从零开始实现GameBoy模拟器 #12 APU

銀葉吉祥
浙江大学 软件工程硕士

欢迎来到从零开始实现GameBoy模拟器第十二章。在本章中，我们将开始APU部分的学习和实现。我们会首先讲解音频信号处理的基本原理，然后讲解GameBoy APU的机能和架构，最后编写代码实现基础的音频输出框架。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-12，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

音频编程基础

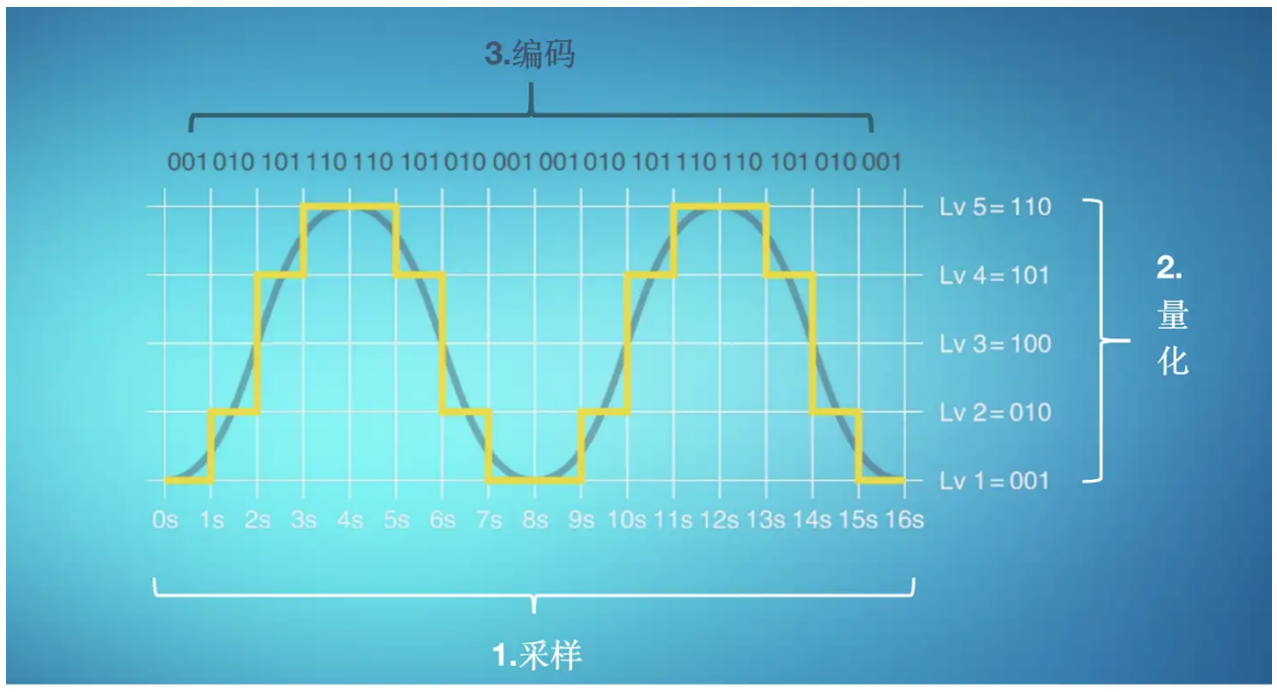
考虑到相比图形编程，大部分读者对音频编程还是比较陌生的，因此我们在这里花一些时间，讲解一下音频编程中的一些基础概念。对这部分内容比较熟悉的读者，可以直接跳到下一节GameBoy APU的架构部分继续阅读。

人耳之所以能够听到声音，是因为空气或者其它介质的振动带动了人耳鼓膜的振动，并通过耳蜗转化成电信号传入了人脑。而空气之所以会振动，是因为音源（例如扬声器）的振动带动了周围空气振动，并通过空气在空间中进行传导。大部分的扬声器构造都是一个带有线圈和永磁体的电磁铁装置，当我们给线圈通上规律变化的电流时，线圈中产生的磁场就会导致线圈与永磁体之间产生相对运动，此时我们只需要将线圈或者磁体的一方固定，而另一方连接到喇叭的振膜上，产生的相对运动就会导致振膜产生物理运动，并推动周围的空气，产生声波。

在理想情况下，扬声器产生的声音应当真实反应我们给扬声器提供的电信号。例如，如果我们给扬声器通上一个频率为256Hz的交变电信号，则扬声器应当产生一个频率为256Hz的声音，即一个C调的音。然而，在真实世界中，所有的扬声器都无法精准还原电信号，而是会产生一定程度的失真。扬声器可以通过改进喇叭材质、通过专门的DSP硬件改变电信号等方式尽可能降低失真，然而失真只可被减少，不可被完全抑制，并且有一部分的失真反而会让声音的音色产生一些独特的韵味，因此并不完全是一件坏事。尽管如此，由于扬声器的失真在大多数情况下取决于用户所使用的硬件，并不为我们程序所控制，因此我们在音频编程时不会考虑失真问题，可以简单认为用户听到的音频就是我们传输给扬声器的一段连续变化的电信号。

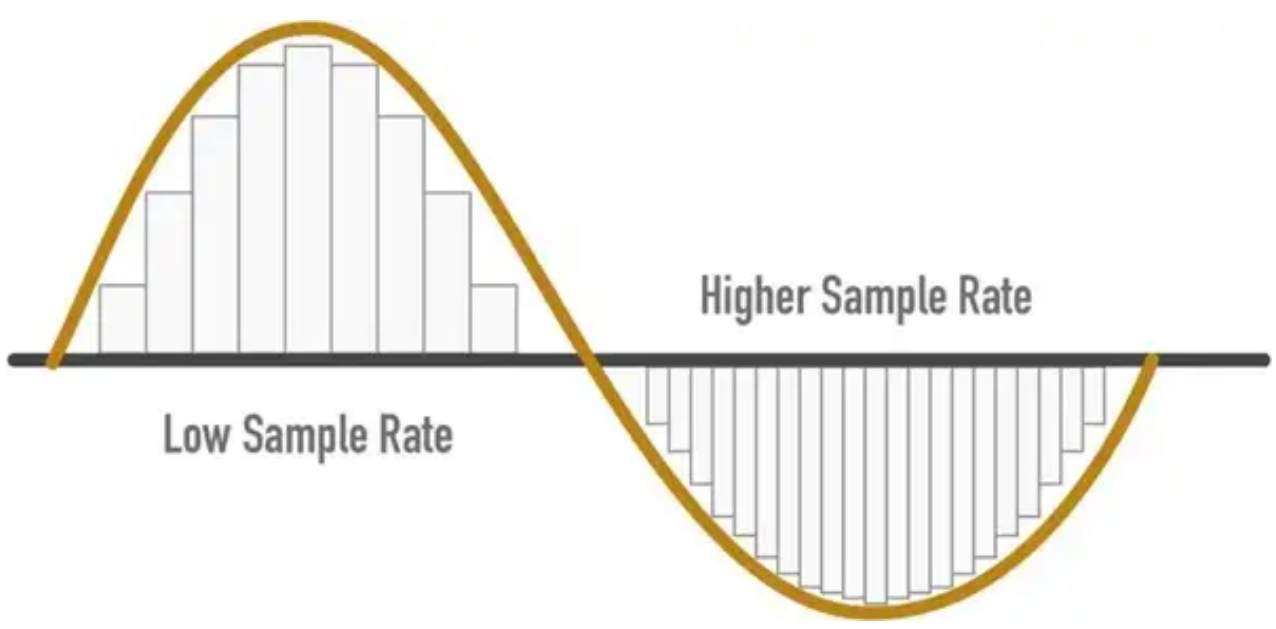
在电信领域（包括计算机）中，所有连续变化的电信号有一个专用名词：模拟信号（analog signal），因此我们传递给扬声器的信号自然也是一种模拟信号。然而，模拟信号并不能被计算机直接处理，由于计算机以字节作为基本的存储和操作单位，其实际上只能操作一个一个单独的数字，因此我们需要一种方式，将连续的模拟信号用字节的形式表示，从而让计算机能够处理音频信号，而这个处理方式就是采样（sampling）。采样的工作原理是在连续的模拟信号上每隔特定的时间测量一次信号的强度，并将测量到的信号强度转换（量化）为一个数字保存在计算机中，从而使用一串连续变化的数字来近似代表一个模拟信号。由于采样后的信号以数字序列的形式来保存，因此其也被称为数字信号（digital signal），用来与模拟信号做区分，而采样的过程

也被称模数转换（analog-digital converting）或者信号编码（signal encoding）。下图比较清晰的展示了信号采样和量化的过程：



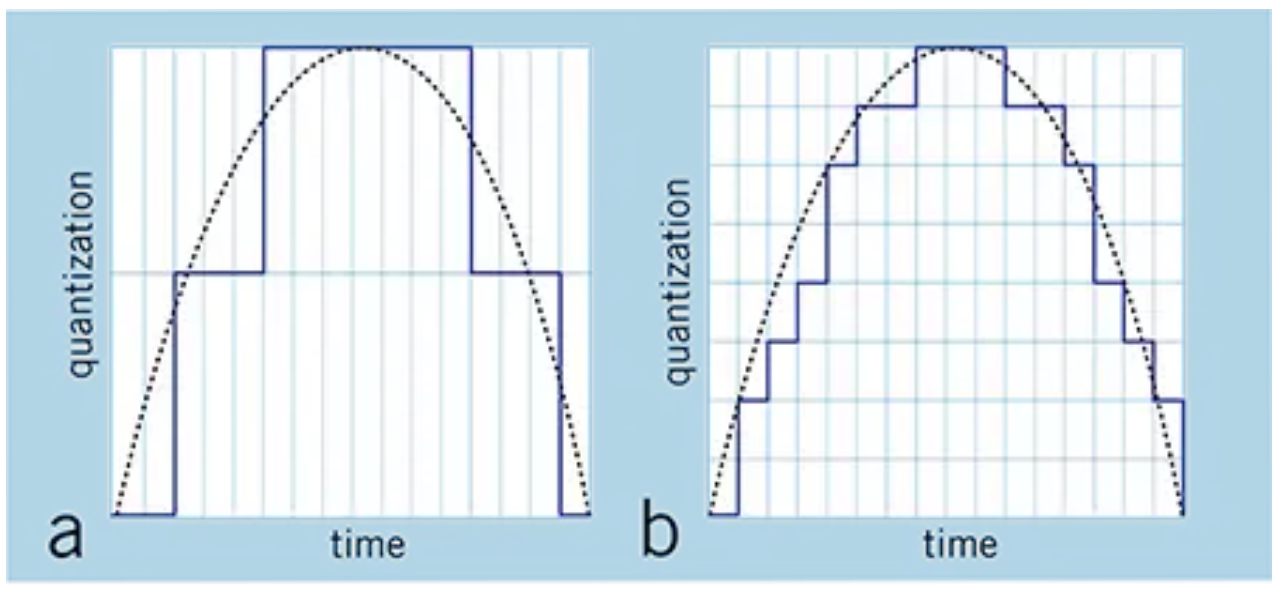
从图中也能明显看到，信号在经过采样和量化后，计算机所存储的数字信号并不能完整表达原始的模拟信号，而是会有一定的失真，这种由于采样和量化导致的失真也被称为数字失真（digital distortion）。在大部分时候，数字失真都是一种需要避免的问题，但是正如所有的失真都有其韵味，数字失真在某些时候也会作为一种特殊的音频处理手法使用，来创作我们所熟知的8比特音乐——也正是GameBoy等掌机的所开创的音乐流派。

一段未压缩的数字信号主要有两个基本参数：采样率和位深度。采样率（sample rate）描述了信号在时间轴上的采样频率，采样率越高，则信号的变化细节能够被更多地保留下来，反映到声音信号上则是能够更大程度保留声音的高频细节，如下图所示：



采样率使用Hz为单位表示，即每秒钟采样次数。常见的音频信号采样率一般为44100Hz、48000Hz、96000Hz、192000Hz等。

位深度（bit depth）则是对于每一个样本，我们在转换成数字后使用多少个比特来存储该数据。我们使用的比特越多，则每一个存储的样本数值与其原本的数值之间的差异越小，即量化误差越小，声音的品质也就更好，如下图所示：



来源：https://legacy.presonus.com/learn/technical-articles/sample-rate-and-bit-depth

常见的音频信号位深度一般为16bit、24bit和32bit。8bit的位深度会导致信号有明显失真，因此在一般场合下不会使用。读者可以观看下面的视频，直观感受不同的位深度对最终音质产生的影响：

https://www.bilibili.com/video/BV1hd4y1M7nt
www.bilibili.com/video/BV1hd4y1M7nt

正如我们之前所说，计算机所有能处理的信号均为数字信号，因此当我们编写代码输出音频信号

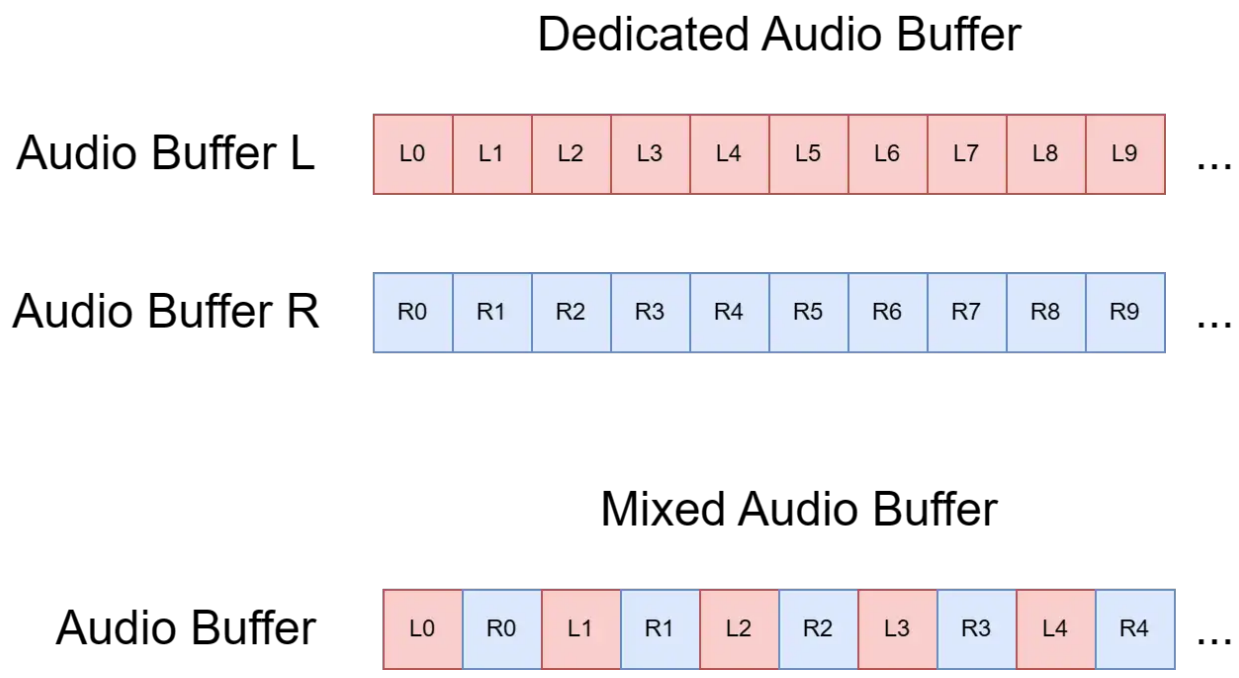
时，我们实际上提供给计算机音频驱动的也是数字信号，即由字节数组定义的信号。当计算机需要实际输出信号给扬声器时，计算机需要将数字信号重新转换为模拟信号，才能产生用于驱动扬声器的电信号，该过程被称为数模转换（digital-analog converting），是模数转换的逆过程。在大部分计算机上，模数转换由一颗焊在主板上的单独的芯片进行，这颗芯片一般被称为解码芯片或者DAC（digital-analog converter）芯片，CPU只需要将数字信号发送给该芯片就可以完成数模转换。例如瑞昱（Realtek）就是生产音频编解码芯片的主要厂商。

一串连续的音频信号构成一个声道（sound channel），并输出到一个扬声器进行播放，而当今主流的计算机都支持至少两个声道的音频信号同时输出，以构成立体声系统（stereo sound system）。立体声音频分为左右两个声道，这两个声道的信号会分别输出给位于听众左侧和右侧的扬声器，从而让声音匹配听众的双耳，让声音具有方位信息。除了立体声系统外，部分设备还支持5.1声道、7.1声道系统，这些系统通过在听众四周放置多个扬声器，组成环绕声系统（surrounding sound system），从而进一步增强声音的沉浸感。

在为多声道设备准备音频信息时，不同的音频API通常会要求使用以下两种方式之一来提交音频信息：

1. 为每一个声道单独分配声音数据缓存。在这种模式下，多声道信号可以简单理解为由多个单声道信号独立输出而成，在数据处理时我们不需要进行任何额外操作。
2. 使用同一个声音数据缓存来保存所有声道的信息。在这种模式下，不同声道信号在同一时间点的数据需要依次写入缓存中，等到所有通道数据都写完以后才统一开始写下一时间点的数据。

下图展示了两种不同的格式下多声道音频数据的保存方法：



两种音频数据保存方式并没有孰优孰劣，一个正确实现的音频驱动在读取两种音频格式时的行为是完全一致的。在LunaSDK中，我们使用第二种混合音频缓存来记录和提交多声道音频信号。

为了使用计算机输出音频信号，一个典型的程序需要执行以下步骤：

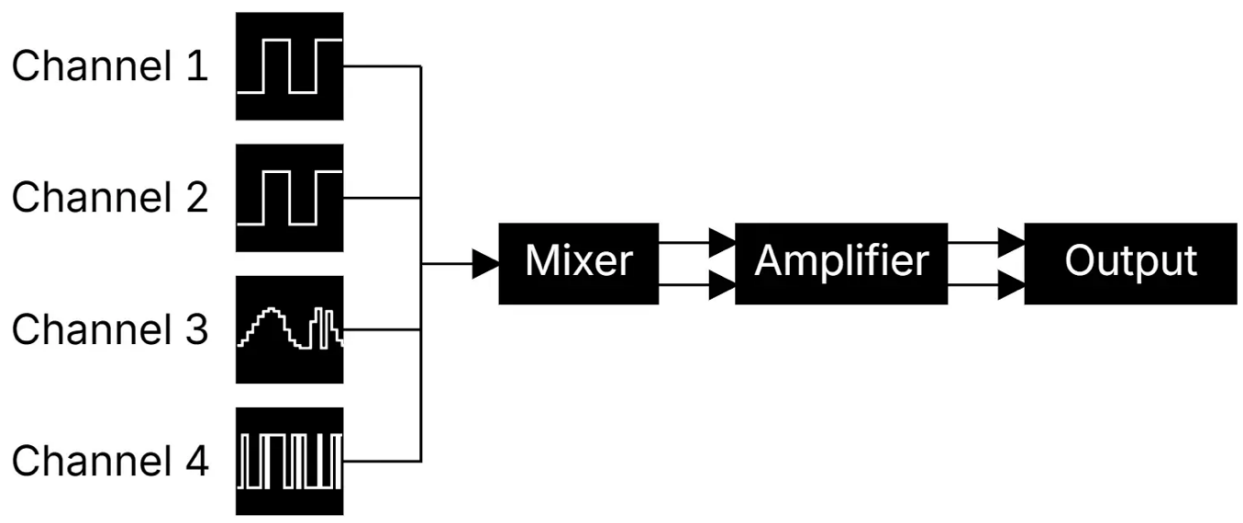
1. 初始化音频API。每一个系统都会提供若干种不同的音频API供程序使用，这些API有的仅具备基础的音频输入输出功能，但是延迟非常低，专注于实时类应用程序使用；有的提供混音、滤波、混响等高级DSP（digital signal processing）功能，但是延迟较高，适合专业离线混音软件使用。对于游戏开发来说，程序通常会选择一个功能较少，但延迟足够低的API，并使用自己设计或者第三方SDK来基于该API实现更高级的音频处理功能。
2. 选择音频设备。与图形设备一样，大部分的系统支持同时接入多个音频设备，例如耳机、扬声器、使用HDMI连接的显示器，这些设备都在系统中有相应的驱动，并可以在系统的音频选项卡中切换优先级。程序会在初始化音频API以后首先查询系统具有的所有音频设备，然后选择一个设备用于输出音频。理论上来说，程序也可以通过创建多个虚拟设备对象来向不同的音频设备发送不同的音频信号，但是大部分程序都是选择音频设备列表中的第一个设备（即系统的默认音频设备），并使用该设备作为音频输出设备。
3. 创建虚拟设备对象。在选择了音频设备以后，程序需要继续查询音频设备支持的声音格式，并使用其中一种声音格式来创建虚拟设备对象。声音格式主要包括音频信号的采样率、位深度、声道数量等信息，每一个音频设备通常都支持多种不同的声音格式，但是该设备在创建了虚拟设备对象以后就不可再更改，只能通过重新创建虚拟设备对象的方式来修改。
4. 准备音频数据。音频数据是由一系列音频采样（sample）构成的数组，每一个采样使用一个数字存储，代表了音频信号在一个特定时间点的信号强度。程序既可以从音频文件中加载音频数据，也可以通过某种数字信号处理算法实时生成音频数据。例如，GameBoy的所有音频数据都是由APU实时生成的，从而仅需要很少的空间就可以存储大量不同样式的游戏音乐。
5. 创建音频缓存。由于音频播放是一个连续的行为，其天生具有异步的特性，因此所有的音频API都要求程序先将需要播放的音频信号写入一个指定的音频缓存中，再由音频驱动异步从缓存中读取音频数据并播放。视音频API的设计不同，有的API要求用户自己创建音频缓存，写入数据以后提交给音频API读取，有的音频API则由音频驱动内部创建缓存，然后在需要音频数据时调用用户程序的回调函数，让用户程序将音频数据写入内部音频缓存中。
6. 将音频数据写入音频缓存。该过程需要在程序运行期间不断执行，才能保证音频的播放不出现间断。在程序创建音频缓存的模式中，音频API通常会提供一个播放队列，程序需要不断往该队列中通过添加新的音频缓存的方式添加新的数据，以保证音频驱动能够一直从队列中取出音频缓存来播放其中的数据；在音频API创建音频缓存的模式种，程序需要保证在每次回调函数被调用时向缓存中写入足够的音频数据，以保证音频驱动一直可以读取新的音频数据播放。

由于音频信号的中断比画面卡顿更加影响游戏体验，因此在大部分游戏中，音频信号的播放控制通常放在单独的音频线程中进行，且该线程相比普通线程具有更高优先级，这就是为什么在部分游戏中，即使画面卡顿严重，游戏的BGM还是能够正常播放的原因之一。

GameBoy的音频处理单元

音频处理单元（Audio Processing Unit，简称APU）是GameBoy中用于产生和播放音频信号的组件。GameBoy的音乐有时候被称为“8-bit音乐”，并不是因为声音信号具有8比特的位深度，而是因为GameBoy的音色代表了当年绝大部分8bit处理器的游戏机的音色，而这种音色实际上与当年游戏机的APU使用的音频信号生成机制有着密不可分的联系。具体来说，GameBoy的APU一共有四个音频信号生成电路，其可以生成四路独立的单声道信号，这四路信号在生成以后会由硬件混音器（mixer）合成，每一路信号都可以独立输出至左声道、右声道或者双声道，从而构成GameBoy的立体声输出。有趣的是，由于GameBoy在硬件上只有一个扬声器，因此如果使用GameBoy的默认外放，其会将立体声输出的左右声道再次混合，变成单声道信号输出，而只有当使用3.5mm耳机输出音频时，GameBoy才会真正播放立体声音频。

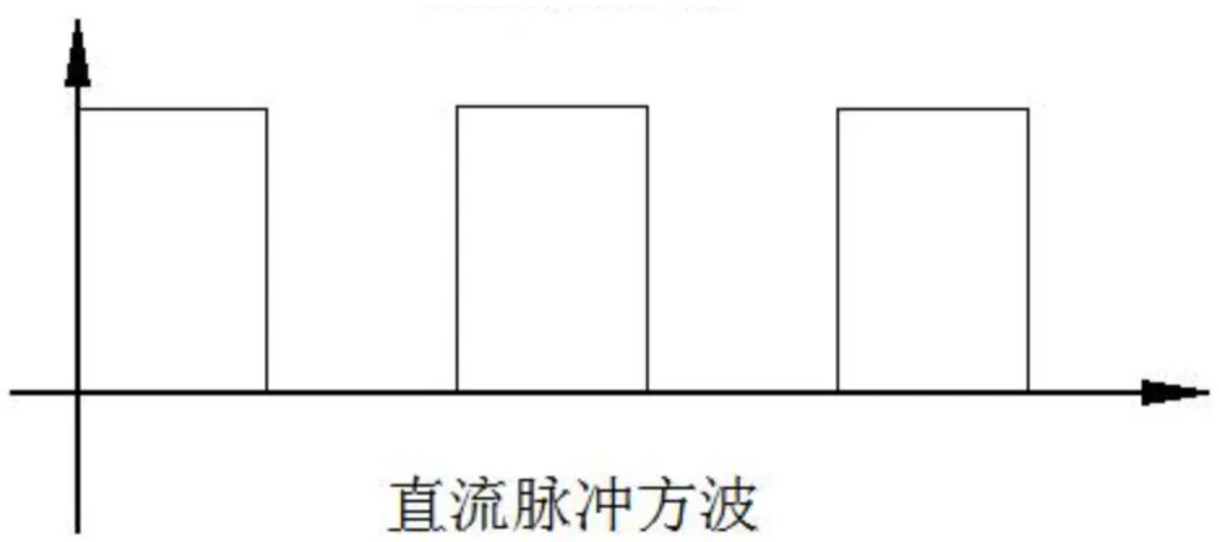
GameBoy的四个音频信号生成电路并不是完全一致的，其可以按照音频生成的原理和效果不同，划分为2+1+1的组合：2个脉冲信号通道，1个波形信号通道和1个噪声信号通道。下图展示了GameBoy APU的大致架构：



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

脉冲信号通道

脉冲信号也叫方波信号，是一种只有0和1的信号，如下所示：



脉冲信号非常容易通过硬件生成，只需要按照一定的时间间隔切换电路的通断状态即可产生，并且产生的音频信号充满了“电子味”，也是在8-bit音乐最有标志性的音色。例如下面这个视频就很好地展示了脉冲信号以及用其制作的8bit音乐：

https://www.bilibili.com/video/BV1h2421F7HR
www.bilibili.com/video/BV1h2421F7HR

GameBoy一共搭载了两个脉冲信号通道：通道0和通道1。大部分游戏使用这两个通道来分别播放背景音乐的主旋律和伴奏。GameBoy允许程序通过寄存器来控制脉冲信号的频率和响度，并且可以使用硬件的sweep和envelope功能来在播放脉冲信号的过程中自动调节通道的频率和响度。因此从某种程度上来说，GameBoy本身就是一个非常好的8bit音乐合成器（笑），而也确实有游戏充分利用了该特性，在GameBoy上开发了音乐制作软件，例如POCKET MUSIC：

波形信号通道

波形信号通道允许程序指定一段自定义的音频数据，并在该通道中反复播放。GameBoy支持的自定义音频数据具有固定的32个采样，每一个采样具有4个比特的位深度，因此总尺寸为16字节。与脉冲通道一样，GameBoy也支持程序自由控制波形信号通道的频率和响度。虽然从理论上来说，程序可以通过不停地更新音频缓存中的32个音频数据采样来播放一段相对完整的预录音频，但是在实际上，大部分游戏都会选择使用波形信号通道来播放一些简单的，具有特定音色的波形，来制作游戏中的一些特殊效果音，例如获取金币时的提示音。

噪声通道

噪声通道通过一个伪随机数产生器来产生随机的音频信号，从而制造类似白噪声的音频信号。尽管噪声通道并没有固定的波形，GameBoy仍然允许程序控制噪声通道产生随机数的频率，从而在一定程度上“控制”噪声的音高。与其他所有通道一样，噪声通道的响度也可以被单独控制，并支持使用envelope随着时间自动调整噪声通道的响度。在实际的游戏中，噪声通道一般用于产生规律性的擦擦音，从而模拟架子鼓中鼓和镲的声音，为BGM添加一个鼓点音色。

VIN通道

除了上述四个常规音频通道以外，单色GameBoy实际上还有第五通道，被称为VIN通道。该通道没有任何音频生成电路，只在混音器中保留有一个模拟信号输入引脚，可以从卡带直接输出音频信号至混音器。该通道设计的本意是让游戏厂商能够在卡带上安装自己的音频芯片，并输出信号给GameBoy APU用于混音输出，但是没有任何在GameBoy上正式发售的商业游戏使用了该特性，因此该通道在CGB（GameBoy Color）及后续机型中被移除。

接入音频API

在了解了音频编程的基础知识后，就让我们开始实际编写代码，将音频API接入我们的模拟器程序吧！首先我们需要在App.hpp中添加音频API的头文件：

```
#include <Luna/AHI/Device.hpp>
```

然后我们在App类中添加下列变量和函数：

```
//! The audio device.
Ref<AHI::IDevice> audio_device;

RV init_audio_resources();
```

LunaSDK使用AHI（Audio Hardware Interface）模块提供系统的音频API，其中AHI::IDevice表示一个创建完毕，可以用于输入或者输出音频信号的虚拟音频设备。init_audio_resources函数用于初始化音频相关的资源，其中就包括选择音频设备、创建虚拟音频设备，以及我们在下一章将会

继续讲解的音频缓存相关资源。接着，我们在App.cpp中实现init_audio_resources函数：

```
RV App::init_audio_resources()
{
    lutry
    {
        // Enumerate and select adapters.
        Vector<Ref<AHI::IAdapter>> adapters;
        luexp(AHI::get_adapters(&adapters, nullptr));
        // Choose primary adapter.
        AHI::IAdapter* choosed_adapter;
        for(auto& ada : adapters)
        {
            if(ada->is_primary())
            {
                choosed_adapter = ada;
                break;
            }
        }
        luassert(choosed_adapter);
        log_info("LunaGB", "Audio adapter: %s", choosed_adapter->get_name());
        // Create device.
        AHI::DeviceDesc desc;
        desc.flags = AHI::DeviceFlag::playback;
        desc.sample_rate = 0;
        desc.playback.adapter = choosed_adapter;
        desc.playback.bit_depth = AHI::BitDepth::f32;
        desc.playback.num_channels = 2;
        luset(audio_device, AHI::new_device(desc));
        // Add playback data callback.
        audio_device->add_playback_data_callback(on_playback_audio);
    }
    lucatchret;
    return ok;
}
```

init_audio_resources函数主要分三步进行：

1. 首先，我们需要枚举系统上安装的所有物理音频设备，该功能通过AHI::get_adapters函数实现。大部分的系统同时支持播放音频和录制音频，因此get_adapters函数的两个参数可以分别用于获取系统的播放设备和录制设备列表。由于我们的模拟器只需要关注播放设备，因此第二个参数可以直接设置为nullptr，表示我们不需要获取录制设备的列表。每一个枚举到的设备要么是播放设备，要么是录制设备，如果有一个设备同时支持播放和录制（例如声卡），那么其在列表中会表现为名称相同的两个独立的设备。
2. 然后，我们需要在获取的设备列表中选择一个设备，用来创建我们的虚拟设备。在大部分情况下，我们只需要直接通过IAdapter::is_primary函数判断该设备是否为系统的默认音频设备，并使用音频设备来作为我们的选择设备就行。对于某一些有特殊需求的专业音频软件，其可能会在设置面板中提供让用户选择音频设备的选项，并可以根据选项在这里选择相应的设备。
3. 然后，我们需要使用选择的设备创建AHI::IDevice虚拟设备对象。每一个IDevice对象都可以同时进行声音播放和声音录制操作，因此大部分的应用程序只需要创建一个AHI::IDevice对象就足够使用。在指定AHI::IDevice的参数是，程序可以为播放和录制设置不同的位深度、通道数量和音频设备，但是采样率必须保持一致。在我们的模拟器中，由于我们只需要音频播放功能，不需要音频录制功能，因此我们只为AHI::IDevice开启了播放功能（通过AHI::DeviceFlag::playback标志位指定），并且只设置了播放时使用的音频参数。在设置音频参数时，我们可以将采样率设置为0，表示让音频驱动自己选择最合适的采样率，从而兼容更多的音频设备。
4. 最后，我们可以通过AHI::IDevice::add_playback_data_callback函数注册音频数据回调函数来向设备提供音频数据。AHI::IDevice会在底层创建单独的音频线程来与硬件同步音频数据，因此一旦我们的on_playback_audio函数被注册，其就会在音频线程中被反复调用，因此当我们需要实际拷贝音频数据时，我们需要使用适当的线程同步机制来保证主线程和音频线程不至于出现数据竞争。

最后，我们需要在App.cpp中实现on_playback_audio函数：

```
u32 on_playback_audio(void* dst_buffer, const AHI::WaveFormat& format, u32 num_
{
    u32 num_frames_read = 0;
    while(num_frames_read < num_frames)
    {
        // Clear audio data for now.
        ((f32*)dst_buffer)[num_frames_read * 2] = 0.0f;
        ((f32*)dst_buffer)[num_frames_read * 2 + 1] = 0.0f;
        ++num_frames_read;
    }
    return num_frames_read;
}
```

在获取播放音频数据的音频回调函数中，音频API会提供三个参数：

1. dst_buffer：用于接收程序的音频数据的缓存。程序需要将音频数据写入到该缓存中。音频API保证该缓存的尺寸至少为num_frames * format.num_channels * format.bit_depth字节。
2. format：程序写入的音频数据需要遵循的格式，包括采样率（sample_rate）、位深度（bit_depth）、通道数量（num_channels）三个属性。
3. num_frames：在本次调用中音频驱动期望程序写入的帧数量，每一帧包括了format.num_channels个音频采样（sample）数据，分别对应不同音频通道在该时间点的音频采样数据。

函数的返回值表示程序在这一次回调中实际写入的音频帧的数量，该值应当小于等于函数的num_frames参数。如果该值小于num_frames，则音频有可能因为程序没有提供足够多的音频数据而出现中断。由于on_playback_audio会被反复调用，且音频被写入缓存到音频被播放之间存在一定的时差，因此只要程序在下一次调用时继续提供了足够的音频帧数据，那么音频基本上仍然能够正常播放。只有当程序持续性地提供过少的音频数据时，音频才会产生明显的卡顿。

需要注意的是，如果on_playback_audio本身开销很大，很长时间后才返回，也会导致音频数据跟不上播放而产生卡顿，因此我们不会在on_playback_audio中实现特别复杂的声音生成和信号处理算法，而是会使用单独的线程执行这些操作，将处理后的音频数据写入一个中间缓存中，并只在on_playback_audio中执行数据拷贝操作，从而让on_playback_audio尽可能快地返回。由于我们在这一章节中还没有真正开始实现APU，因此我们只是简单地输出空信号，从而让音频设备静音。

最后，我们在App::init中调用init_audio_resources函数：

```
RV App::init()
{
    lutry
    {
        /*...*/
        luexp(init_render_resources());
        luexp(init_audio_resources());
    }
    lucatchret;
    return ok;
}
```

此时编译并运行程序，可以看到程序已经能够正常识别音频设备，并创建音频设备对象了（根据系统音频设备的不同，此处显示的名字可能有所不同）：

```
[LunaGB]Info: Audio adapter: 扬声器 (Realtek USB Audio Front)
```

以上就是本章节的全部内容了。在下一章中，我们将实现GameBoy APU的前两个脉冲信号通道，并将信号输出至本章创建的音频设备中，让我们的模拟器第一次开始播放声音。那么，下一章见！

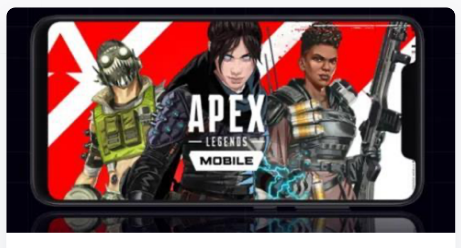
銀葉吉祥：从零开始实现GameBoy模拟器 #13 脉冲音频通道
12 赞同 · 0 评论 文章

编辑于 2024-03-19 22:31 · IP 属地上海

游戏机模拟器 Game Boy（GB）

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



以《ApexM》为例，如何用UI的力量突破设计体验?

腾讯游戏学堂

为Apex搭建的游戏钢炮主机：ROG M11G+i9+RTX2080Ti

Tiger... 发表于Tiger...

Apex安装和使用

在Linux系统下安装Apex库安装流程（按顺序使用如下命令）git clone https://github.com/NVIDIA/apex cd apex pip3 install -v --no-cache-dir ./注意：不能直接使用

小松鼠 发表于数据挖掘

记一次在 RTX 3090 上安装 APEX

千千