Game Boy Sound Emulation

This is a short article on the Game Boy sound hardware with the perspective of emulating it. The aim of this article is to get your emulator to a state which sounds right, but not necessarily pass all sound tests.

Primer on Sound

Before we start to look at things specific to the Game Boy, you must know the basics about sound. A great resource to learn these elementary things is the video by the people at xiph.org.

After watching the section on audio, you can continue reading below.

Small Overview of Sound Hardware

The Game Boy has four distinct audio channels which can output stereo sound[1]. These channels are listed below,

1. Square Wave (Envelope and Sweep)
2. Square Wave (Envelope)
3. Custom Wave (4 bit samples)
4. White Noise (Envelope)

In addition to this, the Game Boy APU has three basic control registers,

1. NR50 - Controls the volume of the left and right stereo channels.
2. NR51 - Controls where the sound from the above channels is panned.
3. NR52 - Controls whether the APU is enabled or disabled.

We will now take a look at the channel two audio, as it is the simplest to understand.

Channel Two

The Game Boy sound channel two produces square waves. A square wave is a sound wave with two distinct amplitudes. The wave has peaks and valleys.

```
___|^^^|_____|^^^|___
```

Let us assign the amplitude values of `1` to the peaks and `0` to the valleys. The channel can produce only one of these values as its output[2]. There are four *wave pattern duties* of these square waves, which are *predetermined*.

Duty	Waveform	Ratio
0	00000001	12.5%
1	00000011	25%
2	00001111	50%
3	11111100	75%

You can select any one of these duties by writing the duty number to the bits `7-6` in the `NR21` register. The channel has two elementary associated values called the *frequency timer* and the *wave duty position*.

The role of frequency timer is to step wave generation. Each T-cycle the frequency timer is decremented by `1`. As soon as it reaches `0`, it is reloaded with a value calculated using the below formula, and the wave duty position register is incremented by `1`.

The frequency timer is calculated with the following formula,

```
Frequency Timer = (2048 - Frequency) * 4;
```

Here, the value of the `Frequency` variable is provided by the NR23 and NR24 registers. The wave duty position is just a pointer to one bit in the wave pattern duty currently selected. So when you try to

get the amplitude of this channel, it can be easily calculated as,

```
Amplitude = WAVE_DUTY_TABLE[WAVE_DUTY_PATTERN][WAVE_DUTY_POSITION]
```

Note: The wave duty position is wrapped back to 0 when it goes above 7.

Frame Sequencer

The frame sequencer (FS) is a component of the audio system which generates clocks for other units (Sweep, Envelope and Length). Every 8192 T-cycles (512 Hz) the FS is stepped, and it *might* clock other units. There is a nice table from the gbdev wiki which shows which unit is clocked and when it is clocked.

Step	Length Ctr	Vol Env	Sweep
0	Clock	-	-
1	-	-	-
2	Clock	-	Clock
3	-	-	-
4	Clock	-	-
5	-	-	-
6	Clock	-	Clock
7	-	Clock	-
Rate	256 Hz	64 Hz	128 Hz

The frame sequencer clocks are derived from the DIV timer. In Normal Speed Mode, falling edges of bit 5 step the FS while in CGB Double Speed Mode, bit 6 is used instead. Here bits 5 and 6 refer to the bits of the upper byte of DIV (internally DIV is 16 bit but only the upper 8 bits are mapped to memory).

Envelope Function

Envelope is a way to adjust the volume of a channel periodically. The NRx2 registers (NR12, NR22, NR42) control the the envelope function for that specific channel.

The register provides the following values to the envelope function,

1. The initial volume of the channel (bits 7-4)
2. The direction of the envelope function (bit 3)
3. The period in which one envelope step should take place (bits 2-0)

Now, what do these values mean exactly?

The first parameter is self-describing, it is the initial volume of the channel which will be adjusted periodically. The second parameter is easy to understand as well, it tells in which direction the envelope will take place (that is will the volume be incremented or decremented). The third parameter specifies the amount of volume clocks the frame sequencer needs to provide for an envelope step to take place.

In addition to the above, we maintain two extra values called the period timer and the current volume.

How do all these tie together then?

On a trigger event, (event where the bit 7 of NRx4 is enabled), the internal period timer is loaded with the third parameter of NRx2, and the current volume register is loaded with the first parameter.

Now on a volume clock from the frame sequencer the following steps take place,

1. If the period (parameter three, not the period timer!) is zero, return.
2. If the period timer is a non-zero value, decrement it.
3. Now, if due to the previous step, the period timer becomes zero only then continue with step 4.
4. Reload the period timer with the period (parameter 3).
5. If the current volume is below 0xF and the envelope direction is upwards or if the current volume is above 0x0 and envelope direction is downwards, we increment or decrement the current volume respectively.

All of the steps, in pseudo-code:

```
if period != 0 {
    if period_timer > 0 {
        period_timer -= 1
    }

    if period_timer == 0 {
```

```
        period_timer = period

        if (current_volume < 0xF and is_upwards) or (current_volume > 0x0 and !is_upwards) {
            if is_upwards {
                adjustment = +1
            } else {
                adjustment = -1
            }

            current_volume += adjustment
        }
    }
}
```

Now, with this the output values of the channel change a bit too. The current duty cycle is multiplied by the current volume and scaled by the DAC (Digital to Analog Converter).

```
# A value between 0x0 to 0xF will be fed into the DAC.
dac_input = DUTY_CYCLE_OUTPUT * CURRENT_VOLUME

# A value between -1.0 to +1.0 will be emitted by the DAC.
dac_output = (dac_input / 7.5) - 1.0

return dac_output
```

This section also applies verbatim to channel one and channel four. (Channel three doesn't have an envelope unit).

Sweep Function

Sweep is a way to adjust the frequency of a channel periodically. It is controlled by the NR10 register for channel one. (None of the other channels have a sweep unit).

The register provides the following values to the sweep function,

1. The period in which one sweep step should take place (bits 6-4)
2. The direction of the sweep function (bit 3)
3. The amount by which the shadow frequency should be shifted (bits 2-0)

The first parameter specifies the amount of sweep clocks the frame sequencer needs to provide for an sweep step to take place. The second parameter is similar to the envelope direction, it specifies whether the frequency will increase or decrease over time. The third parameter specifies the value by which the previous frequency will shifted to get a new frequency.

We will maintain three extra values called *sweep enabled*, *shadow frequency* and *sweep timer*.

Now on a sweep clock from the frame sequencer the following steps occur,

1. If the sweep timer is greater than zero, we decrement it.
2. Now, if due to previous operation the sweep timer becomes zero only then continue with step 3.
3. The sweep timer is reloaded with the sweep period value. **If sweep period is zero, then sweep timer is loaded with the value 8 instead.**
4. If sweep is enabled and sweep period is non-zero, a new frequency is calculated. In this step we also perform an additional routine called as the overflow check.
5. Now, if the new frequency is less than 2048 and the sweep shift is non-zero, then the shadow frequency and frequency registers are loaded with this new frequency.
6. We now perform the overflow check again.

The pseudo-code to calculate the new frequency is as follows,

```
new_frequency = shadow_frequency >> sweep_shift

// Is decrementing is the sweep direction.
if is_decrementing {
    new_frequency = shadow_frequency - new_frequency
} else {
    new_frequency = shadow_frequency + new_frequency
}

/* overflow check */
if new_frequency > 2047 {
    channel_enabled = false
}
```

```
return new_frequency
```

The pseudo-code for the procedure described above the frequency calculation routine is as follows,

```
if sweep_timer > 0 {
    sweep_timer -= 1
}

if sweep_timer == 0 {
    if sweep_period > 0 {
        sweep_timer = sweep_period
    } else {
        sweep_timer = 8
    }

    if sweep_enabled and sweep_period > 0 {
        new_frequency = calculate_frequency()

        if new_frequency <= 2047 and sweep_shift > 0 {
            frequency = new_frequency
            shadow_frequency = new_frequency

            /* for overflow check */
            calculate_frequency()
        }
    }
}
```

On a trigger event, the following things happen (which also enable sweep)

1. The shadow frequency register is loaded with the current frequency.
2. The sweep timer is loaded with sweep period. **If sweep period is zero, then the sweep timer is loaded with the value 8 instead.**
3. The sweep enabled register is set if the sweep period is non-zero OR sweep shift is non-zero.
4. If sweep shift is non-zero we run the overflow check. (See the frequency calculation routine).

Length Function

Each channel, has a length function associated with it. If this function is enabled, then a special timer, called the length timer, disables the channel when a certain time has passed. For example, in channel two, the length data is stored in the lower 6 bits of NR21. The length data is extracted, and the length timer is set with the following formula,

```
Length Timer = 64 - Length Data
```

Note: Replace 64 with 256 in case of channel three only.

Whenever a length clock is provided by the frame sequencer AND bit 6 of NR24 register is set, the length timer is decremented by one.

If the length timer, (only) as a result of the above decrement reaches the value of zero then the channel is disabled until the next trigger event.

On a trigger event, the length timer is loaded with the value 64 or 256 (channel three only) if the length timer is equal to zero.

Channel Three

We now come on to channel three. It is also really simple to understand and you won't have a problem.

In channels one and two we played back predefined square waves at custom frequencies, in channel three instead of playing predetermined sequences the ROM instead provides 32 4-bit custom samples for us to playback.

These 4-bit samples are stored in a region called the Wave RAM. This region is located at $FF30-$FF3F.

It is a 16 byte region, therefore there are two samples contained within one byte. We play the upper four bits of a byte before the lower four.

This channel also provides a volume shift register. This register specifies the amount by which the samples are shifted to the right before being played back (essentially controlling volume).

The table given below matches the bit pattern stored in that register to the actual shift amount.

Bit pattern	Shift Amount
0	4
1	0
2	1
3	2

This channel as all other channels does have a length function, but no envelope and sweep.

Channel Four

Channel four is configured to output white noise with envelope. It has two new registers, the LFSR (Linear Feedback Shift Register) and the polynomial counter. The LFSR is 15 bits long.

The polynomial register is the control centre of this channel. The following parameters are supplied by the register,

1. The amount the divisor need to be shifted (bits 7-4)
2. Width of the counter (bit 3)
3. The base divisor code (bits 2-0)

The frequency timer of this channel is calculated a bit differently,

```
Frequency Timer = Divisor << Shift Amount
```

The divisor code is mapped to the actual divisor as follows,

Divisor code	Divisor
0	8
1	16
2	32
3	48
4	64
5	80
6	96
7	112

The second parameter controls the “regularity” of the channel, as if enabled the output of the channel becomes more regular and resembles a tone.

Whenever the frequency timer expires the following operations take place,

1. The frequency timer is reloaded using the above formula.
2. The XOR result of the 0th and 1st bit of LFSR is computed.
3. The LFSR is shifted right by one bit and the above XOR result is stored in bit 14.
4. If the width mode bit is set, the XOR result is also stored in bit 6.

In pseudo-code,

```
if frequency_timer == 0 {
    frequency_timer = (divisor_code > 0 ? (divisor_code << 4) : 8) << shift_amount

    xor_result = (LFSR & 0b01) ^ ((LFSR & 0b10) >> 1)
    LFSR = (LFSR >> 1) | (xor_result << 14)

    if width_mode == 1 {
        LFSR &= !(1 << 6)
        LFSR |= xor_result << 6
    }
}
```

The amplitude of the channel is simply the bit 0 of LFSR inverted. (Take into account envelope of-course).

```
Amplitude = ~LFSR & 0x01;
```

On a trigger event, all the bits of LFSR are set to 1.

Simple Mixing and Panning

The NR51 specifies where the Game Boy sound channels are panned.

In pseudo-code, panning could look like this,

```
// Assume that we are working with the Left stereo channel.

// NR51 bit 5 controls the panning of channel 2 to left channel.
if (nr51 & 0x20) != 0 {
    amplitude = channel_two_amp
} else {
    amplitude = 0
}
```

While simple mixing is just averaging the amplitudes (panning comes before mixing).

```
Left Sample = (ch1 + ch2 + ch3 + ch4) / 4.0
```

NR52

The NR52 can be thought of as the master switch.

If the bit 7 of NR52 is reset, then all of the sound system is immediately shut off. This has the benefit that if a game doesn't support audio, then switching off the audio system saves power. The bottom four bits of the register are the individual channel status bits, and are read only. They describe whether the respective channel is enabled or disabled.

Providing Samples to the Audio Device

Lets assume that you maintain a buffer of size 1024 which contains 512 samples of stereo PCM data. (32-bit float PCM). These samples are interleaved in the audio buffer as,

```
LR, LR, LR, ...
```

Suppose that the sample rate of the audio device is 48000 Hz (48kHz). It means that 48000 samples will be pushed to the audio device every second. Each CPU FREQUENCY / SAMPLE RATE T-cycles (roughly 87 T-cycles in this case) we get the amplitude from each channel and mix them (see above section).

We then use the volume values in NR50 to generate two samples,

```
Left Sample = Left Volume (in NR50) * Left Amplitude
Right Sample = Right Volume (in NR50) * Right Amplitude
```

Those samples are then interleaved in the audio buffer as described earlier.

We only run the emulator until the audio buffer is full. We then delay until the already queued samples are below a particular threshold (let us say 1 buffer in length) queue these newly generated samples and clear our audio buffer. We repeat these steps again and again.

This provides a natural way to accurately synchronize the emulation, while minimizing audio cracks and pops. (But this might increase screen tearing as we disable V-Sync).

In The End

I hope this article was helpful to you in building your emulator.

I would like to thank the people over at r/EmuDev and the Discord server for their help, especially Simuuz and Dillon.

I would also like to thank you for reading this article.

Cheers!

Further Reading

Related to Sound on the Game Boy

1. gbdev Wiki
2. GBSOUND.txt
3. GhostSonic's post on Reddit

General Documentation of the Game Boy

1. GBEDG

2. Pan Docs

Other Emulators

1. Argentum GB (My own emulator, not very accurate)
2. CryBoy (Matthew Berry's emulator, bit more accurate)
3. SameBoy (LIJI's emulator, the most accurate emulator around)

You can look at the source code of these emulators as a reference.

Footnotes

1. The Game Boy only has hardware that can playback mono sound, but you can plug in a headphone for stereo. ↩

2. The envelope function described further ahead in the post complicates it a little bit. ↩