初始值

NR12（0xFF12）寄存器的含义如下表所示：

4~7	3	0~2
初始音量	envelope方向	envelope时间间隔

NR13（0xFF13）寄存器的含义如下表所示：

0~7
sample counter初始值（低8位）

NR14（0xFF14）寄存器的含义如下表所示：

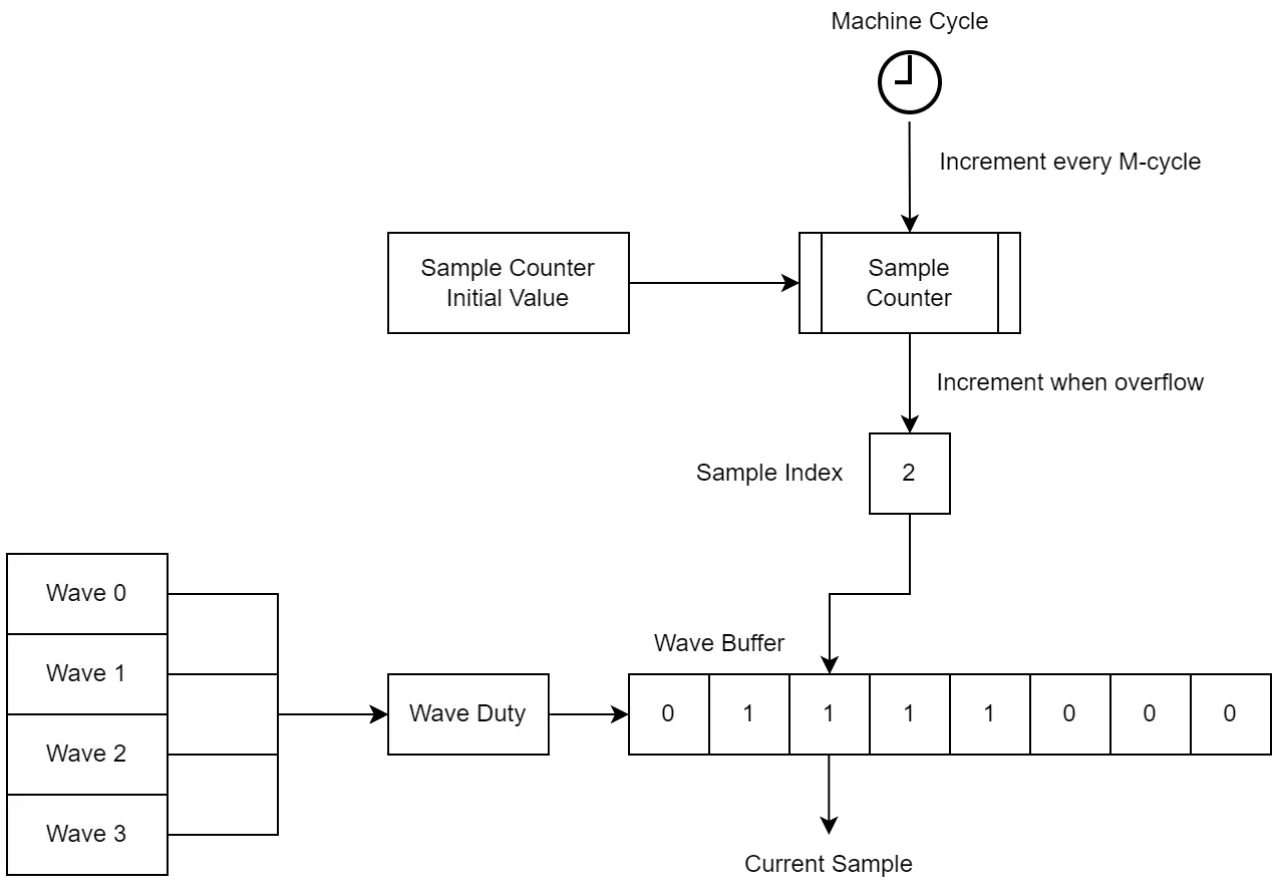
7	6	3~5	0~2
触发通道	开启length timer	-	sample counter初始值（高3位）

由于寄存器较多，我们在这里并不直接介绍每个寄存器位的功能，而是将其结合到通道对应的功能中进行介绍。这些功能包括：

1. 波形信号生成。
2. sweep。
3. envelope。
4. length timer。

脉冲波形信号生成

通道1通过向NR14.7中写入值1来触发。触发（trigger）通道的操作不仅会激活（enable）通道，还会初始化通道所有内部寄存器的状态，从而让大部分对通道控制寄存器的修改生效。所有的通道都可以在激活的情况下反复触发，每一次触发都相当于重置（reset）通道所有的内部状态至使用寄存器设定的初始状态。在触发通道1以后，通道1的脉冲波形信号生成电路立即开始工作。脉冲波形信号生成的硬件原理图如下所示：



在生成脉冲信号时，APU实际上只是在以不同的速度循环播放一段长度为8个采样的音频信号，这段信号存储在APU内置的Wave Buffer中，位深度为1，即每一个采样使用一个bit表示，其值要么是0要么是1。APU提供了四种不同的波形预设可供使用，分为标记为Wave 0~Wave 3，程序可以通过设置通过NR11.6和NR11.7的值选择使用哪一种波形。四种波形的造型如下：

Value (binary)	Duty cycle	Waveform
00	12.5 %	
01	25 %	
10	50 %	
11	75 %	

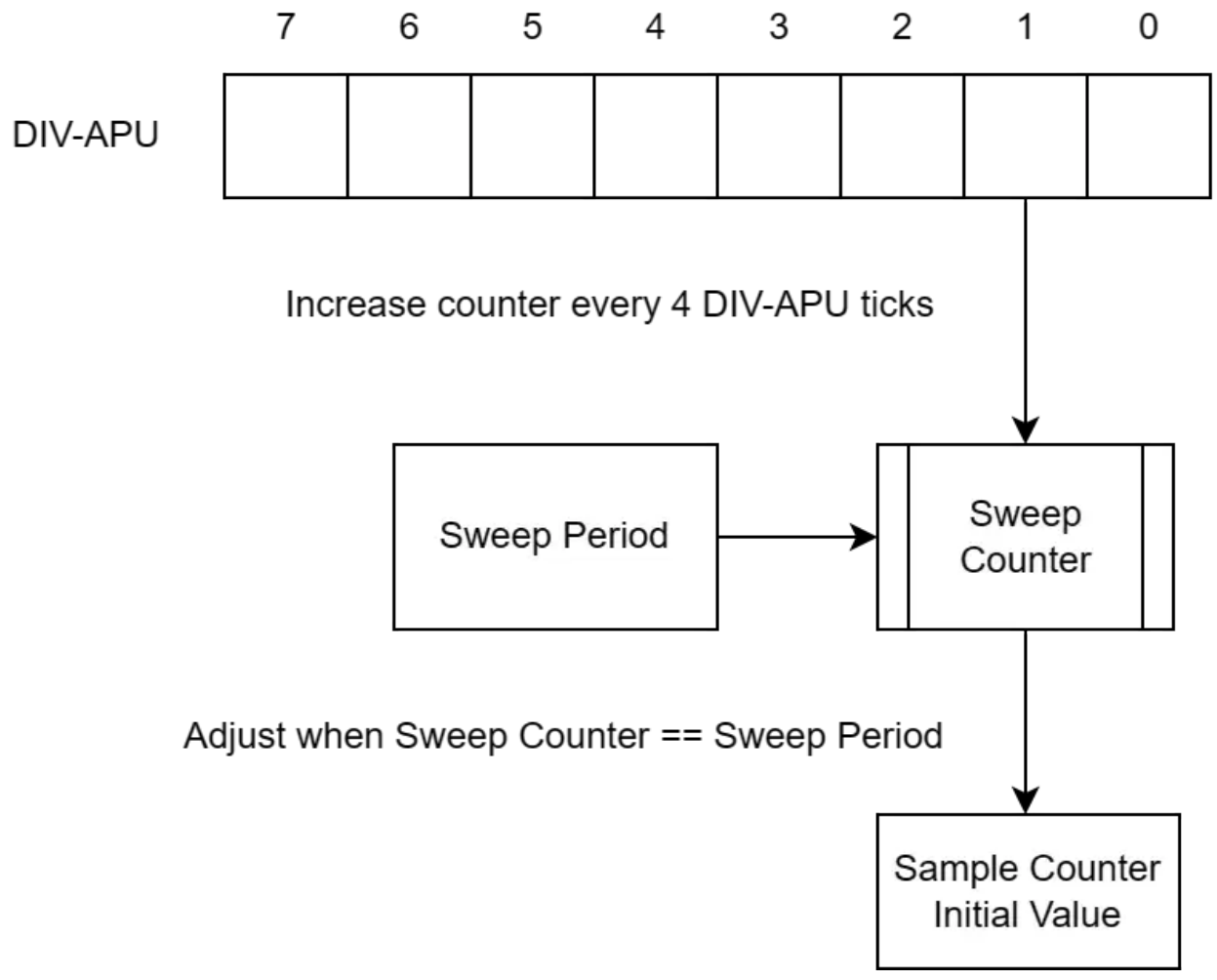
可以看到Wave 1和Wave 3的形状实际上是一样的，只是将信号进行了反转，两者在听感上并没有任何区别，因此程序实际上只能选择三种不同听感的波形。

在有了声音数据以后，我们就需要以一定的速度播放这段数据，以生成随时间变化的数字信号。APU使用一个内置的计数器（图中的sample counter）来记录每一个sample播放持续的时间。sample counter为11位寄存器，其值可以在0~2047之间变化。在通道1激活的时候，APU会读取NR13和NR14的高3位，一共11位的数据来设置sample counter的初值。在每一个**机器周期**中，APU会将sample counter的值增加1，如果sample counter的值发生了溢出，即超过了2047，则APU会将sample index指针向前移动一位，输出wave buffer中下一个sample的值，同时重置sample counter的值为初始值。如果sample index的值超过了7，则其会被重置为0，从而循环输出sample buffer中的sample。

从上图的波形形状可以看出，所有的波形都是以8个sample为一个振动周期进行循环。同时，又由于sample counter在每一个机器周期会增加1，因此每一个sample的播放时长为 $\frac{2048-Initial}{1048576}$ 秒。由此我们可以得出，音频信号的周期为 $\frac{2048-Initial}{1048576} * 8 = \frac{2048-Initial}{131072}$ 秒，频率为 $\frac{131072}{2048-Initial}$ Hz。又由于我们的Initial可以设置为0~2047之间的值，因此GameBoy最终能够输出的声音频率范围为64~131072Hz。

sweep

通道一具有被称为frequency sweep的功能。frequency sweep功能通过设置NR10.4~6为任意非0值开启，通过设置为0关闭。在frequency sweep功能开启的情况下，APU会每隔固定的时间自动增加或者减少NR13和NR14中存储的sample counter初始值，从而改变通道1输出的声音的频率。sweep功能的硬件原理图如下所示：



与生成信号所使用的硬件不同，sweep的计时并不是由时钟信号直接驱动，而是使用一个APU内置的8位DIV-APU寄存器的数值变化来驱动。DIV-APU由GameBoy Timer组件的DIV寄存器驱动，每当DIV寄存器的bit 4（第五位）的值从1变化为0时，DIV-APU的值就会加1。由于DIV寄存器本身的递增频率是16384Hz，因此DIV-APU寄存器的递增频率为512Hz。由于DIV本身是一个可以写入的寄存器（写入DIV会将DIV的数值清零），因此虽然在通常情况下DIV-APU是以固定频率更新，但是游戏程序也可以通过频繁将DIV清零来不断触发bit 4的下降沿，从而是的DIV-APU的更新频率变快。DIV-APU除了用于驱动sweep的计时外，还用于驱动下文中将会讲述的envelope和length timer的计时，我们会在讲解envelope和length timer时一并提及。

sweep的工作以迭代（iteration）为周期重复进行。APU使用一个内部的Sweep Counter计数器来存储当前迭代所消耗的时间周期。在每次迭代开始时，Sweep Counter的值会被清零。在一次sweep迭代过程中，每当DIV-APU的值变化4次时，Sweep Counter的值就会被加1。当Sweep Counter在加1后的值等于Sweep Period时，APU就会执行一次sweep操作，该操作会使用以下公式计算修改后的Sampler Counter Initial Value的值，并写入NR13和NR14中：

$$L_{t+1} = L_t \pm \frac{L_t}{2^{step}}$$

其中，$step$ 变量为每次sweep操作的步长，由NR10.0~2寄存器决定。sweep操作既可以增加Initial Value的值，也可以减少Initial Value的值。如果NR10.3的值为1，则每次sweep操作会减少Initial Value的值，否则增加Initial Value的值。

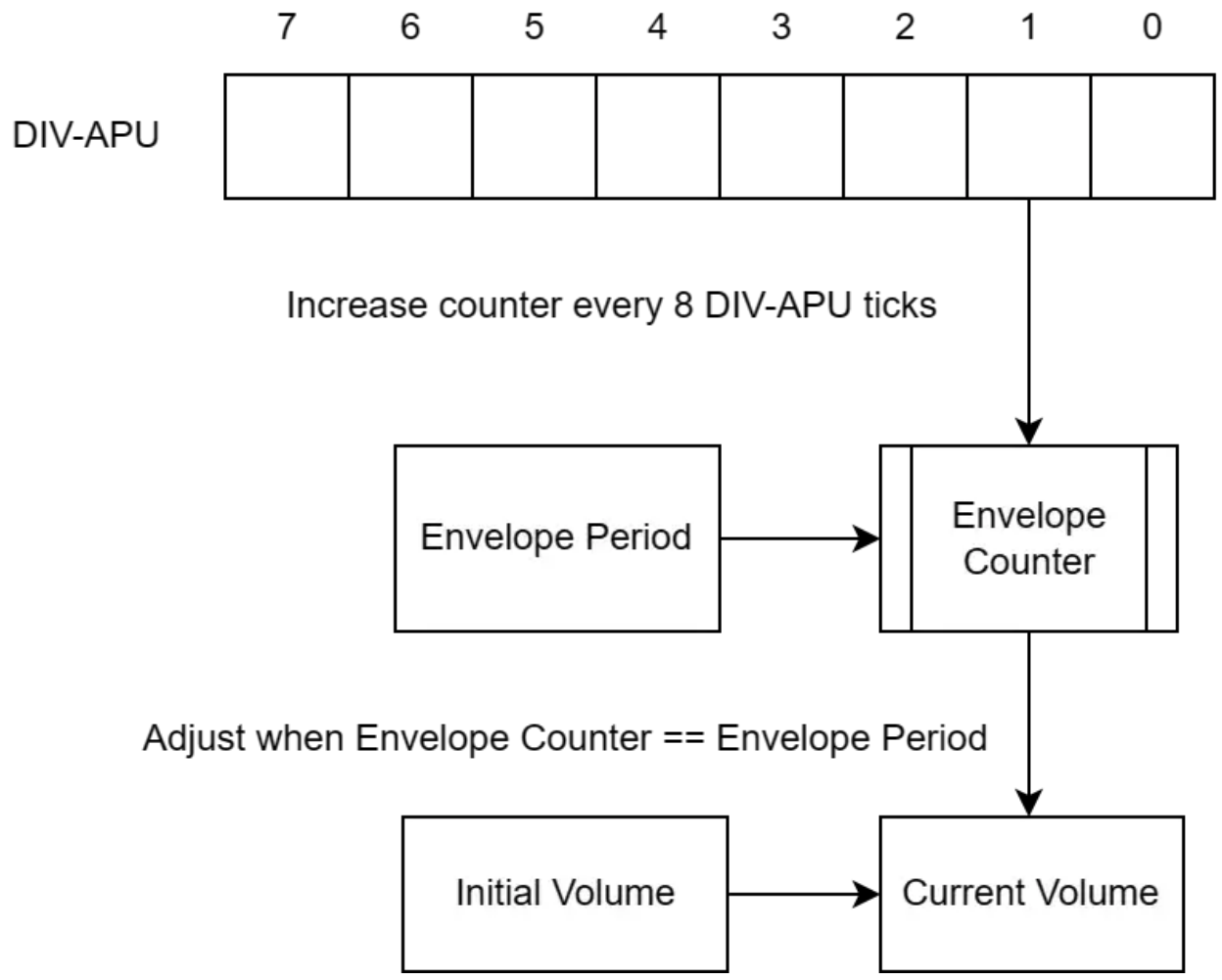
当sweep操作修改Initial Value完毕后，本次迭代结束，此时Sweep Counter的值会重新变为0，开始下一次迭代。

sweep保证计算后的Initial Value的值在[0, 2047]范围内，如果sweep在计算过程中发生了溢出（无论是上溢还是下溢），则通道会直接被禁用，且溢出的值不会被写入寄存器中。在sweep的迭代过程中，如果程序修改了Sweep Period的值，则被修改的值不会立即生效，而是会延迟到下一次sweep迭代开始时生效，唯一的例外是如果写入的值是0，则sweep功能会立即关闭，并在下一次Sweep Period的值变为非零时立即重新开始新一次的迭代。

如果在sweep迭代过程中，程序通过向NR14.7中写入值1来触发了通道1，则通道1的所有状态会被重置，其中就包括立即重置sweep并开始新一次的迭代。

envelope

通道1、2、4具有envelope功能，通道1的envelope功能可以通过设置NR12.0~2为任意非0值开启，通过设置为0关闭。envelope功能与sweep功能类似，只不过其是定期修改通道的输出音量，而sweep功能是定期修改通道的输出音频频率。envelope的硬件原理图如下所示：



可以看到envelope的硬件原理与sweep高度相似，其也是基于计时器和迭代运行的。APU使用Envelope Counter寄存器来记录每一次envelope迭代运行的时间，当一次迭代开始时，Envelope Counter的值会重置为0。在一次envelope迭代过程中，每当DIV-APU的值变化**8**次时，Envelope Counter的值就会被加1。当Envelope Counter在加1后的值等于NR12.0~2 Envelope Period时，APU就会执行一次envelope操作，并根据NR12.3的值将音量增大或者减少一个单位：如果NR12.3的值为1，则每一次envelope迭代会增加1个音量单位，否则就减小一个音量单位。在调整音量后，Envelope Counter的值会重置为0，并进入下一次迭代。

APU使用内置的寄存器Current Volume来保存当前的音量，当通道1被触发（向NR14.7中写入值1）时，APU会使用NR12.4~7 Initial Volume寄存器的值来初始化Current Volume寄存器。在之后的envelope操作中，所有的操作都只会修改内部的Current Volume寄存器，而不会将结果写回到NR12.4~7 Initial Volume寄存器中。envelope操作保证操作后的音量在[0, 15]范围中，如果envelope操作会导致音量值溢出，则该操作会被忽略。

在通道1处于启用状态时，修改Envelope Period和Initial Volume均不会产生效果，程序需要在修改后重新触发（向NR14.7中写入值1）一次通道1来重置通道1的状态，从而让Envelope Period和Initial Volume的修改生效。在重置通道1状态时，envelope会立即重新开始一次新的迭代。

length timer

所有4个通道都支持length timer功能，该功能允许程序设定一个倒计时时间，在时间到了的情况下自动关闭（禁用）对应的通道，从而实现播放某一个音频特定时长的功能。length timer的硬件原理图如下所示：

与sweep和envelope一样，length timer也使用DIV-APU寄存器驱动来进行计时。APU使用Length Timer寄存器来记录每一个通道已经运行的时间。以通道1为例，当通道1被触发时，APU会读取NR11.0~5寄存器中的值（范围为0~63）来初始化Length Timer。一旦Length Timer被初始化后，其值就不能再被外部修改，除非再次触发通道1来刷新length timer状态。length timer功能可以在通道处于启用状态时动态开关，当NR14.6的值为1时，length timer功能就处于启用状态，否则就处于禁用状态。启用和禁用length timer的操作不会修改任何内部寄存器的值。在length timer处于启用状态时，每当DIV-APU的值变化2次，APU就会将Length Timer寄存器的值加1。如果在加1操作后，Length Timer寄存器的值达到了64，则APU就会禁用通道1。

DAC

每一个通道的信号末端都有一个DAC电路，用于将通道输出的数字信号转换成模拟信号，并输出给混音器使用。DAC接受的数字信号范围为0x00~0x0F，即Sample（0~1）与Current Volume（0~15）相乘后的结果，并输出从-1~1的连续模拟信号。当输入信号为0x00时，DAC输出正1电平；当输入信号为0x0F时，DAC输出负1电平。

通道1的DAC的开关通过NR12寄存器控制，当NR12.3~7均为0时，DAC会被关闭，否则DAC将会被打开并持续工作。DAC和音频通道本身的开闭与否独立控制，当DAC被关闭（设置NR12.3~7为0）时，其会同步关闭通道，但是当DAC开启时，通道不会被同步开启，而是需要向NR14.7中写入值1触发通道开开启。同样，通道也可以在DAC工作的情况下通过sweep或者length timer功能被独立关闭。

当DAC或音频通道被关闭时，DAC会持续输出其最后输出的信号，且这个信号并不一定是0，因此会产生被称为audio pop的现象，即音频信号的中点并不在0电平上。为了避免该问题，游戏程序不应通过禁用DAC的方式关闭音频通道，而是应当先对NR12寄存器写入0x08，从而将初始音量设置为0，但不关闭DAC，再通过向NR14.7中写入值1触发通道，从而应用新写入NR12中的

值。

为了解决audio pop带来的持续性问题，GameBoy的混音器中内置一个高通滤波器（high pass filter），可以用于移除音频信号的常量位移。我们将会在之后的章节中实现该功能。

混音器

混音器（Mixer）负责混合从四个音频通道传入的音频模拟信号，并对混合信号进行音量控制，分通道以后输出至GameBoy内置扬声器或耳机接口。混音器的参数通过NR50~NR52控制。

NR50（0xFF24）寄存器的含义如下表所示：

7	4~6	3	0~2
VIN左声道	左声道音量	VIN右声道	右声道音量

- VIN左声道：VIN通道信号是否输出给左声道。1：输出，0：不输出。由于我们并不实现VIN通道，因此该值不适用。
- 左声道音量：左声道的音量总控，范围为0~7，对应从静音至最大输出音量。
- VIN右声道：VIN通道信号是否输出给右声道。1：输出，0：不输出。由于我们并不实现VIN通道，因此该值不适用。
- 右声道音量：右声道的音量总控，范围为0~7，对应从静音至最大输出音量。

NR51（0xFF25）寄存器的含义如下表所示：

7	6	5	4	3	2	1	0
CH4左声道	CH3左声道	CH2左声道	CH1左声道	CH4右声道	CH3右声道	CH2右声道	CH1右声道

- CH4左声道：通道4的信号是否输出给左声道。1：输出，0：不输出。
- CH3左声道：通道3的信号是否输出给左声道。1：输出，0：不输出。
- CH2左声道：通道2的信号是否输出给左声道。1：输出，0：不输出。
- CH1左声道：通道1的信号是否输出给左声道。1：输出，0：不输出。
- CH4右声道：通道4的信号是否输出给右声道。1：输出，0：不输出。
- CH3右声道：通道3的信号是否输出给右声道。1：输出，0：不输出。
- CH2右声道：通道2的信号是否输出给右声道。1：输出，0：不输出。
- CH1右声道：通道1的信号是否输出给右声道。1：输出，0：不输出。

NR52（0xFF26）寄存器的含义如下表所示：

7	4~6	3	2	1	0
启用APU	-	通道4标志位	通道3标志位	通道2标志位	通道1标志位

- 启用APU：该标志位控制是否启用APU。1：启用，0：禁用。禁用APU会使得APU组件完全断电，所有的功能全部停止运行，其大概会为GameBoy节省16%的电量消耗。禁用APU同时会将所有寄存器清零，并使得除了NR52以及NRx1之外的所有寄存器变为只读。
- 通道1~4标志位：这四个标志位用于标记对应的通道是否处于激活状态。所有四个标志位都为只读，APU会根据当前通道的实际激活状态更新标志位的值，而程序只能读取值，无法修改值。如果程序想要启用某一个通道，需要向NRx4.7写入1来触发通道。

实现APU

在大致了解了APU的各个组件后，就让我们开始一步步实现APU吧！首先我们需要新建两个文件：APU.hpp和APU.cpp，用于保存我们的APU部分代码。APU.hpp的代码如下：

```
#pragma once
#include <Luna/Runtime/MemoryUtils.hpp>
using namespace Luna;

struct Emulator;
struct APU
{
    // Registers.

    // CH1 registers.

    //! 0xFF10
    u8 nr10_ch1_sweep;
    //! 0xFF11
    u8 nr11_ch1_length_timer_duty_cycle;
    //! 0xFF12
    u8 nr12_ch1_volume_envelope;
    //! 0xFF13
    u8 nr13_ch1_period_low;
```

```
    //! 0xFF14
    u8 nr14_ch1_period_high_control;

    // Master control registers.

    //! 0xFF24
    u8 nr50_master_volume_vin_panning;
    //! 0xFF25
    u8 nr51_master_panning;
    //! 0xFF26
    u8 nr52_master_control;

    // Master control states.

    //! Whether APU is enabled.
    bool is_enabled() const { return bit_test(&nr52_master_control, 7); }
    //! Disables APU.
    void disable();

    void init();
    void tick(Emulator* emu);
    u8 bus_read(u16 addr);
    void bus_write(u16 addr, u8 data);
};
```

与GameBoy的大部分其它组件一样，APU主要通过init、tick、bus_read和bus_write四个函数接入模拟器中。我们同时定义了APU通道1的寄存器以及总控寄存器，这些寄存器会用于存储APU的控制参数。is_enabled会检测NR52寄存器的bit 7，用于判断当前是否开启了APU，而disable会在NR52.7被写入0时调用，以将APU的所有寄存器状态清零。APU.cpp的代码如下：

```
#include "APU.hpp"
#include "Emulator.hpp"
#include <Luna/Runtime/Log.hpp>

void APU::disable()
{
    // Clear all registers.
    memzero(this);
}
void APU::init()
{
    memzero(this);
}
void APU::tick(Emulator* emu)
{
    if(!is_enabled()) return;
    // TODO...
}
u8 APU::bus_read(u16 addr)
{
    // CH1 registers.
    if(addr >= 0xFF10 && addr <= 0xFF14)
    {
        if(addr == 0xFF11)
        {
            // lower 6 bits of NR11 is write-only.
            return nr11_ch1_length_timer_duty_cycle & 0xC0;
        }
        if(addr == 0xFF14)
        {
            // only bit 6 is readable.
            return nr14_ch1_period_high_control & 0x40;
        }
        return (&nr10_ch1_sweep)[addr - 0xFF10];
    }
    // Master control registers.
    if(addr >= 0xFF24 && addr <= 0xFF26)
    {
        return (&nr50_master_volume_vin_panning)[addr - 0xFF24];
    }
    log_error("LunaGB", "Unsupported bus read address: 0x%04X", (u32)addr);
    return 0xFF;
}
void APU::bus_write(u16 addr, u8 data)
{
    // CH1 registers.
    if(addr >= 0xFF10 && addr <= 0xFF14)
    {
```

```
        if(!is_enabled())
        {
            // Only NRx1 is writable.
            if(addr == 0xFF11)
            {
                nr11_ch1_length_timer_duty_cycle = data;
            }
        }
        else
        {
            if(addr == 0xFF14 && bit_test(&data, 7))
            {
                // CH1 trigger.
                enable_ch1();
                data &= 0x7F;
            }
            (&nr10_ch1_sweep)[addr - 0xFF10] = data;
        }
        return;
    }
    // Master control registers.
    if(addr >= 0xFF24 && addr <= 0xFF26)
    {
        if(addr == 0xFF26)
        {
            bool enabled_before = is_enabled();
            // Only bit 7 is writable.
            nr52_master_control = (data & 0x80) | (nr52_master_control & 0x7F);
            if(enabled_before && !is_enabled())
            {
                disable();
            }
            return;
        }
        // All registers except NR52 is read-only if APU is not enabled.
        if(!is_enabled()) return;
        (&nr50_master_volume_vin_panning)[addr - 0xFF24] = data;
        return;
    }
    log_error("LunaGB", "Unsupported bus write address: 0x%04X", (u32)addr);
}
```

上述代码的要点如下：

1. 在tick函数中，我们仅在APU被启动的时候才执行实际的APU更新代码，因此需要首先判断APU是否被启动，并在APU关闭时直接退出更新函数。
2. 在bus_read函数中，我们需要根据寄存器的读写规则，对读取的数值进行操作：NR11的低6位为只写，NR14除了bit 6以外的所有位均为只写。对于这些只写的位，我们在这里将返回值统一设置为1。
3. 在bus_write函数中，由于大部分寄存器在APU关闭时均为只读，因此我们需要首先判断当前APU的开启状态，并在状态为关闭时禁用除了NR52以及NR11之外的所有寄存器的写入操作。同时，当APU处于开启状态时，我们需要判断NR14的bit 7是否被写入1，并在写入1时调用enable_ch1函数触发通道1。同时，NR52寄存器只有bit 7是可写入的，因此我们需要使用位操作保留其余位的原始值，而只更新bit 7的值，并在bit 7从1变化为0时调用disable函数关闭APU。

接着，我们修改Emulator.hpp和Emulator.cpp，将APU接入模拟器中。Emulator类的修改如下：

```
/*...*/
#include "RTC.hpp"
#include "APU.hpp"
using namespace Luna;

/*...*/

struct Emulator
{
    /*...*/
    RTC rtc;
    APU apu;

    RV init(Path cartridge_path, const void* cartridge_data, usize cartridge_da
    /*...*/
};
```

Emulator.cpp的修改如下：

```
RV Emulator::init(Path cartridge_path, const void* cartridge_data, usize cartr
{
    /*...*/
    rtc.init();
    apu.init();
    /*...*/
}
void Emulator::tick(u32 mcycles)
{
    u32 tick_cycles = mcycles * 4;
    for(u32 i = 0; i < tick_cycles; ++i)
    {
        /*...*/
        ppu.tick(this);
        apu.tick(this);
    }
}
/*...*/
u8 Emulator::bus_read(u16 addr)
{
    /*...*/
    if(addr == 0xFF0F)
    {
        // IF
        return int_flags | 0xE0;
    }
    if(addr >= 0xFF10 && addr <= 0xFF3F)
    {
        return apu.bus_read(addr);
    }
    /*...*/
}
void Emulator::bus_write(u16 addr, u8 data)
{
    /*...*/
    if(addr == 0xFF0F)
    {
        // IF
        int_flags = data & 0x1F;
        return;
    }
    if(addr >= 0xFF10 && addr <= 0xFF3F)
    {
        apu.bus_write(addr, data);
        return;
    }
    /*...*/
}
```

在模拟器部分的代码中，我们只是简单地调用了APU的初始化、更新，以及总线读写函数，其余的代码都会在APU.hpp和APU.cpp中实现。

实现DIV-APU

首先我们先来实现DIV-APU及其更新逻辑，该内部寄存器会用于驱动APU的sweep、envelope和length timer的计时。在APU.hpp中添加下列变量和函数：

```
struct APU
{
    /*...*/
    u8 nr52_master_control;

    // APU internal state.

    // Stores the timer DIV value in last tick to detect div value change.
    u8 last_div;
    // The DIV-APU counter, increases every time DIV's bit 4 goes from 1 to 0.
    u8 div_apu;
    void tick_div_apu(Emulator* emu);

    // Master control states.
    /*...*/
};
```

我们添加了两个变量：last_div和div_apu。其中，div_apu用于记录当前DIV-APU内部寄存器的值，last_div用于记录上一次APU tick时Timer组件的DIV寄存器的值，用于检测DIV寄存器值变化的下降沿。tick_div_apu用于在每一次APU tick时更新DIV-APU寄存器，并根据当前的值调用APU对应的功能。现在，在APU.cpp中实现tick_div_apu：

```
void APU::tick_div_apu(Emulator* emu)
{
    u8 div = emu->timer.read_div();
    // When DIV bit 4 goes from 1 to 0...
    if(bit_test(&(last_div), 4) && !bit_test(&div, 4))
    {
        // 512Hz.
        ++div_apu;
        if((div_apu % 2) == 0)
        {
            // Length is ticked at 256Hz.
            tick_ch1_length();
        }
        if((div_apu % 4) == 0)
        {
            // Sweep is ticked at 128Hz.
            tick_ch1_sweep();
        }
        if((div_apu % 8) == 0)
        {
            // Envelope is ticked at 64Hz.
            tick_ch1_envelope();
        }
    }
    last_div = div;
}
```

可以看到，在每一次DIV-APU更新时，程序会读取一次Timer的DIV寄存器的值，并与之前的DIV值比较，以判断其bit 4是否从1变成了0。当DIV寄存器的bit 4出现下降沿时，其首先会增加div_apu的值，然后通过判断div_apu余2、4、8后的余数是否为0来每隔2、4、8次DIV-APU值变化时触发对应的功能。tick_ch1_length、tick_ch1_sweep和tick_ch1_envelope分别用于更新一次通道1的length timer、sweep和envelope状态，我们会在接下来实现通道1对应功能时实现这些函数。

接下来，让我们修改APU::tick函数，以调用tick_div_apu函数：

```
void APU::tick(Emulator* emu)
{
    if(!is_enabled()) return;
    // DIV-APU is ticked at 4194304 Hz.
    tick_div_apu(emu);
    // TODO...
}
```

如此便完成了DIV-APU相关部分的内容。

实现通道1的脉冲波形生成

接下来，我们开始实现通道1的脉冲波形生成组件。首先我们在APU类中添加相关的变量和辅助函数：

```
struct APU
{
    /*...*/
    //! Disables APU.
    void disable();
    //! Whether CH1 is enabled.
    bool ch1_enabled() const { return bit_test(&nr52_master_control, 0); }

    // CH1 states.
    // Audio generation states.
    u8 ch1_sample_index;
    u8 ch1_volume;
    u16 ch1_period_counter;
    f32 ch1_output_sample;

    //! Whether CH1 DAC is powered on.
    bool ch1_dac_on() const { return (nr12_ch1_volume_envelope & 0xF8) != 0; }
    void enable_ch1();
    void disable_ch1();
```

```
    u8 ch1_wave_type() const { return (nr11_ch1_length_timer_duty_cycle & 0xC0)
    u8 ch1_initial_volume() const { return (nr12_ch1_volume_envelope & 0xF0) >>
    u16 ch1_period() const { return (u16)nr13_ch1_period_low + (((u16)(nr14_ch1
    void tick_ch1(Emulator* emu);

    void init();
    /*...*/
};
```

各个变量和函数的含义如下：

1. ch1_enabled用于查询当前通道1是否被激活，其通过判断NR52寄存器的bit 0来完成判断。
2. ch1_sample_index用于记录我们当前输出的sample的序号，范围为0~7。
3. ch1_volume用于记录当前通道1的音量，范围为0~F。由于之后会实现的envelope功能不会改变NR12中的initial volume值，因此我们需要一个内部变量来记录当前生效的通道1音量。
4. ch1_period_counter用于记录我们在当前sample上停留的时间，即上文中的sample counter，用于在时间溢出时切换到下一个sample进行播放。
5. ch1_output_sample存储通道1最终输出的信号电平值，范围为[-1~1]。由于我们的模拟器无法真的输出一个模拟信号，因此我们使用32位浮点数来尽可能精确地表示模拟信号的电平，并通过极高的采样率（1048576Hz）来使得输出信号与模拟信号尽可能接近。
6. ch1_dac_on用于判断当前DAC是否处于开始状态。当NR12.3~7均为0时，DAC处于关闭状态，否则处于开启状态。
7. enable_ch1和disable_ch1用于开启和关闭通道1。
8. ch1_wave_type、ch1_initial_volume和ch1_period分别读取通道1对应的寄存器的值，以返回通道1当前的波形、初始音量和sample counter的初始值。
9. 最后，tick_ch1用于更新通道1的各个寄存器的内部状态。

首先我们来实现enable_ch1和disable_ch1。enable_ch1通过向NR14.7写入1来调用，实现如下：

```
void APU::enable_ch1()
{
    bit_set(&nr52_master_control, 0);
    // Save NR1x states to registers.
    ch1_sample_index = 0;
    ch1_volume = ch1_initial_volume();
    ch1_period_counter = ch1_period();
}
```

可以看到，enable_ch1主要的功能就是读取各个寄存器中程序设定的值，然后将他们保存在内部寄存器中，并将所有的counter类寄存器的值清零，从而开始新的一轮迭代。同时，enable_ch1还会将NR52.0设置为1，以通知程序当前通道1处于启动状态。

disable_ch1通常由sweep和length timer调用，以在参数溢出或者计时到的时候自动关闭通道1。disable_ch1只需要将NR52.0设置为0即可：

```
void APU::disable_ch1()
{
    bit_reset(&nr52_master_control, 0);
}
```

tick_ch1用于生成和输出通道1的脉冲波形信号，其代码如下：

```
#include <Luna/Runtime/Math/Math.hpp>

constexpr u8 pulse_wave_0[8] = {1, 1, 1, 1, 1, 1, 1, 0};
constexpr u8 pulse_wave_1[8] = {0, 1, 1, 1, 1, 1, 1, 0};
constexpr u8 pulse_wave_2[8] = {0, 1, 1, 1, 1, 0, 0, 0};
constexpr u8 pulse_wave_3[8] = {1, 0, 0, 0, 0, 0, 0, 1};
inline f32 dac(u8 sample)
{
    return lerp(-1.0f, 1.0f, ((f32)(15 - sample)) / 15.0f);
}
void APU::tick_ch1(Emulator* emu)
{
    if(!ch1_dac_on())
    {
        disable_ch1();
        return;
    }
    ++ch1_period_counter;
    if(ch1_period_counter >= 0x800)
    {
        // advance to next sample.
        ch1_sample_index = (ch1_sample_index + 1) % 8;
```

```
        ch1_period_counter = ch1_period();
    }
    u8 sample = 0;
    switch(ch1_wave_type())
    {
        case 0: sample = pulse_wave_0[ch1_sample_index]; break;
        case 1: sample = pulse_wave_1[ch1_sample_index]; break;
        case 2: sample = pulse_wave_2[ch1_sample_index]; break;
        case 3: sample = pulse_wave_3[ch1_sample_index]; break;
        default: break;
    }
    ch1_output_sample = dac(sample * ch1_volume);
}
```

tick_ch1的流程如下：

1. 首先，我们判断当前DAC是否处于关闭状态。如果当前DAC处于关闭状态，则我们需要同时关闭通道1的波形生成硬件，并提前返回，不再执行之后的波形生成流程。
2. 接着，我们对ch1_period_counter的值增加1，如果ch1_period_counter在增加后大于等于2048（即十六进制下的0x800），则我们将ch1_sample_index的值加1，从而开始输出下一个sample的值，然后将ch1_period_counter重置为ch1_period返回的值，即sample counter的初始值。在修改ch1_sample_index时，我们将增加的结果与8求余数，以便最终的数值可以在0~7之间循环。
3. 最后，我们调用ch1_wave_type，并根据返回的波形类型序号读取不同的波形数据。我们通过长度为8的数组定义了四段可选波形的数据，每段波形都由8个sample构成，且值只有0或者1，对应上文中的低电平和高电平信号。
4. 最后，我们将波形数据与当前通道的音量相乘，得到一个范围在0~15的sample，然后调用dac函数将该sample值转换为模拟电平值。dac使用lerp（线性插值）将输入信号映射到1~-1范围，并写入ch1_output_sample，以供后续混音器电路读取使用。

最后，我们在APU::tick中加入对tick_ch1的调用：

```
void APU::tick(Emulator* emu)
{
    if(!is_enabled()) return;
    // DIV-APU is ticked at 4194304 Hz.
    tick_div_apu(emu);
    // APU is ticked at 1048576 Hz, trus has 1048576 sample rate.
    if((emu->clock_cycles % 4) == 0)
    {
        // Tick CH1.
        if(ch1_enabled())
        {
            tick_ch1(emu);
        }
    }
}
```

通道脉冲信号生成组件在每一个机器周期中更新一次，因此我们判断当前的时钟周期计数是否为4的倍数，从而在每4个时钟周期调用一次tick_ch1。如此便完成了通道1脉冲波形生成的功能。

实现通道1的sweep功能

接下来我们来实现通道1的sweep功能。在APU类中添加以下变量和函数：

```
struct APU
{
    /*...*/
    f32 ch1_output_sample;
    // Sweep states.
    u8 ch1_sweep_iteration_counter;
    u8 ch1_sweep_iteration_pace;
    /*...*/
    void disable_ch1();
    u8 ch1_sweep_pace() const { return (nr10_ch1_sweep & 0x70) >> 4; }
    bool ch1_sweep_subtraction() const { return bit_test(&nr10_ch1_sweep, 3); }
    u8 ch1_sweep_individual_step() const { return nr10_ch1_sweep & 0x07; }
    /*...*/
    u16 ch1_period() const { return (u16)nr13_ch1_period_low + (((u16)(nr14_ch1
    void set_ch1_period(u16 period)
    {
        nr13_ch1_period_low = (u8)(period & 0xFF);
        nr14_ch1_period_high_control = (nr14_ch1_period_high_control & 0xF8) +
    }
    void tick_ch1_sweep();
```

```
    void tick_ch1(Emulator* emu);
    /*...*/
};
```

各个变量和函数的含义如下：

1. ch1_sweep_iteration_counter用于记录当前迭代已经经过的时间，其从0开始增加，当该值等于ch1_sweep_iteration_pace时，我们就会执行一次sweep操作，并重置该计数器为0。
2. ch1_sweep_iteration_pace用于记录当前sweep迭代的延时。由于用户在一次sweep期间对NR10.4~6的修改会延迟到下一个sweep迭代才生效，因此我们需要在每一次sweep迭代开始时缓存当前的NR10.4~6的值，从而在一次sweep迭代期间保持该值不变。
3. ch1_sweep_pace、ch1_sweep_subtraction和ch1_sweep_individual_step函数是三个辅助函数，用于通过位操作从寄存器中提取sweep部分的参数。
4. set_ch1_period是一个辅助函数，用于在sweep功能更新sample counter初始值时将初始值写回对应的寄存器中。
5. tick_ch1_sweep在tick_div_apu中每当DIV-APU值变化4次时调用一次，以更新sweep的内部状态。

接着修改APU::enable_ch1函数，在通道1触发时初始化这些内部状态变量：

```
void APU::enable_ch1()
{
    /*...*/
    ch1_volume = ch1_initial_volume();
    ch1_sweep_iteration_counter = 0;
    ch1_sweep_iteration_pace = ch1_sweep_pace();
}
```

根据sweep的性质，在通常情况下，向NR10.4~6写入值不会立即生效，而是会延迟到下一个sweep迭代才生效，唯一的例外是如果写入的值是0，则其会立即停止当前sweep的迭代，并在下一次设置为非0值时开始新一次的迭代，因此我们需要修改APU::bus_write函数，添加对于NR10.4~6寄存器的写入判断，并在开启新一次迭代的时候重置内部寄存器的状态：

```
void APU::bus_write(u16 addr, u8 data)
{
    // CH1 registers.
    if(addr >= 0xFF10 && addr <= 0xFF14)
    {
        if(!is_enabled())
        {
            /*...*/
        }
        else
        {
            if(addr == 0xFF10)
            {
                if((nr10_ch1_sweep & 0x70) == 0 && ((data & 0x70) != 0))
                {
                    // Restart sweep iteration.
                    ch1_sweep_iteration_counter = 0;
                    ch1_sweep_iteration_pace = (data & 0x70) >> 4;
                }
            }
            if(addr == 0xFF14 && bit_test(&data, 7))
            {
                /*...*/
            }
            (&nr10_ch1_sweep)[addr - 0xFF10] = data;
            return;
        }
        /*...*/
}
```

最后就是实现sweep功能的核心函数——tick_ch1_sweep了：

```
void APU::tick_ch1_sweep()
{
    if(ch1_enabled() && ch1_sweep_pace())
    {
        ++ch1_sweep_iteration_counter;
        if(ch1_sweep_iteration_counter == ch1_sweep_iteration_pace)
        {
            // Computes period after modification.
            i32 period = (i32)ch1_period();
```

```
            u8 step = ch1_sweep_individual_step();
            if(ch1_sweep_subtraction())
            {
                period -= period / (1 << step);
            }
            else
            {
                period += period / (1 << step);
            }
            // If period is out of valid range after sweep, the channel is
            // disabled.
            if(period > 0x07FF || period <= 0)
            {
                disable_ch1();
            }
            else
            {
                // Write period back.
                set_ch1_period((u16)period);
            }
            ch1_sweep_iteration_counter = 0;
            // Reload iteration pace when one iteration completes.
            ch1_sweep_iteration_pace = ch1_sweep_pace();
        }
    }
}
```

该函数的实现要点如下：

1. tick_ch1_sweep在运行时会首先判断通道1是否处于开启状态，并且当前NR10.4~6是否为非零值，只有在两者均满足的情况下才会真正执行sweep的状态更新操作。
2. 在每一次sweep更新时，我们对ch1_sweep_iteration_counter变量的值增加1，当ch1_sweep_iteration_counter的值达到ch1_sweep_iteration_pace时，就会触发一次sweep操作，我们使用上述sweep规则计算新的sample counter inital value值，并调用set_ch1_period写入到NR13和NR14寄存器中。如果新的sample counter inital value值超过了能够表示的范围（0~2047），则我们调用disable_ch1禁用通道1。最后，我们重置ch1_sweep_iteration_counter为0，并重新读取用户设定的sweep period值，用于更新ch1_sweep_iteration_pace变量，进入下一次迭代。

如此便完成了通道1的sweep功能的实现。

实现通道1的envelope功能

接下来我们来实现通道1的envelope功能。首先我们在APU类中添加以下变量和函数：

```
struct APU
{
    /*...*/
    u8 ch1_sweep_iteration_pace;
    // Envelope states.
    bool ch1_envelope_iteration_increase;
    u8 ch1_envelope_iteration_pace;
    u8 ch1_envelope_iteration_counter;

    /*...*/
    u8 ch1_wave_type() const { return (nr11_ch1_length_timer_duty_cycle & 0xC0)
    u8 ch1_envelope_pace() const { return nr12_ch1_volume_envelope & 0x07; }
    bool ch1_envelope_increase() const { return bit_test(&nr12_ch1_volume_enve
    u8 ch1_initial_volume() const { return (nr12_ch1_volume_envelope & 0xF0) >>
    /*...*/

    void tick_ch1_sweep();
    void tick_ch1_envelope();
    void tick_ch1(Emulator* emu);

    /*...*/
};
```

各个变量和函数的含义如下：

1. ch1_envelope_iteration_increase用于记录NR12.3在通道触发时的值，以确保后续程序对寄存器的修改不会影响当前envelope的运行。
2. ch1_envelope_iteration_pace用于记录NR12.0~2在通道触发时的值，以确保后续程序对寄存器的修改不会影响当前envelope的运行。
3. ch1_envelope_iteration_counter用于记录当前迭代已经经过时间，从而在倒计时结束时执行实

行的envelope操作。
4. ch1_envelope_pace和ch1_envelope_increase用于方便地获取NR12.0~2和NR12.3寄存器的值。
5. tick_ch1_envelope在tick_div_apu中每当DIV-APU值变化8次时调用一次，以更新envelope的内部状态。

接着修改APU::enable_ch1函数，在通道1触发时初始化这些内部状态变量：

```
void APU::enable_ch1()
{
    /*...*/
    ch1_sweep_iteration_pace = ch1_sweep_pace();
    ch1_envelope_iteration_increase = ch1_envelope_increase();
    ch1_envelope_iteration_pace = ch1_envelope_pace();
    ch1_envelope_iteration_counter = 0;
}
```

由于对NR12的所有修改都需要重新触发通道1才会生效，因此我们不需要和实现sweep时一样在bus_write时对NR12寄存器写入的值进行特殊判断，只需要在enable_ch1时重设envelope状态即可。

最后让我们来实现APU::tick_ch1_envelope函数，代码如下：

```
void APU::tick_ch1_envelope()
{
    if(ch1_enabled() && ch1_envelope_iteration_pace)
    {
        ++ch1_envelope_iteration_counter;
        if(ch1_envelope_iteration_counter >= ch1_envelope_iteration_pace)
        {
            if(ch1_envelope_iteration_increase)
            {
                if(ch1_volume < 15)
                {
                    ++ch1_volume;
                }
            }
            else
            {
                if(ch1_volume > 0)
                {
                    --ch1_volume;
                }
            }
            ch1_envelope_iteration_counter = 0;
        }
    }
}
```

该函数的实现要点如下：

1. envelope只会在通道1开启，且NR12.0~2初始值非0的情况下运行，因此我们在tick_ch1_envelope首先对其进行判断，并在条件不满足时不运行接下来的操作。
2. 在envelope开启的情况下，每一次tick时该函数会将ch1_envelope_iteration_counter的值加1，一旦ch1_envelope_iteration_counter值在增加后等于NR12.0~2初始值，则会进行一次实际的envelope操作。
3. 在执行envelope操作的时候，APU会根据NR12.3的值，将当前音量调低或者调高一个单位，但是保证调整之后的值在[0, 15]范围内。当当前音量已经到达最小或者最大值时，envelope操作不再对音量进行调节。

如此便完成了通道1的envelope功能的实现。

实现通道1的length timer

接下来我们来实现通道1的length timer功能。首先我们在APU类中添加以下变量和函数：

```
struct APU
{
    /*...*/
    u8 ch1_envelope_iteration_counter;
    // Length timer states.
    u8 ch1_length_timer;
    /*...*/
    u8 ch1_sweep_individual_step() const { return nr10_ch1_sweep & 0x07; }
    u8 ch1_initial_length_timer() const { return (nr11_ch1_length_timer_duty_cy
```

```
    u8 ch1_wave_type() const { return (nr11_ch1_length_timer_duty_cycle & 0xC0)
    /*...*/
    void set_ch1_period(u16 period)
    {
        nr13_ch1_period_low = (u8)(period & 0xFF);
        nr14_ch1_period_high_control = (nr14_ch1_period_high_control & 0xF8) +
    }
    bool ch1_length_enabled() const { return bit_test(&nr14_ch1_period_high_cor

    void tick_ch1_sweep();
    void tick_ch1_envelope();
    void tick_ch1_length();
    void tick_ch1(Emulator* emu);
    /*...*/
};
```

各个变量和函数的含义如下：

1. ch1_length_timer用于记录当前通道的关闭倒计时，该值从NR11.0~5开始增加，当该值达到64时，APU就会关闭通道1。
2. ch1_initial_length_timer和ch1_length_enabled用于方便地获取NR11.0~5和NR14.6寄存器的值。
3. tick_ch1_length在tick_div_apu中每当DIV-APU值变化2次时调用一次，以更新length timer的内部状态。

接着修改APU::enable_ch1函数，在通道1触发时初始化这些内部状态变量：

```
void APU::enable_ch1()
{
    /*...*/
    ch1_envelope_iteration_counter = 0;
    ch1_length_timer = ch1_initial_length_timer();
}
```

length timer的初始值在通道被触发时设置，在通道开启期间修改NR11.0~5的值并不会影响正在进行的length timer的状态，因此我们只需要在通道被触发时设置ch1_length_timer的值就行。

最后让我们来实现APU::tick_ch1_length函数，代码如下：

```
void APU::tick_ch1_length()
{
    if(ch1_enabled() && ch1_length_enabled())
    {
        ++ch1_length_timer;
        if(ch1_length_timer >= 64)
        {
            disable_ch1();
        }
    }
}
```

该函数较为简单，其在通道1激活，且length timer功能激活时将ch1_length_timer的值增加1，当ch1_length_timer在增加1之后的值达到64时，便禁用通道1。如此便完成了通道1的length timer功能的实现。

实现mixer

在实现通道1所有的功能以后，我们就需要将每一次APU::tick产生的通道1的音频sample收集起来，保存在一个连续数组中，以构成一段连续音频数据。根据APU::tick_ch1函数可知，通道1在每一次tick时输出的sample存储在ch1_output_sample变量中，因此我们需要在每一次APU::tick时从ch1_output_sample中读取声波，然后存储在一段音频缓冲中，供上一章初始化好的音频API读取并播放。

首先我们需要定义用于存储音频数据的缓冲数组变量。在实现PPU的时候，我们曾经使用过LunaSDK提供的RingDeque容器来实现像素的FIFO队列，在这里我们也可以使用同一个容器来定义用于存储音频数据的FIFO队列。在App类中定义以下变量：

```
#include <Luna/Runtime/RingDeque.hpp>
#include <Luna/Runtime/SpinLock.hpp>
/*...*/
constexpr usize AUDIO_BUFFER_MAX_SIZE = 65536;

struct App
{
```

```
    /*...*/
    //! The audio device.
    Ref<AHI::IDevice> audio_device;
    // Stores samples generated by APU.
    RingDeque<f32> audio_buffer_l;
    RingDeque<f32> audio_buffer_r;
    SpinLock audio_buffer_lock;

    /*...*/
};
```

我们为左声道和右声道分别定义了音频数组，从而让模拟器可以向两个声道单独独立音频数据。SpinLock是LunaSDK提供的轻量级自旋锁，由于我们的音频数据是在主线程中写入，并在音频线程中读取，因此我们需要使用互斥锁来同步对audio_buffer_l和audio_buffer_r变量的访问。同时，我们对音频数组中最大能够容纳的音频数据量进行限制，为65536个sample，即十六分之一秒的音频数据。接着，我们在App::init_audio_resources中为音频缓存预先分配好内存空间：

```
RV App::init_audio_resources()
{
    lutry
    {
        /*...*/
        luset(audio_device, AHI::new_device(desc));
        audio_buffer_l.reserve(AUDIO_BUFFER_MAX_SIZE);
        audio_buffer_r.reserve(AUDIO_BUFFER_MAX_SIZE);
        // Add playback data callback.
        audio_device->add_playback_data_callback(on_playback_audio);
    }
    lucatchret;
    return ok;
}
```

对音频缓存的最大尺寸做限制的目的是防止当音频API由于某些原因没有及时将音频数据取出时，不断加入队列的音频数据导致音频缓存占用过多的内存，并造成明显的声音延迟（因为音频缓存必须要按顺序播放音频，因此如果音频播放的速度跟不上生成的速度，则等待播放的音频数据会越来越多）。在音频缓存有最大限制的情况下，当模拟器在输出音频信号时，如果发现音频缓存的数据已满，就会用最新的数据覆盖最早的数据，从而保证最新的数据能够一直写入缓存，并不造成内存消耗量的持续上升。

在有了音频缓存以后，我们就可以读取通道1的音频数据，并将其压入缓存了。根据APU的硬件图可知，四个通道的数据在从DAC输出后，其需要首先进入mixer（混音器）进行声道分配和音量控制，而mixer使用NR5x系列寄存器进行参数控制。我们首先修改APU类，加入以下函数：

```
struct APU
{
    /*...*/
    // Master control states.

    //! Whether APU is enabled.
    bool is_enabled() const { return bit_test(&nr52_master_control, 7); }
    //! Disables APU.
    void disable();
    //! Whether CH1 is enabled.
    bool ch1_enabled() const { return bit_test(&nr52_master_control, 0); }
    //! Whether CH1 is outputted to right channel.
    bool ch1_r_enabled() const { return bit_test(&nr51_master_panning, 0); }
    //! Whether CH1 is outputted to left channel.
    bool ch1_l_enabled() const { return bit_test(&nr51_master_panning, 4); }
    //! Right channel master volume (0~15).
    u8 right_volume() const { return nr50_master_volume_vin_panning & 0x07; }
    //! Left channel master volume (0~15).
    u8 left_volume() const { return (nr50_master_volume_vin_panning & 0x70) >>

    // CH1 states.
    /*...*/
};
```

可以看到，新加入的ch1_r_enabled、ch1_l_enabled、right_volume和left_volume函数仅仅是将NR5x寄存器中对应的设置值从寄存器中提取出来，供我们在实现mixer时使用。在有了这些函数以后，我们便可以在APU::tick中进行mixer的混音操作，并将混音后的音频数据写入音频缓存了：

```
#include "App.hpp"
```

```
void APU::tick(Emulator* emu)
{
    if(!is_enabled()) return;
    // DIV-APU is ticked at 4194304 Hz.
    tick_div_apu(emu);
    // APU is ticked at 1048576 Hz, trus has 1048576 sample rate.
    if((emu->clock_cycles % 4) == 0)
    {
        // Tick CH1.
        if(ch1_enabled())
        {
            tick_ch1(emu);
        }
        // Mixer.
        // Output volume range in [-4, 4].
        f32 sample_l = 0.0f;
        f32 sample_r = 0.0f;
        if(ch1_dac_on() && ch1_l_enabled()) sample_l += ch1_output_sample;
        if(ch1_dac_on() && ch1_r_enabled()) sample_r += ch1_output_sample;
        // Volume control.
        // Scale output volume to [-1, 1].
        sample_l /= 4.0f;
        sample_r /= 4.0f;
        sample_l *= ((f32)left_volume()) / 7.0f;
        sample_r *= ((f32)right_volume()) / 7.0f;
        // Output samples.
        LockGuard guard(g_app->audio_buffer_lock);
        // Restrict audio buffer size to store at most 65536 samples
        // (about 1/16 second of audio data).
        if(g_app->audio_buffer_l.size() >= AUDIO_BUFFER_MAX_SIZE) g_app->audio_
        if(g_app->audio_buffer_r.size() >= AUDIO_BUFFER_MAX_SIZE) g_app->audio_
        g_app->audio_buffer_l.push_back(sample_l);
        g_app->audio_buffer_r.push_back(sample_r);
    }
}
```

该函数的实现要点如下：

1. 我们首先定义了sample_l和sample_r，用于保存该时间点在左右两个声道输出的声波信号。这两个信号从0开始，然后根据通道1的DAC是否启用，以及通道1的左右声道输出是否启动，选择性混合通道1的数据。
2. 由于我们之后会使用sample_l和sample_r来混合所有四个声道的数据，每一个声道数据的输出范围都是[-1, 1]，因此最后我们会得到一个范围在[-4, 4]之间变化的数据。为了满足音频API“数据必须在[-1, 1]之间”的格式要求，我们在混音后会将数据除以4，以将数据的变化范围从[-4, 4]减少为[-1, 1]。该操作也可以看成是将四个通道输出至左右声道的数据求和以后计算平均值。
3. 在归一化数据以后，我们使用left_volume和right_volume读取APU当前左右声道的全局音量，并与左右声道的数据相乘，以缩放对应声道的音量。由于寄存器使用0~7的数值来表示音量调节范围，因此我们需要在应用音量之前将其除以7，以缩放到0~1范围内，再将音量数值与信号相乘。
4. 最后，我们需要将音频缓冲加锁，然后将生成的音频数据写入音频缓冲中。在写入数据之前，我们需要首先判断音频缓冲中的数据尺寸是否已经达到了最大值，并在音频数据满的时候丢弃最早写入的音频数据，以保证新的音频数据写入以后不至于超出音频数据尺寸的最大值。

输出音频

最后，我们需要改写上一章中实现的on_playback_audio音频回调函数，以读取我们模拟器写入音频缓存的数据，并输出给实际的系统音频接口播放。由于我们的系统音频设备采样率与模拟器的输出采样率并不一致，模拟器的输出采样率为1048576Hz，而音频设备的输出采样率一般为44100Hz或者48000Hz，因此我们需要进行音频数据的重采样（resample），以将模拟器的输出音频数据转换为音频设备需要的音频数据。on_playback_audio的实现如下：

```
u32 on_playback_audio(void* dst_buffer, const AHI::WaveFormat& format, u32 num_
{
    LockGuard guard(g_app->audio_buffer_lock);
    u32 num_frames_read = 0;
    while(num_frames_read < num_frames)
    {
        f64 timestamp = (f64)num_frames_read / (f64)format.sample_rate;
        f64 sample_index = timestamp * 1048576.0;
        // Perform linear interpolation between two sample values if the sample
        u32 sample_1_index = (u32)floor(sample_index);
        u32 sample_2_index = (u32)ceil(sample_index);
        if(sample_2_index >= g_app->audio_buffer_l.size()) break;
        if(sample_2_index >= g_app->audio_buffer_r.size()) break;
```

```
        f32 sample_1_l = g_app->audio_buffer_l[sample_1_index];
        f32 sample_2_l = g_app->audio_buffer_l[sample_2_index];
        f32 sample_1_r = g_app->audio_buffer_r[sample_1_index];
        f32 sample_2_r = g_app->audio_buffer_r[sample_2_index];
        ((f32*)dst_buffer)[num_frames_read * 2] = lerp(sample_1_l, sample_2_l,
        ((f32*)dst_buffer)[num_frames_read * 2 + 1] = lerp(sample_1_r, sample_2
        ++num_frames_read;
    }
    if(num_frames_read)
    {
        // Remove read audio samples from buffer.
        f64 delta_time = (f64)num_frames_read / (f64)format.sample_rate;
        usize num_samples = (usize)(delta_time * 1048576.0);
        num_samples = min(num_samples, g_app->audio_buffer_l.size());
        g_app->audio_buffer_l.erase(g_app->audio_buffer_l.begin(), g_app->audio
        g_app->audio_buffer_r.erase(g_app->audio_buffer_r.begin(), g_app->audio
    }
    return num_frames_read;
}
```

on_playback_audio的实现可以分为两个步骤：重采样模拟器的音频数据并写入音频设备的音频缓冲区，以及从模拟器的音频缓冲队列中移除已处理的音频数据。为了执行音频重采样，我们根据当前输出的音频sample的索引以及音频设备采样率，计算出每个音频采样在音频时间线上的采样时间点（timestamp），然后将时间点乘以模拟器的采样率（1048576），就可以得出对应的采样在模拟器输出的音频数据中的索引。由于模拟器音频数据和声音设备音频数据的采样率并不一致，经过上述过程计算出的索引有极大可能并不是一个整数，而是指向两个音频采样之间的一个数据点。在这种情况下，我们会分别读取该索引点前后的两个音频采样数据，并对其进行线性插值（linear interpolation，lerp）来混合这两个音频采样，以近似两个音频采样中间时刻的音频采样数据。

在读取了音频数据以后，我们就需要将对应的数据从模拟器的音频缓冲中移除。我们使用num_frames_read来记录当前函数总共输出的音频帧数（每一帧包括一左一右两个采样数据），该值初始化为0，并在每次我们写入一帧数据时就加1，直到达到音频驱动要求的帧数（num_frames）。接着，我们将num_frames_read与音频驱动的采样率相除，求得输出的音频片段的总时长，再乘以模拟器的输出采样率，以确定需要移除的采样数量，然后调用RingDeque::erase函数从音频缓存的头部移除对应的数据。

在完成了以上操作以后，我们可以编译并运行模拟器，此时在加载卡带后，我们应当已经能够听到通道1的输出音频了。

实现通道2的音频输出

在实现通道1的输出以后，我们就可以依样画葫芦地实现通道2的音频输出。通道2的音频输出与通道1在寄存器布局和功能上基本一致，唯一的不同是通道2没有sweep功能，也因此没有NR20寄存器，而NR21~NR24的寄存器功能与NR11~NR14完全一致，因此不再赘述，我们直接给出通道2的代码实现。APU.hpp修改如下：

```
struct APU
{
    // Registers.

    // CH1 registers.

    /*...*/

    // CH2 registers.

    //! 0xFF16
    u8 nr21_ch2_length_timer_duty_cycle;
    //! 0xFF17
    u8 nr22_ch2_volume_envelope;
    //! 0xFF18
    u8 nr23_ch2_period_low;
    //! 0xFF19
    u8 nr24_ch2_period_high_control;

    // Master control registers.

    /*...*/
    //! Whether CH1 is enabled.
    bool ch1_enabled() const { return bit_test(&nr52_master_control, 0); }
    //! Whether CH2 is enabled.
    bool ch2_enabled() const { return bit_test(&nr52_master_control, 1); }
    //! Whether CH1 is outputted to right channel.
    bool ch1_r_enabled() const { return bit_test(&nr51_master_panning, 0); }
```

```
    //! Whether CH2 is outputted to right channel.
    bool ch2_r_enabled() const { return bit_test(&nr51_master_panning, 1); }
    //! Whether CH1 is outputted to left channel.
    bool ch1_l_enabled() const { return bit_test(&nr51_master_panning, 4); }
    //! Whether CH2 is outputted to left channel.
    bool ch2_l_enabled() const { return bit_test(&nr51_master_panning, 5); }
    //! Right channel master volume (0~15).
    u8 right_volume() const { return nr50_master_volume_vin_panning & 0x07; }
    //! Left channel master volume (0~15).
    u8 left_volume() const { return (nr50_master_volume_vin_panning & 0x70) >>

    // CH1 states.
    /*...*/

    // CH2 states.
    // Audio generation states.
    u8 ch2_sample_index;
    u8 ch2_volume;
    u16 ch2_period_counter;
    f32 ch2_output_sample;
    // Envelope states.
    bool ch2_envelope_iteration_increase;
    u8 ch2_envelope_iteration_pace;
    u8 ch2_envelope_iteration_counter;
    // Length timer states.
    u8 ch2_length_timer;

    //! Whether CH2 DAC is powered on.
    bool ch2_dac_on() const { return (nr22_ch2_volume_envelope & 0xF8) != 0; }
    void enable_ch2();
    void disable_ch2();
    u8 ch2_initial_length_timer() const { return (nr21_ch2_length_timer_duty_cy
    u8 ch2_wave_type() const { return (nr21_ch2_length_timer_duty_cycle & 0xC0)
    u8 ch2_envelope_pace() const { return nr22_ch2_volume_envelope & 0x07; }
    bool ch2_envelope_increase() const { return bit_test(&nr22_ch2_volume_enve
    u8 ch2_initial_volume() const { return (nr22_ch2_volume_envelope & 0xF0) >>
    u16 ch2_period() const { return (u16)nr23_ch2_period_low + (((u16)(nr24_ch2
    bool ch2_length_enabled() const { return bit_test(&nr24_ch2_period_high_cor

    void tick_ch2_envelope();
    void tick_ch2_length();
    void tick_ch2(Emulator* emu);

    void init();
    void tick(Emulator* emu);
    u8 bus_read(u16 addr);
    void bus_write(u16 addr, u8 data);
};
```

APU.cpp修改如下：

```
#include "APU.hpp"
#include "Emulator.hpp"
#include <Luna/Runtime/Log.hpp>
#include <Luna/Runtime/Math/Math.hpp>
#include "App.hpp"
void APU::tick_div_apu(Emulator* emu)
{
    u8 div = emu->timer.read_div();
    // When DIV bit 4 goes from 1 to 0...
    if(bit_test(&(last_div), 4) && !bit_test(&div, 4))
    {
        // 512Hz.
        ++div_apu;
        if((div_apu % 2) == 0)
        {
            // Length is ticked at 256Hz.
            tick_ch1_length();
            tick_ch2_length();
        }
        if((div_apu % 4) == 0)
        {
            // Sweep is ticked at 128Hz.
            tick_ch1_sweep();
        }
        if((div_apu % 8) == 0)
        {
            // Envelope is ticked at 64Hz.
```

```
            tick_ch1_envelope();
            tick_ch2_envelope();
        }
    }
    last_div = div;
}
/*...*/
void APU::tick_ch1(Emulator* emu)
{
    /*...*/
}
void APU::enable_ch2()
{
    bit_set(&nr52_master_control, 1);
    ch2_sample_index = 0;
    ch2_volume = ch2_initial_volume();
    ch2_period_counter = 0;
    ch2_envelope_iteration_increase = ch2_envelope_increase();
    ch2_envelope_iteration_pace = ch2_envelope_pace();
    ch2_envelope_iteration_counter = 0;
    ch2_length_timer = ch2_initial_length_timer();
}
void APU::disable_ch2()
{
    bit_reset(&nr52_master_control, 1);
}
void APU::tick_ch2_envelope()
{
    if(ch2_enabled() && ch2_envelope_iteration_pace)
    {
        ++ch2_envelope_iteration_counter;
        if(ch2_envelope_iteration_counter >= ch2_envelope_iteration_pace)
        {
            if(ch2_envelope_iteration_increase)
            {
                if(ch2_volume < 15)
                {
                    ++ch2_volume;
                }
            }
            else
            {
                if(ch2_volume > 0)
                {
                    --ch2_volume;
                }
            }
            ch2_envelope_iteration_counter = 0;
        }
    }
}
void APU::tick_ch2_length()
{
    if(ch2_enabled() && ch2_length_enabled())
    {
        ++ch2_length_timer;
        if(ch2_length_timer >= 64)
        {
            disable_ch2();
        }
    }
}
void APU::tick_ch2(Emulator* emu)
{
    if(!ch2_dac_on())
    {
        disable_ch2();
        return;
    }
    ++ch2_period_counter;
    if(ch2_period_counter >= 0x800)
    {
        // advance to next sample.
        ch2_sample_index = (ch2_sample_index + 1) % 8;
        ch2_period_counter = ch2_period();
    }
    u8 sample = 0;
    switch(ch2_wave_type())
    {
```

```
        case 0: sample = pulse_wave_0[ch2_sample_index]; break;
        case 1: sample = pulse_wave_1[ch2_sample_index]; break;
        case 2: sample = pulse_wave_2[ch2_sample_index]; break;
        case 3: sample = pulse_wave_3[ch2_sample_index]; break;
        default: break;
    }
    ch2_output_sample = dac(sample * ch2_volume);
}
/*...*/
void APU::tick(Emulator* emu)
{
    if(!is_enabled()) return;
    // DIV-APU is ticked at 4194304 Hz.
    tick_div_apu(emu);
    // APU is ticked at 1048576 Hz, trus has 1048576 sample rate.
    if((emu->clock_cycles % 4) == 0)
    {
        // Tick CH1.
        if(ch1_enabled())
        {
            tick_ch1(emu);
        }
        // Tick CH2.
        if(ch2_enabled())
        {
            tick_ch2(emu);
        }
        // Mixer.
        // Output volume range in [-4, 4].
        f32 sample_l = 0.0f;
        f32 sample_r = 0.0f;
        if(ch1_dac_on() && ch1_l_enabled()) sample_l += ch1_output_sample;
        if(ch1_dac_on() && ch1_r_enabled()) sample_r += ch1_output_sample;
        if(ch2_dac_on() && ch2_l_enabled()) sample_l += ch2_output_sample;
        if(ch2_dac_on() && ch2_r_enabled()) sample_r += ch2_output_sample;
        // Volume control.
        /*...*/
    }
}
u8 APU::bus_read(u16 addr)
{
    // CH1 registers.
    if(addr >= 0xFF10 && addr <= 0xFF14)
    {
        /*...*/
    }
    // CH2 registers.
    if(addr >= 0xFF16 && addr <= 0xFF19)
    {
        if(addr == 0xFF16)
        {
            // lower 6 bits of NR21 is write-only.
            return nr21_ch2_length_timer_duty_cycle & 0xC0;
        }
        if(addr == 0xFF19)
        {
            // only bit 6 is readable.
            return nr24_ch2_period_high_control & 0x40;
        }
        return (&nr21_ch2_length_timer_duty_cycle)[addr - 0xFF16];
    }
    // Master control registers.
    /*...*/
}
void APU::bus_write(u16 addr, u8 data)
{
    // CH1 registers.
    if(addr >= 0xFF10 && addr <= 0xFF14)
    {
        /*...*/
    }
    // CH2 registers.
    if(addr >= 0xFF16 && addr <= 0xFF19)
    {
        if(!is_enabled())
        {
            // Only NRx1 is writable.
            if(addr == 0xFF16)
            {
```

```
                nr21_ch2_length_timer_duty_cycle = data;
            }
        }
        else
        {
            if(addr == 0xFF19 && bit_test(&data, 7))
            {
                // CH2 trigger.
                enable_ch2();
                data &= 0x7F;
            }
            (&nr21_ch2_length_timer_duty_cycle)[addr - 0xFF16] = data;
        }
        return;
    }
    // Master control registers.
    /*...*/
}
```

编译并运行模拟器，此时在加载卡带后，我们应当已经能够听到通道1和通道2的输出音频了。

实现APU

最后，让我们在之前编写的调试窗口中添加对APU相关内部状态的展示，以便我们更方便观察APU的运行状态吧！在DebugWindow类中添加apu_gui函数：

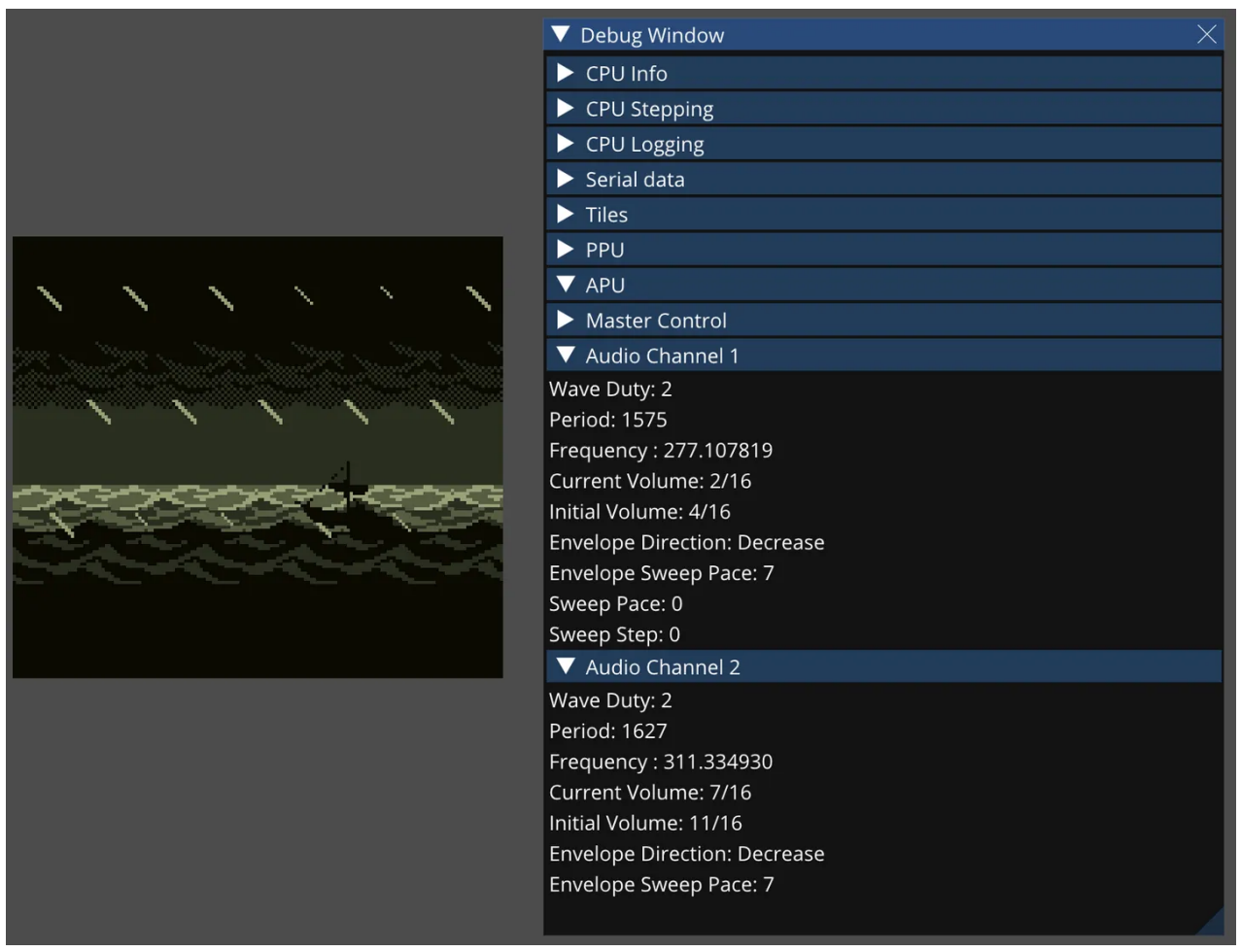
```
struct DebugWindow
{
    /*...*/
    void ppu_gui();
    void apu_gui();
};
```

然后在DebugWindow.cpp中实现并调用apu_gui函数：

```
void DebugWindow::gui()
{
    if(ImGui::Begin("Debug Window", &show))
    {
        /*...*/
        ppu_gui();
        apu_gui();
    }
    ImGui::End();
}
/*...*/
void DebugWindow::apu_gui()
{
    if (g_app->emulator)
    {
        if (ImGui::CollapsingHeader("APU"))
        {
            if(ImGui::CollapsingHeader("Master Control"))
            {
                ImGui::Text("Audio %s", g_app->emulator->apu.is_enabled() ? "En
                ImGui::Text("Channel 1 %s", g_app->emulator->apu.ch1_enabled()
                ImGui::Text("Channel 2 %s", g_app->emulator->apu.ch2_enabled()
                ImGui::Text("Left channel");
                ImGui::Text("Volume: %u/7", (u32)g_app->emulator->apu.left_volu
                ImGui::Text("Channel 1 %s", g_app->emulator->apu.ch1_l_enabled(
                ImGui::Text("Channel 2 %s", g_app->emulator->apu.ch2_l_enabled(
                ImGui::Text("Right channel");
                ImGui::Text("Volume: %u/7", (u32)g_app->emulator->apu.right_vol
                ImGui::Text("Channel 1 %s", g_app->emulator->apu.ch1_r_enabled(
                ImGui::Text("Channel 2 %s", g_app->emulator->apu.ch2_r_enabled(
            }
            if(ImGui::CollapsingHeader("Audio Channel 1"))
            {
                ImGui::Text("Wave Duty: %u", (u32)g_app->emulator->apu.ch1_wave
                ImGui::Text("Period: %u", (u32)g_app->emulator->apu.ch1_period(
                ImGui::Text("Frequency : %f", 131072.0f / (2048.0f - (f32)g_app
                ImGui::Text("Current Volume: %u/16", (u32)g_app->emulator->apu.
                ImGui::Text("Initial Volume: %u/16", (u32)g_app->emulator->apu.
                if(g_app->emulator->apu.ch1_envelope_iteration_pace != 0)
                {
                    ImGui::Text("Envelope Direction: %s", g_app->emulator->apu.
                    ImGui::Text("Envelope Sweep Pace: %u", (u32)g_app->emulator
```

```
                }
                else
                {
                    ImGui::Text("Envelope Disabled");
                }
                ImGui::Text("Sweep Pace: %u", (u32)g_app->emulator->apu.ch1_swe
                ImGui::Text("Sweep Step: %u", (u32)g_app->emulator->apu.ch1_swe
            }
            if(ImGui::CollapsingHeader("Audio Channel 2"))
            {
                ImGui::Text("Wave Duty: %u", (u32)g_app->emulator->apu.ch2_wave
                ImGui::Text("Period: %u", (u32)g_app->emulator->apu.ch2_period(
                ImGui::Text("Frequency : %f", 131072.0f / (2048.0f - (f32)g_app
                ImGui::Text("Current Volume: %u/16", (u32)g_app->emulator->apu.
                ImGui::Text("Initial Volume: %u/16", (u32)g_app->emulator->apu.
                if(g_app->emulator->apu.ch2_envelope_iteration_pace)
                {
                    ImGui::Text("Envelope Direction: %s", g_app->emulator->apu.
                    ImGui::Text("Envelope Sweep Pace: %u", (u32)g_app->emulato
                }
                else
                {
                    ImGui::Text("Envelope Disabled");
                }
            }
        }
    }
}
```

此时运行模拟器，我们就可以在调试面板中看到通道1和通道2的内部运行状态了：



以上就是本章节的全部内容了。本章节的实现细节较多，希望读者能够好好学习和消化。在下一章中，我们将继续实现APU的通道3和通道4的音频输出，从而补全APU的所有音频生成和输出功能。那么，下一章见！

还没有评论，发表第一个评论吧

文章被以下专栏收录

吉祥的游戏制作笔记
游戏制作中的技术、艺术和设计灵感记录

推荐阅读

安卓逆向之模拟器检测

甜甜

安卓模拟器运行卡顿原因七大解决方法

模拟器VT设置教程帮助很多大佬解决了在使用模拟器玩游戏时的卡顿和闪退等问题。 但是并不是所有的卡顿和闪退都能把原因丢到电脑是否开启VT这个事情上的，比如目前发现装有360安全卫士的电脑...

雷电模拟器

安卓模拟器一键宏设置教程

一.什么是一键宏 一键宏是指宏指令，主要作用是一键触发多个点击事件；游戏玩家可以用来设置一键连招，一键发言等功能；因此成为一键宏。 二.如何设置一键宏打开雷电模拟器，点击右侧栏按键...

雷电模拟器

开启电脑VT，模拟器使用体验更好！电脑开启VT详细教程

原文地址：电脑怎么开VT（虚拟化技术）VT是英文Virtualization Technology的缩写，是虚拟化技术的意思。这种技术能够令单个CPU模拟多个CPU并行，将一台电脑分成了多个独立的电脑，每台“虚...

雷电模拟器