


## 从零开始实现GameBoy模拟器 #14 波形和噪声音频通道



銀葉吉祥

浙江大学 软件工程硕士

已关注

15 人赞同了该文章

目录

收起

- 波形通道
  - 0xFF1A - NR30
  - 0xFF1B - NR31
  - 0xFF1C - NR32
  - 0xFF1D - NR33
  - 0xFF1E - NR34
- 实现波形通道
- 添加调试面板信息显示
- 噪声通道
  - 噪声信号生成
  - 0xFF20 - NR41
  - 0xFF21 - NR42
  - 0xFF22 - NR43
  - 0xFF23 - NR44
- 实现噪声通道
- 添加调试面板信息显示



欢迎来到从零开始实现GameBoy模拟器第十四章。在本章中，我们将接着上一章的内容，继续实现GameBoy APU的波形和噪声音频通道。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-14，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。

### 波形通道

GameBoy APU的通道3被称为波形通道（Wave channel）。波形通道是GameBoy中唯一可以允许用户自定义需要播放的音频的波形的通道。波形通道需要播放的数据保存在APU的波形RAM（wave RAM）中，程序可以通过地址0xFF30~0xFF3F读写波形RAM中的数据。波形RAM具有16字节的长度，每一个字节用于存放2个波形采样，因此波形RAM总共可以存放32个采样的波形数据，每一个采样具有4比特的位深度。

当通道3处于开启模式时，APU会按照地址从低到高循环播放波形数据。对于每一个字节中的两个采样，APU首先播放高4位的采样，再播放低4位的采样，因此APU的播放顺序类似：0xFF30的高4位、0xFF30的低四位、0xFF31的高4位、0xFF31的低四位，以此类推。当通道3被触发时，其会跳过波形RAM的第一个采样，从0xFF30的低四位开始往后播放，而不是高四位开始。造成该特性的原因未知。

当通道3处于启动状态时，读写波形RAM的数据会导致访问冲突，从而导致返回及写入的数据出错，因此实际的游戏只会在通道3处于关闭状态时才对波形RAM的数据进行读写操作。

除了NR5X的总控寄存器外，通道3的功能还通过以下几个寄存器控制：

#### 0xFF1A - NR30

该寄存器的含义如下表所示：

7	0~6
开启/关闭DAC	-

与通道1、通道2通过设置NR12、NR22的3~7位不同的是，通道3提供单独的控制位来开启和关闭DAC。当NR30.7为1时，通道3的DAC开启，否则DAC关闭。通道3的DAC开启和关闭的行为与通道1、通道2一致。

### 0xFF1B - NR31

该寄存器的含义如下表所示：

0~7
length timer初始值

NR31寄存器用于设置通道3的length timer的定时时间。相比通道1和通道2的length timer，通道3的length timer初始值具有8位，因此有效范围在0~255之间，一旦length timer计数器中的值大于等于256，则通道3会被关闭。通道3的length timer的其余行为与通道1、通道2完全一致。

### 0xFF1C - NR32

该寄存器的含义如下表所示：

7	5~6	0~4
-	输出音量	-

通道3没有通道1、通道2的envelope功能，而是允许用户通过向NR32写入数值直接控制通道的输出音量。输出音量使用NR32的位5~6控制，不同的值对应的含义如下表所示：

NR32.5~6值	输出音量
0	0%（静音）
1	100%（输出波形RAM的原始数据）
2	50%（将波形RAM的数据右移一位后输出）
3	25%（将波形RAM的数据右移两位后输出）

### 0xFF1D - NR33

该寄存器的含义如下表所示：

0~7
sample counter初始值（低8位）

该寄存器的含义与NR13、NR23完全一致，此处不再赘述。需要注意的是，通道3的sample counter会在一个机器周期内tick两次，因此其更新频率为2097152Hz，而非通道1和通道2的1048576Hz，因此通道3中每一个采样的播放时长为 $\frac{2048-Initial}{2097152}$  秒，正好是同参数下通道1和通道2每一个采样播放时长的一半。

### 0xFF1E - NR34

该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）

该寄存器的含义与NR14、NR24完全一致，此处不再赘述。

### 实现波形通道

在了解了波形通道的原理后，就让我们实际编程实现波形通道吧！首先我们修改APU类，添加通道3的相关变量和函数：

```
struct APU
{
    /*...*/
    u8 nr24_ch2_period_high_control;

    // CH3 registers.

    //! 0xFF1A
    u8 nr30_ch3_dac_enable;
    //! 0xFF1B
    u8 nr31_ch3_length_timer;
    //! 0xFF1C
    u8 nr32_ch3_output_level;
    //! 0xFF1D
    u8 nr33_ch3_period_low;
    //! 0xFF1E
```

```
u8 nr34_ch3_period_high_control;

// Master control registers.

/*...*/
u8 nr52_master_control;

///! 0xFF30~0xFF3F.
u8 wave_pattern_ram[16];

// APU internal state.

/*...*/
///! 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 CH3 is enabled.
bool ch3_enabled() const { return bit_test(&nr52_master_control, 2); }
///! 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 CH3 is outputted to right channel.
bool ch3_r_enabled() const { return bit_test(&nr51_master_panning, 2); }
///! 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); }
///! Whether CH3 is outputted to left channel.
bool ch3_l_enabled() const { return bit_test(&nr51_master_panning, 6); }
///! 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) >>

/*...*/
void tick_ch2(Emulator* emu);

// CH3 states.
// Audio generation states.
u8 ch3_sample_index;
u16 ch3_period_counter;
f32 ch3_output_sample;
// Length timer states.
u8 ch3_length_timer;

///! Whether CH3 DAC is powered on.
bool ch3_dac_on() const { return bit_test(&nr30_ch3_dac_enable, 7); }
void enable_ch3();
void disable_ch3();
u8 ch3_initial_length_timer() const { return nr31_ch3_length_timer; }
u8 ch3_output_level() const { return (nr32_ch3_output_level & 0x60) >> 5; }
u16 ch3_period() const { return (u16)nr33_ch3_period_low + (((u16)(nr34_ch3_period_high_control) & 0x0F) << 16); }
bool ch3_length_enabled() const { return bit_test(&nr34_ch3_period_high_control, 7); }
u8 ch3_wave_pattern(u8 index) const;

void tick_ch3_length();
void tick_ch3(Emulator* emu);

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

上述变量和函数的解释如下：

- 首先，我们添加了NR30~NR34寄存器对应的变量，以及波形RAM对应的变量（wave\_pattern\_ram）。
- 然后，我们添加了ch3\_enabled、ch3\_r\_enabled、ch3\_l\_enabled三个函数，用于检测NR51和NR52寄存器与通道3相关的标志位是否为1。
- 最后，我们添加了一些通道3的辅助函数，以及实现通道3功能所需要的内部状态变量，这些函数和变量的说明如下：
  - ch3\_sample\_index用于存储当前输出的sample的索引，范围为0~31。
  - ch3\_period\_counter用于存储当前采样输出的持续时间，其值从sampler counter初始值开始，在每一次通道3状态更新时加1，并在值达到2048时将ch3\_sample\_index增加1，并重新开始计时。
  - ch3\_output\_sample用于存储通道3当前输出的音频采样。

- 4. ch3\_length\_timer用于存储通道3的length timer的当前计时。
- 5. ch3\_dac\_on、enable\_ch3、disable\_ch3、ch3\_initial\_length\_timer、ch3\_period、ch3\_length\_enabled均与通道1和通道2对应的函数功能一致，此处不再赘述。
- 6. ch3\_output\_level是一个辅助函数，用于从NR32寄存器中提取当前输出音量的枚举值，用于在通道输出时对音量进行调整。
- 7. ch3\_wave\_pattern是一个辅助函数，用于从波形RAM从提取指定的索引对应的波形数据。
- 8. tick\_ch3\_length和tick\_ch3分别用于更新通道3的length timer和波形生成器的状态。

在完成对APU类的修改后，我们需要在APU.cpp中将通道3接入APU的各个函数中。首先是bus\_read和bus\_write函数：

```
u8 APU::bus_read(u16 addr)
{
    /*...*/
    // CH2 registers.
    if(addr >= 0xFF16 && addr <= 0xFF19)
    {
        /*...*/
    }
    // CH3 registers.
    if(addr >= 0xFF1A && addr <= 0xFF1E)
    {
        if(addr == 0xFF1B)
        {
            // NR31 is write-only.
            return 0;
        }
        if(addr == 0xFF1E)
        {
            // only bit 6 is readable.
            return nr34_ch3_period_high_control & 0x40;
        }
        return (&nr30_ch3_dac_enable)[addr - 0xFF1A];
    }
    // Master control registers.
    if(addr >= 0xFF24 && addr <= 0xFF26)
    {
        return (&nr50_master_volume_vin_panning)[addr - 0xFF24];
    }
    // Wave RAM
    if(addr >= 0xFF30 && addr <= 0xFF3F)
    {
        return wave_pattern_ram[addr - 0xFF30];
    }
    log_error("LunaGB", "Unsupported bus read address: 0x%04X", (u32)addr);
    return 0xFF;
}

void APU::bus_write(u16 addr, u8 data)
{
    /*...*/
    // CH2 registers.
    if(addr >= 0xFF16 && addr <= 0xFF19)
    {
        /*...*/
    }
    // CH3 registers.
    if(addr >= 0xFF1A && addr <= 0xFF1E)
    {
        if(!is_enabled())
        {
            // Only NRx1 is writable.
            if(addr == 0xFF1B)
            {
                nr31_ch3_length_timer = data;
            }
        }
        else
        {
            if(addr == 0xFF1E && bit_test(&data, 7))
            {
                // CH3 trigger.
                enable_ch3();
                data &= 0x7F;
            }
            (&nr30_ch3_dac_enable)[addr - 0xFF1A] = data;
        }
    }
    return;
}
```

```
// Master control registers.
if(addr >= 0xFF24 && addr <= 0xFF26)
{
    /*...*/
}
// Wave RAM
if(addr >= 0xFF30 && addr <= 0xFF3F)
{
    wave_pattern_ram[addr - 0xFF30] = data;
    return;
}
log_error("LunaGB", "Unsupported bus write address: 0x%04X", (u32)addr);
}
```

通道3中对bus\_read和bus\_write的实现与通道1、通道2基本相同，除了由于NR31并没有NR11、NR21中高2位的波形选择寄存器，因此NR31的所有寄存器都只支持写入，不能读取，因此对于NR31的读取我们直接返回0。

然后我们需要修改APU::tick\_div\_apu和APU::tick的代码，添加对tick\_ch3\_length和tick\_ch3的调用，如下所示：

```
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();
            tick_ch3_length();
        }
        /*...*/
    }
    last_div = div;
}

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 CH2.
        if(ch2_enabled())
        {
            tick_ch2(emu);
        }
        // Tick CH3.
        if(ch3_enabled())
        {
            tick_ch3(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;
        if(ch3_dac_on() && ch3_l_enabled()) sample_l += ch3_output_sample;
        if(ch3_dac_on() && ch3_r_enabled()) sample_r += ch3_output_sample;
        /*...*/
    }
}
```

接着我们需要实现上述添加的通道3的辅助函数。ch3\_wave\_pattern的实现如下：

```
u8 APU::ch3_wave_pattern(u8 index) const
{
    }
```



```
    luassert(index < 32);
    u8 base = index / 2;
    u8 wave = wave_pattern_ram[base];
    // Read upper then lower.
    wave = (index % 2) ? (wave & 0x0F) : ((wave >> 4) & 0x0F);
    return wave;
}
```

由于波形RAM中每个字节包括两个采样的数据，ch3\_wave\_pattern首先将传入的索引号除以2，以确定需要读取的字节，然后再将索引号取2的模来确定需要读取高四位的数据还是低四位的数据，并返回最终的数据。

enable\_ch3和disable\_ch3的实现如下：

```
void APU::enable_ch3()
{
    bit_set(&nr52_master_control, 2);
    ch3_period_counter = 0;
    ch3_sample_index = 1;
    ch3_length_timer = ch3_initial_length_timer();
}
void APU::disable_ch3()
{
    bit_reset(&nr52_master_control, 2);
}
```

两者的实现与通道1、通道2的类似。在enable\_ch3中，我们需要初始化通道3各个内部变量的值，需要注意的是ch3\_sample\_index的初始值为1，而不是0。

tick\_ch3\_length的实现如下：

```
void APU::tick_ch3_length()
{
    if(ch3_enabled() && ch3_length_enabled())
    {
        ++ch3_length_timer;
        if(ch3_length_timer >= 256)
        {
            disable_ch3();
        }
    }
}
```

通道3会在计数器值达到256的时候关闭通道3，而不是在64的时候关闭，其余特性和通道1及通道2保持一致。

tick\_ch3的实现如下：

```
void APU::tick_ch3(Emulator* emu)
{
    if(!ch3_dac_on())
    {
        disable_ch3();
    }
    // The CH3 of APU is ticked 2 times per M-cycle.
    for(u32 i = 0; i < 2; ++i)
    {
        ++ch3_period_counter;
        if(ch3_period_counter >= 0x800)
        {
            // advance to next sample.
            ch3_sample_index = (ch3_sample_index + 1) % 32;
            ch3_period_counter = ch3_period();
        }
    }
    u8 wave = ch3_wave_pattern(ch3_sample_index);
    u8 level = ch3_output_level();
    switch(level)
    {
        case 0: wave = 0; break;
        case 1: break;
        case 2: wave >>= 1; break;
        case 3: wave >>= 2; break;
        default: lupanic(); break;
    }
    ch3_output_sample = dac(wave);
}
```

```
    }
```

与通道1和通道2不同的是，通道3的sample counter会在每个机器周期中更新两次，因此我们在这里使用一个for循环来重复执行两次更新计数器。接着，在计算最终音频采样值时，我们会先调用ch3\_wave\_pattern获取当前索引号对应的音频采样值，然后调用ch3\_output\_level获得当前输出音量的枚举值，再根据不同的音量枚举值缩放当前音频音量后调用dac转换成浮点信号输出。

### 添加调试面板信息显示

在完成上述代码后，通道3的所有功能便实现完毕。我们可以在DebugWindow::apu\_gui函数中添加以下代码，以可视化通道3中程序写入的波形以及音频参数：

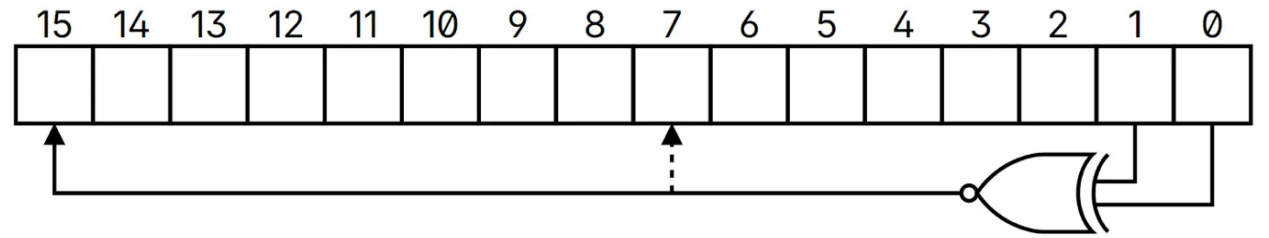
```
void DebugWindow::apu_gui()
{
    if (g_app->emulator)
    {
        if (ImGui::CollapsingHeader("APU"))
        {
            /*...*/
            if(ImGui::CollapsingHeader("Audio Channel 2"))
            {
                /*...*/
            }
            if(ImGui::CollapsingHeader("Audio Channel 3"))
            {
                f32 waveform[32];
                for(u8 i = 0; i < 32; ++i)
                {
                    waveform[i] = (f32)g_app->emulator->apu.ch3_wave_pattern(i);
                }
                ImGui::PlotLines("Waveform", waveform, 32, 0, NULL, 0.0f, 15.0f);
                ImGui::Text("Period: %u", (u32)g_app->emulator->apu.ch3_period());
                u8 output_level = g_app->emulator->apu.ch3_output_level();
                switch(output_level)
                {
                    case 0: ImGui::Text("Volume: 0%"); break;
                    case 1: ImGui::Text("Volume: 100%"); break;
                    case 2: ImGui::Text("Volume: 50%"); break;
                    case 3: ImGui::Text("Volume: 25%"); break;
                    default: break;
                }
            }
        }
    }
}
```

### 噪声通道

GameBoy APU的通道4被称为噪声通道（Noise channel）。噪声通道通过特定的硬件电路产生一个值为0或1的伪随机数，并将该伪随机数乘以通道的音量（0x00~0x0F）后作为数字信号输出。由于声音信号通过伪随机数生成，噪声通道的输出信号会在0和最大电平之间随机切换，从而产生一种类似白噪声的沙沙声，这种声音经常用于表现音乐中的鼓点、节拍等没有明显音高的声音。

### 噪声信号生成

噪声通道的输出信号通过一个被称为线性反馈位移寄存器（Linear-feedback shift register，简称LFSR）的硬件寄存器来产生，该寄存器是一个16位的寄存器，并且有两个工作模式：长模式（long mode）和短模式（short mode）。寄存器的硬件如下图所示：



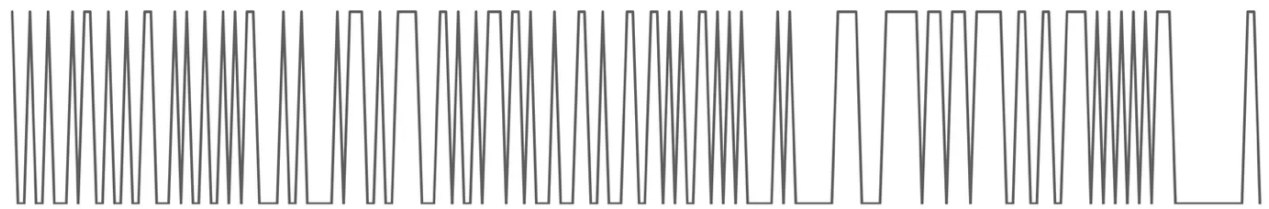
当通道4被触发时，该寄存器的所有值都会被设置为0。接下来在LFSR被更新（tick）时，LFSR都会执行以下步骤来更新寄存器中的值：

1. 将寄存器的bit 0和bit 1的值进行异或非操作（如果两个值相等则输出1，否则输出0），并将结果写入bit 15中。
2. 在短模式下，同时将上一步的结果写入bit 7中。

3. 将寄存器的值整体右移一位，然后将bit 0的值作为输出信号使用。

可以看到，在短模式下，由于bit 8~15的数据在右移后会被写入bit 7的数据覆盖，因此只有低8位参与了实际的随机数生成，这也是短模式名称的由来。

在正常工作时，通道4输出的信号为一个周期不确定的脉冲信号，如下图所示：



通道4输出信号片段

唯一的例外是当LFSR的所有位的值均为1时，无论该寄存器如何更新，其所有位的值均不会改变，因此其会持续输出信号1，导致噪声通道静音。在通常情况下，LFSR的更新不会导致其所有位均变为1，但是如果程序恰好在LFSR的低8位均为1时从长模式切换为短模式，则会触发该异常情况。该异常情况可以通过重新触发通道4来重置LFSR的值解决。

### 0xFF20 - NR41

该寄存器的含义如下表所示：

6~7	0~5
-	length timer初始值

NR41寄存器用于设置通道4的length timer的定时时间，其行为与通道1、通道2的length timer完全一致。

### 0xFF21 - NR42

NR42（0xFF21）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

NR42寄存器的参数和含义与NR12完全一致，此处不再赘述。

### 0xFF22 - NR43

该寄存器的含义如下表所示：

4~7	3	0~2
时钟除数位移（clock shift）	LFSR模式	时钟除数（clock divider）

NR43.3用于选择LFSR的工作模式，当其值为1时，LFSR工作在短模式，否则工作在长模式。

NR43.0~2及NR43.4~7用于控制LFSR的更新频率，其更新频率的计算公式为： $\frac{262144}{divider \times 2^{shift}}$ ，其中divider对应NR43.0~2的值，shift对应NR43.4~7的值。如果divider的值为0，则其会被当作0.5处理，以确保公式下方的值永远不为0。在实践中，我们可以采取和通道1、2、3类似的方法，使用一个sample counter来记录当前sample已经播放的时钟周期数，并当该值达到 $4 \times divider \times 2^{shift}$ 时更新一次LFSR的输出值。

### 0xFF23 - NR44

该寄存器的含义如下表所示：

7	6	0~5
触发通道	开启length timer	-

该寄存器的含义与NR14、NR24、NR34基本一致，除了其没有sample counter初始值的部分，因此不再赘述。

### 实现噪声通道

在了解了波形通道的原理后，就让我们实际编程实现噪声通道吧！首先我们修改APU类，添加通道4的相关变量和函数：

```
struct APU
{
    // ...
}
```



```

u8 nr34_ch3_period_high_control;

// CH4 registers.

///! 0xFF20
u8 nr41_ch4_length_timer;
///! 0xFF21
u8 nr42_ch4_volume_envelope;
///! 0xFF22
u8 nr43_ch4_freq_randomness;
///! 0xFF23
u8 nr44_ch4_control;

// Master control registers.

/*...*/
bool ch3_enabled() const { return bit_test(&nr52_master_control, 2); }
///! Whether CH4 is enabled.
bool ch4_enabled() const { return bit_test(&nr52_master_control, 3); }
/*...*/
bool ch3_r_enabled() const { return bit_test(&nr51_master_panning, 2); }
///! Whether CH4 is outputted to right channel.
bool ch4_r_enabled() const { return bit_test(&nr51_master_panning, 3); }
/*...*/
bool ch3_l_enabled() const { return bit_test(&nr51_master_panning, 6); }
///! Whether CH4 is outputted to left channel.
bool ch4_l_enabled() const { return bit_test(&nr51_master_panning, 7); }

/*...*/
void tick_ch3(Emulator* emu);

// CH4 states.
// Audio generation states.
u16 ch4_lfsr;
u8 ch4_volume;
u32 ch4_period_counter; // max value is 917504(2^15 * 7 * 4)
f32 ch4_output_sample;
// Envelope states.
bool ch4_envelope_iteration_increase;
u8 ch4_envelope_iteration_pace;
u8 ch4_envelope_iteration_counter;
// Length timer states.
u8 ch4_length_timer;

void enable_ch4();
void disable_ch4();
u8 ch4_initial_length_timer() const { return nr41_ch4_length_timer & 0x3F; }
u8 ch4_initial_volume() const { return (nr42_ch4_volume_envelope & 0xF0) >> 4; }
u32 ch4_period() const { return (u32)ch4_clock_divider() * (1 << ch4_clock_divider()); }
bool ch4_length_enabled() const { return bit_test(&nr44_ch4_control, 6); }
u8 ch4_clock_divider() const { return nr43_ch4_freq_randomness & 0x07; }
u8 ch4_lfsr_width() const { return (nr43_ch4_freq_randomness & 0x08) >> 3; }
u8 ch4_clock_shift() const { return (nr43_ch4_freq_randomness & 0xF0) >> 4; }

void update_lfsr(bool short_mode);
void tick_ch4_envelope();
void tick_ch4_length();
void tick_ch4(Emulator* emu);

/*...*/
};

```

上述变量和函数的解释如下:

1. 首先，我们添加了NR41~NR44寄存器对应的变量。
2. 然后，我们添加了ch4\_enabled、ch4\_r\_enabled、ch4\_l\_enabled三个函数，用于检测NR51和NR52寄存器与通道4相关的标志位是否为1。
3. 最后，我们添加了一些通道4的辅助函数，以及实现通道3功能所需要的内部状态变量，这些函数和变量的说明如下：
  1. ch4\_lfsr用于表示用于生成噪声信号的LFSR寄存器。
  2. ch4\_volume用于表示通道4的当前音量，该音量会随着envelope功能而动态调节。
  3. ch4\_period\_counter用于存储通道4上一次更新LFSR后经过的机器周期，以确定下一次更新LFSR的时机。由于该值最多为917504 ( $2^{15} * 7 * 4$ )，因此我们使用32位整数来存储该值。
  4. ch4\_output\_sample用于存储通道4当前输出的音频采样。
  5. ch3\_length\_timer用于存储通道3的length timer的当前计时。
  6. ch4\_envelope\_iteration\_pace、ch4\_envelope\_iteration\_pace、

- ch4\_envelope\_iteration\_counter用于存储通道4的envelope功能内部状态，其实现与通道1、通道2完全一致，此处不再赘述。
- 7. ch4\_length\_timer用于存储通道4的长度timer功能内部状态，其实现与通道1、通道2、通道3完全一致，此处不再赘述。
- 8. enable\_ch4、disable\_ch4、ch4\_initial\_length\_timer、ch4\_initial\_volume、ch4\_length\_enabled均与通道1和通道2对应的函数功能一致，此处不再赘述。
- 9. ch4\_clock\_divider和ch4\_clock\_shift用于读取NR43的时钟设置，并在ch4\_period中计算出两次LFSR更新之间的时钟周期，作为ch4\_period\_counter的比较值，来判断当前是否需要更新LFSR的状态。
- 10. ch4\_lfsr\_width用于读取NR43对于LFSR工作模式的设置。
- 11. update\_lfsr用于更新LFSR的内部状态。
- 12. tick\_ch4\_envelope、tick\_ch4\_length和tick\_ch4分别用于更新通道4的envelope、length timer和波形生成器的状态。

在完成对APU类的修改后，我们需要在APU.cpp中将通道3接入APU的各个函数中。首先是bus\_read和bus\_write函数：

```
u8 APU::bus_read(u16 addr)
{
    /*...*/
    // CH3 registers.
    if(addr >= 0xFF1A && addr <= 0xFF1E)
    {
        /*...*/
    }
    // CH4 registers.
    if(addr >= 0xFF20 && addr <= 0xFF23)
    {
        if(addr == 0xFF20)
        {
            // NR41 is write-only.
            return 0;
        }
        if(addr == 0xFF23)
        {
            // only bit 6 is readable.
            return nr44_ch4_control & 0x40;
        }
        return (&nr41_ch4_length_timer)[addr - 0xFF20];
    }
    /*...*/
}

void APU::bus_write(u16 addr, u8 data)
{
    /*...*/
    // CH3 registers.
    if(addr >= 0xFF1A && addr <= 0xFF1E)
    {
        /*...*/
    }
    // CH4 registers.
    if(addr >= 0xFF20 && addr <= 0xFF23)
    {
        if(!is_enabled())
        {
            // Only NRx1 is writable.
            if(addr == 0xFF20)
            {
                nr41_ch4_length_timer = data;
            }
        }
        else
        {
            if(addr == 0xFF23 && bit_test(&data, 7))
            {
                // CH4 trigger.
                enable_ch4();
                data &= 0x7F;
            }
            (&nr41_ch4_length_timer)[addr - 0xFF20] = data;
        }
        return;
    }
    /*...*/
}
```

通道4中对bus\_read和bus\_write的实现与通道1、通道2基本相同，除了由于NR41并没有NR11、

NR21中高2位的波形选择寄存器，因此NR41的所有寄存器都只支持写入，不能读取，因此对于NR41的读取我们直接返回0。

然后我们需要修改APU::tick\_div\_apu和APU::tick的代码，添加对tick\_ch4\_envelope、tick\_ch4\_length和tick\_ch4的调用，如下所示：

```
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();
            tick_ch3_length();
            tick_ch4_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();
            tick_ch4_envelope();
        }
    }
    last_div = div;
}

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 CH3.
        if(ch3_enabled())
        {
            tick_ch3(emu);
        }
        // Tick CH4.
        if(ch4_enabled())
        {
            tick_ch4(emu);
        }
        // Mixer.
        // Output volume range in [-4, 4].
        f32 sample_l = 0.0f;
        f32 sample_r = 0.0f;
        /*...*/
        if(ch3_r_enabled()) sample_r += ch3_output_sample;
        if(ch4_l_enabled()) sample_l += ch4_output_sample;
        if(ch4_r_enabled()) sample_r += ch4_output_sample;
        // Volume control.
        /*...*/
    }
}
```

enable\_ch4和disable\_ch4的实现如下：

```
void APU::enable_ch4()
{
    bit_set(&nr52_master_control, 3);
    ch4_period_counter = 0;
    ch4_length_timer = ch4_initial_length_timer();
    ch4_volume = ch4_initial_volume();
    ch4_envelope_iteration_increase = bit_test(&nr42_ch4_volume_envelope, 3);
}
```

```
        ch4_envelope_iteration_pace = nr42_ch4_volume_envelope & 0x07;
        ch4_envelope_iteration_counter = 0;
        ch4_lfsr = 0;
    }
    void APU::disable_ch4()
    {
        bit_reset(&nr52_master_control, 3);
    }
```

两个函数的实现与通道1、通道2的基本一致，此处不再赘述。

tick\_ch4\_envelope、tick\_ch4\_length和tick\_ch4的实现如下：

```
void APU::tick_ch4_envelope()
{
    if(ch4_enabled() && ch4_envelope_iteration_pace)
    {
        ++ch4_envelope_iteration_counter;
        if(ch4_envelope_iteration_counter >= ch4_envelope_iteration_pace)
        {
            if(ch4_envelope_iteration_increase)
            {
                if(ch4_volume < 15)
                {
                    ++ch4_volume;
                }
            }
            else
            {
                if(ch4_volume > 0)
                {
                    --ch4_volume;
                }
            }
            ch4_envelope_iteration_counter = 0;
        }
    }
}
void APU::tick_ch4_length()
{
    if(ch4_enabled() && ch4_length_enabled())
    {
        ++ch4_length_timer;
        if(ch4_length_timer >= 64)
        {
            disable_ch4();
        }
    }
}
```

两个函数的实现与通道1、通道2的基本一致，此处不再赘述。

tick\_ch4的函数实现如下：

```
void APU::tick_ch4(Emulator* emu)
{
    ++ch4_period_counter;
    if(ch4_period_counter >= ch4_period())
    {
        // advance to next sample.
        update_lfsr(!ch4_lfsr_width());
        ch4_period_counter = 0;
    }
    u8 wave = ch4_lfsr & 0x01;
    wave *= ch4_volume;
    ch4_output_sample = dac(wave);
}
```

在每一次调用tick\_ch4时，我们会将ch4\_period\_counter的值加1，并在其值大于等于ch4\_period的返回值时调用update\_lfsr，并将ch4\_period\_counter的值归零。同时，在每一次调用tick\_ch4时，我们读取ch4\_lfsr的最低位，并将其与ch4\_volume相乘，求得最终的输出音频信号，并调用dac函数转换为模拟信号后写入ch4\_output\_sample中。

update\_lfsr的实现如下：

```
void APU::update_lfsr(bool short_mode)
```

```
{
    u8 b0 = ch4_lfsr & 0x01;
    u8 b1 = (ch4_lfsr & 0x02) >> 1;
    bool v = (b0 == b1);
    ch4_lfsr = (ch4_lfsr & 0x7FFF) + (v ? 0x8000 : 0);
    if(short_mode)
    {
        ch4_lfsr = (ch4_lfsr & 0xFF7F) + (v ? 0x80 : 0);
    }
    ch4_lfsr >>= 1;
}
```

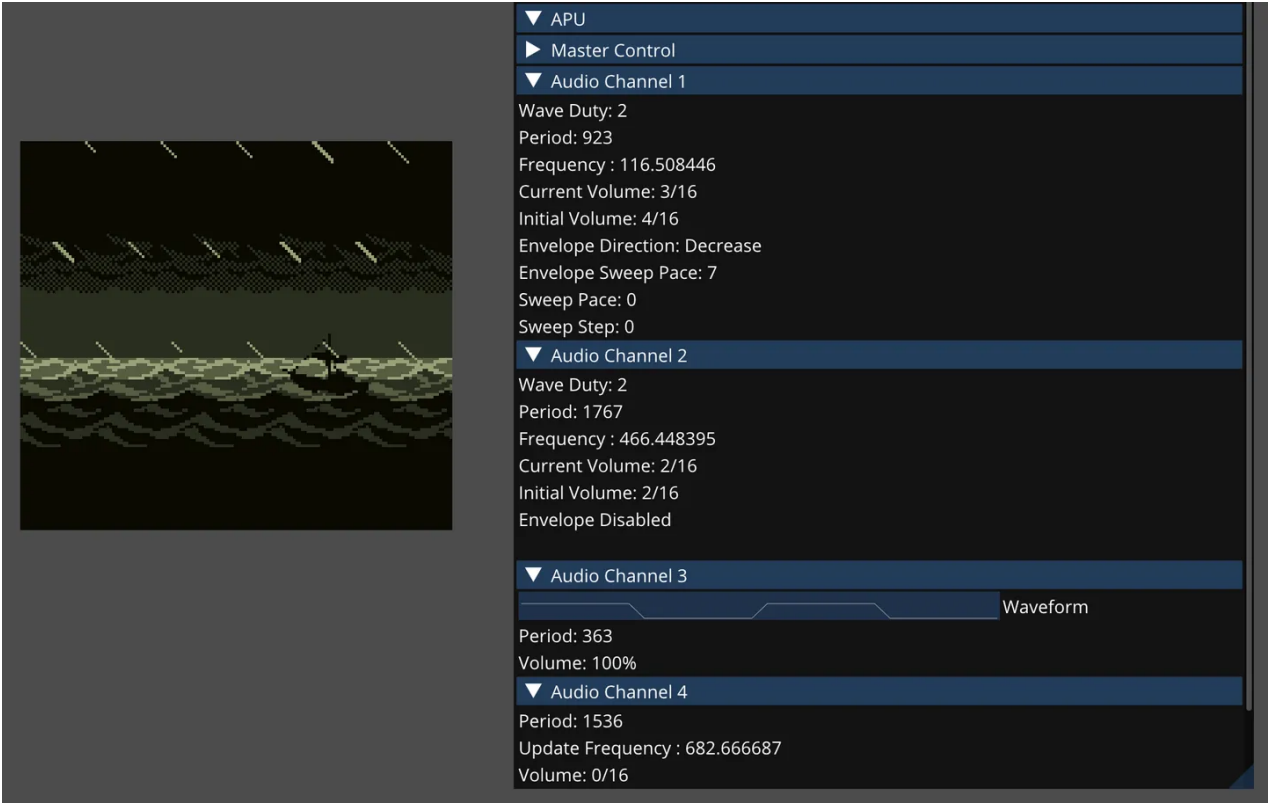
根据上文描述的算法，我们首先提取LFSR寄存器中的bit 0和bit 1的数值，判断这两个数值是否相等，并根据结果设置LFSR寄存器的最高位。当LFSR以短模式运行时，我们同时将该值设置到LFSR的bit 7中。最后，我们将LFSR的值整体右移一位，以完成一次更新操作。

### 添加调试面板信息显示

在完成上述代码后，通道4的所有功能便实现完毕。我们可以在DebugWindow::apu\_gui函数中添加以下代码，以可视化通道4中程序写入的波形以及音频参数：

```
void DebugWindow::apu_gui()
{
    if (g_app->emulator)
    {
        if (ImGui::CollapsingHeader("APU"))
        {
            /*...*/
            if(ImGui::CollapsingHeader("Audio Channel 3"))
            {
                /*...*/
            }
            if(ImGui::CollapsingHeader("Audio Channel 4"))
            {
                ImGui::Text("Period: %u", (u32)g_app->emulator->apu.ch4_period);
                ImGui::Text("Update Frequency : %f", 1048576.0f / (f32)g_app->emulator->apu.ch4_update_freq);
                ImGui::Text("Volume: %u/16", (u32)g_app->emulator->apu.ch4_volume);
                ImGui::Text("Initial Volume: %u/16", (u32)g_app->emulator->apu.ch4_initial_volume);
                ImGui::Text("Envelope Direction: %s", g_app->emulator->apu.ch4_envelope_direction);
                ImGui::Text("Envelope Sweep Pace: %u", (u32)g_app->emulator->apu.ch4_envelope_sweep_pace);
            }
        }
    }
}
```

此时编译并运行模拟器，读者应该就能听到APU的四个通道产生的完整音频输出了。



恭喜！在学习完了本章内容后，您已经完成了GameBoy模拟器98%的内容了。下一章将是我们GameBoy模拟器之旅的最终章，我们将为我们的APU实现高通滤波器，并回顾我们的整个GameBoy模拟器学习之旅，为本专题系列文章画上一个完美的句号。那么，下一章见！

銀葉吉祥：从零开始实现GameBoy模拟器 #15 高通滤波器，总结

28 赞同 · 3 评论 文章



编辑于 2024-03-19 22:31 · IP 属地上海

游戏机模拟器

Game Boy（GB）



欢迎参与讨论

1 条评论

默认

最新



开始思考



03-13 · IP 属地江苏

回复

喜欢

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

折腾树莓派3跑retropie玩模拟器的几点感受和吐槽

过年前买了块树莓派3打算玩模拟器用。树莓派上可以选的主流模拟器前端有retropie和recalbox可以选择，底层都是emulation station + retroarch的样子，后者貌似对树莓派3的板载蓝牙驱动适配...

凤凰院胸针



为了找出最好用的安卓模拟器，我进行了一次众测

阿虚同学

发表于阿虚同学

2020年优秀的手机模拟器

有电脑的朋友，一般都喜欢在电脑上安装一个手机模拟器，对于喜欢打游戏的同学来说，它可以在电脑上操作手机端的游戏；但今天我给大家介绍一下，不用来打游戏，手机模拟器还能有哪些好处呢，...

南柯一梦

小白入门安卓（2）Genymotion模拟器下载安装运

前面已经对AS进行了基本的配置，现在觉得下一步应该是把模拟器先弄上去先。这里觉得AVD不好用，所以根据大佬的建议，装Genymotion模拟器。1.4 Genymotion模拟器安装 跟着这个享受生活 发表于项目

赞同 15



1 条评论

分享

喜欢

收藏

申请转载

