Synth 101

Homework II: Analysis of a Synth Juce Plugin

For the second homework the Synth 101 group has been asked to analyze a Synth Juce Plugin. We've chosen to take a look at an emulator of the Nintendo Gameboy audio hardware called PAPU. This plugin basically consists of two combined square waves with the addition of a noise. The three signals can be altered through different knobs and parameters which are going to be discussed in a few lines.
The plugin exploits two different libraries in its implementation, the first is developed by the plugin programmer himself and it's called gin. This library is based on Juce objects and its role is to help from the GUI management and the parameters settings. The second one, Gb_apu, is a gameboy sound chip emulator based on the use of a three channel buffer called blip buffer. We are going to discuss better the libraries after in this report
The report has been subdivided in different sections:

- Description of the GUI
- Plugin structure and sound generation
- An example of processed sound
- Relationship between audio and GUI

1 - Description of the GUI

The Graphical User Interface is composed of four main sections: the first three of them contain the controls for each single oscillator, while the fourth one contains the output triggered scope, present on the right. Each of the first three sections comprehend two on/off buttons that decide whether to send the signal to the left and right channels and several knobs whose purpose is to control the output waveshape, all referring to the addressed oscillator.

The first square oscillator presents the following parameters:

- **PW:** duty cycle of the squarewave
- **Attack:** sets the time it takes for the signal to rise
- **Release:** sets the time it takes for the sound to decay when the key is released
- **Tune:** shift in semitones on played sounds
- **Fine:** detune effect on the played sound
- **Sweep:** factor that regulates the time interval between when changing frequency during the" sweep effect"
- **Shift:** factor that regulates the frequency interval in the "sweep effect"

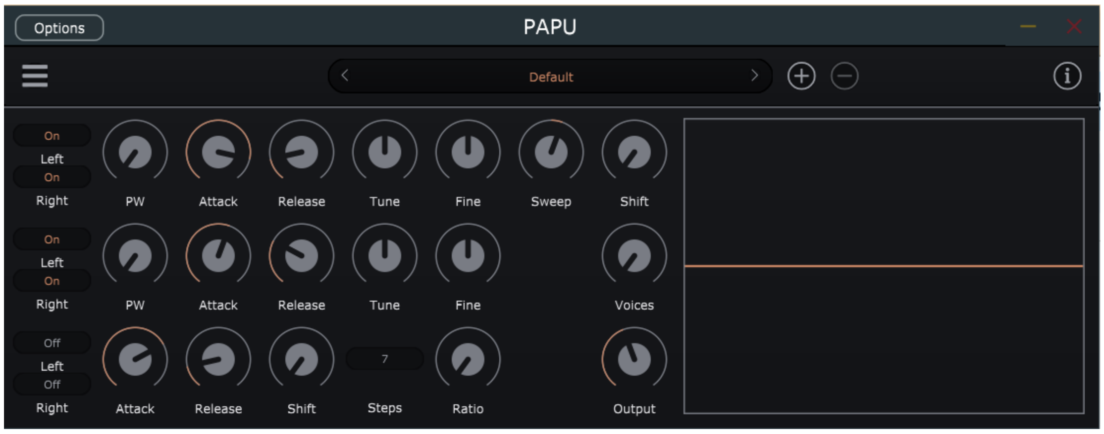
For the second square oscillator, the same thing stands as above, except for the "sweep" and "shift" that are not present here.

For the noise generation instead:

- **Attack:** sets the time it takes for the signal to rise
- **Release:** sets the time it takes for the sound to decay when the key is released
- **Shift:** the different sound waves are also shifted in time when superimposed, this parameter controls it.
- **Steps:** the generation of the noise is an “addictive one” in the sense that a single noise signal is generated. The steps parameter indicates how many times this wave is superimposed.
- **Ratio:** is basically the amount of amplification of the noise signal compared to the square ones.

There are some additional parameters that instead affect every synth:

- **Voices:** number of keys that can be pressed at the same time
- **Output:** global output volume



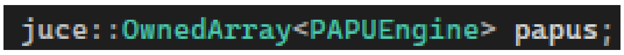
2 - Plugin structure and sound generation

In this section we will address several essential libraries and functions of this plug-in that we believed to be important. We are going to start by discussing the library *gb_apu*, an introduction to this is essential to understand the main features of the plugin, because it is there that the actual sound synthesis happens. One of the things this library makes possible is for the user to set some parameters for the generation of sound. Another relevant implementation is the so-called *blip buffer* that basically consists of three channels: two for the square signals and one for the noise. Once the buffer is filled it’s obviously possible to read it and to pass the samples to an out buffer.

To better understand the mode of operation of this library it is also necessary to briefly introduce the *Gb_apu* class. This class is a sort of container for the whole library, it is here that the square signals and the noise are sent to the blip buffer. Furthermore in this class are implemented the methods that allow to set the overall volume and equalization of the emulator.

The actual shape of the square and noise functions is set instead in another file, still contained in *gb_apu*, which is *Gb_oscs*. Here we can find different structures (structs) such as *Gb_square* or *Gb_noise* dedicated respectively to the generation of the square and the noise signals. It’s important to notice that these are not classes but structures, so basically they are collections of data, useful to unify parameters related to a single waveform.

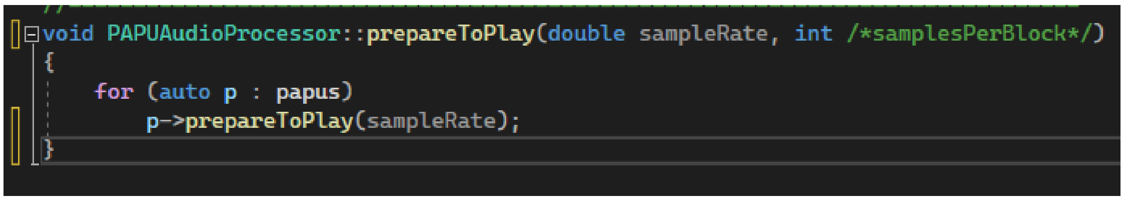
Another important role is played by the classes contained in the *pluginProcessor* file of the juce project. Here we can find the *PAPUEngine* and *PAPUAudioProcessor* classes, who both have a quite common juce class implementation, including the *prepareToPlay()*, *processBlock()* and *releaseResources()* methods. *PAPUAudioProcessor* in particular inherits from the *juce::AudioProcessor* class, so we can consider it as a “classic” juce class. The *PAPUEngine* one is instead a base class but it’s always called inside the processor. An example of this could be found in the image shown below, where we can see that the class *PAPUAudioProcessor* contains an owned array (an array of pointers) to some specific instances of the class *PAPUEngine*.



```
juce::OwnedArray<PAPUEngine> papus;
```

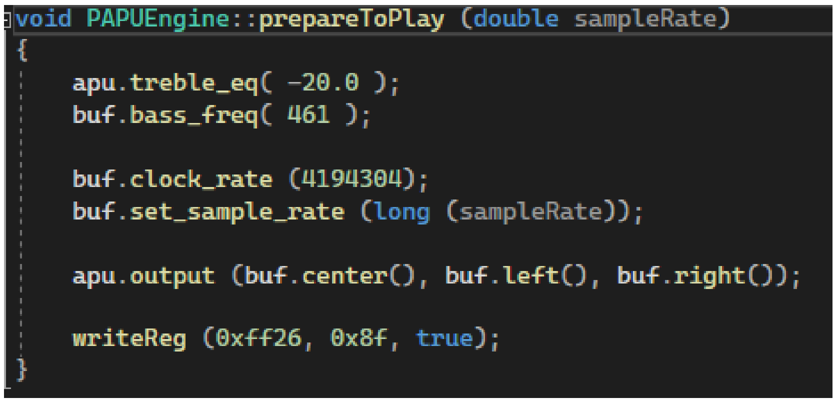
This array is strictly linked to the *voices* parameter present in the GUI that makes the user able to set how many keys are concurrently playable. Indeed we can see in the *processBlock* method that this parameter is used to iterate on the *papus* array.

When addressing the *prepareToPlay()* method of the *PAPUAudioProcessor* class, we can see that for every *PAPUEngine* instance is executed its *prepareToPlay()*.



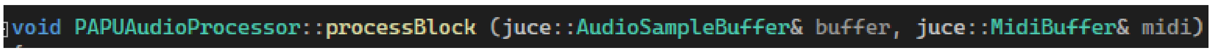
```
void PAPUAudioProcessor::prepareToPlay(double sampleRate, int /*samplesPerBlock*/)
{
    for (auto p : papus)
        p->prepareToPlay(sampleRate);
}
```

It’s important to notice that the arrow is used instead of the point, just because *p* is not an actual object but a pointer to it.
As we can see in the image below the *PAPUEngine::prepareToPlay* starts setting some parameters that come from the *Gb_apu* library, for example the sample rate of the blip buffer or the channel allocation of the output. In any case the parameter *buf* refers to the blip buffer, and the *apu* one is an instance of the *Gb_Apu* class previously described.



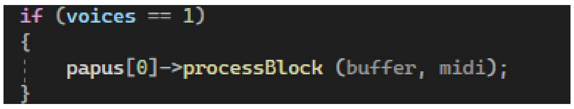
```
void PAPUEngine::prepareToPlay (double sampleRate)
{
    apu.treble_eq( -20.0 );
    buf.bass_freq( 461 );

    buf.clock_rate (4194304);
    buf.set_sample_rate (long (sampleRate));

    apu.output (buf.center(), buf.left(), buf.right());

    writeReg (0xff26, 0x8f, true);
}
```

Another two important methods of the PluginProcessor.cpp file are the two process blocks: the *PAPUAudioProcessor* one and *PAPUEngine*’s one**.** The first is the cornerstone of the plugin itself, in fact it receives the midi messages from the *juce::MidiBuffer* and it executes different methods to deal with the contents of it.



```
void PAPUAudioProcessor::processBlock (juce::AudioSampleBuffer& buffer, juce::MidiBuffer& midi)
```

This is a quite dense function, as we said before the parameter voice is very important here, due to the fact that it rules the main condition of this part of the code. As we can see here, if there's only one voice the process block of *PAPUEngine* class is going to be executed.
The processBlocks of the two classes are quite similar, but they are developed to work in different conditions. As we can see the *PAPUEngine* one is executed in the case described

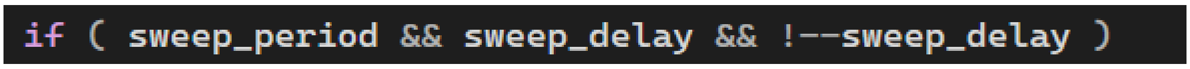


```
if (voices == 1)
{
    papus[0]->processBlock (buffer, midi);
}
```

in the upper image and it contains two main functions: the runOscs one and the runUntil one. The first one is the function that passes the parameters (encoding the midi messages that is passed as the parameter midi) to the apu library that creates the waveforms and the second one reads on the blip buffer and writes on the *juce::AudioSampleBuffer* (buffer parameter).
The procedure is different if instead we have more than one voice. In fact the output AudioBuffer cannot be filled directly in the *PAPUEngine* class that represents, in a very abstract way, a single key pressed, but it has to be filled with the contributions of each of the notes played by the user. So in this case first of all for each element of the *papus* array is executed a method called prepareBlock, that sets the different parameters for every oscillator but without writing on the output juce buffer. The reading of the blip buffer takes place after and not in the single *PAPUEngine* but in the *PAPUAudioProcessor*, in a method also this time called "runUntil".
In the *PAPUAudioProcessor::processBlock* is then also a mono fifoBus filled that reads from the *AudioSampleBuffer* in output and transforms it into a mono, adding the left and the right channels and dividing them for two.
This buffer is used for the signal visualization on the scope that is present in the GUI.

3 - An example of processed sound

For reasons of space, we cannot describe how all the different parameters of the GUI impact on the output sound, so we've decided to stick to a representative example that involves one of the most recognisable effects of this plugin, which is the "sweep".
In a nutshell, this effect causes the frequency of a given note to "sweep" towards higher or lower frequencies, generating one of the classic gameboy sounds. The "sweep" and "shift" knobs regulate the way in which the sweep happens, as we can see inside *Gb_Square::clock_sweep()* in the file *Gb_Oscs.cpp*. Each execution of this method firstly triggers the following if-statement:

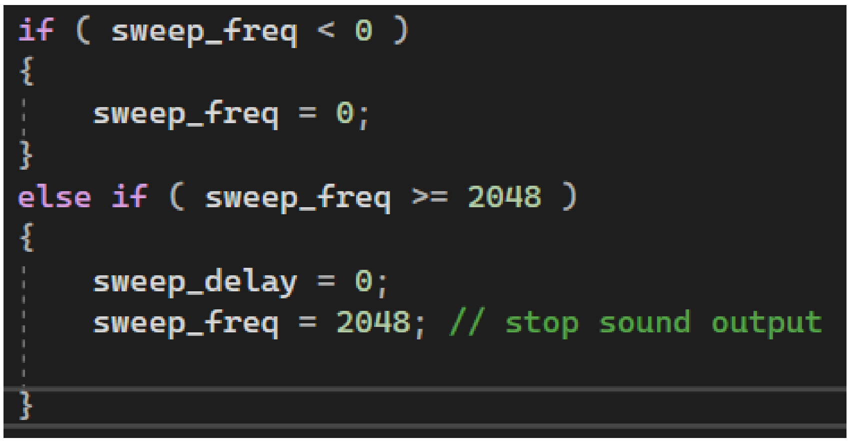


```
if ( sweep_period && sweep_delay && !--sweep_delay )
```

This means that in order to enter in the section of the code related to the sweep, both *sweep_period* and *sweep_delay* have to be different from 0, furthermore the current value of *sweep_delay* after being pre-decremented should be 0, because of the NOT operator. If any of these conditions is not matched we simply skip to the end of the method.
In order to understand the following section it is important to introduce some different parameters, which all have a determined value dependent on knobs position and/or on the frequency of the note currently being played:

- *sweep_period*, which is the absolute value of the *sweep* knob (3 bit)
- *sweep_dir*, the sign of the *sweep* knob parameter, 0 if positive, 1 if negative (1 bit)
- *sweep_shift,* that is the value of the *shift* knob (3 bit)
- *sweep_freq,* representing the frequency at which we want to play the sound (11 bit)

In case the if statement presented above is true we proceed to set *sweep_delay* = *sweep_period* and *frequency* = *sweep_freq*, effectively changing the current frequency. A new parameter called *offset*, which corresponds to the change in frequency that we're going to apply to our current frequency, is then defined by right-shifting the value *sweep_freq* of *sweep_shift* positions. This makes *offset* a variable with a dimension that can go from 11 to (11 - 7) = 4 bits, since the maximum value of *sweep_shift* is 7. We then check if *sweep_dir* is different from 0, or more simply equal to 1, case in which we have to implement a negative sweep and therefore have to change the sign of offset with the command *offset* = *-offset.*
In line 143 *sweep_freq* += *offset,* finally realizes the aforementioned change in frequency, in particular, if offset is positive we hear an upwards sweep, while if it is negative we hear a sweep towards lower frequencies.
It's relevant to underline that *sweep_shift* has the essential role of establishing how fast *sweep_freq* is going to increase or decrease, higher values of *sweep_shift* will cause a lower *offset* value, and will therefore cause the sweep to be much slower and to involve a lot of frequencies. On the other hand lower values of *sweep_shift* will make the variable *offset* to be bigger, therefore making the sweep sound much faster as it involves less frequencies.
In the next lines of code, the end point of the sweep is determined:



```
if ( sweep_freq < 0 )
{
    sweep_freq = 0;
}
else if ( sweep_freq >= 2048 )
{
    sweep_delay = 0;
    sweep_freq = 2048; // stop sound output

}
```

In case the sweep is negative it means that *sweep_freq* is currently decreasing, as a result in order not to have sound glitches we have to check if it becomes lower than 0. If it does, we simply set its value to 0, making the frequency steady and stopping the sweep effect for the current note. In case of positive sweep instead in order to stop the sound we check whether *sweep_freq* is greater or equal to 2048, if it is we set *sweep_delay* to 0 (therefore in the next execution, the first if statement we encounter will be false and we will skip the *clock_sweep* code). Another crucial aspect in *clock_sweep()* method regards the line of code:

period = (2048 - *frequency*) * 4

which basically will make lower frequencies last longer with respect to higher frequencies during the sweep.

4 - Relationship between audio and GUI

This section will go over the methods used to make data coming from the interactive elements of the GUI be consistent to what we hear. The communication between the GUI and the audio implementation of the plugin is deeply rooted in a smart data storing and data management system. In detail SOCALABS decided to exploit registers, creating an architecture capable of transferring short and relevant information efficiently through the various cpp files.
The first register used in the program is the one at address “start_addr = 65296;”, which is equivalent to the hexadecimal number 0xff10. In the pluginProcessor.cpp we use registers going from 0xff10 to 0xff26 and, in particular, registers going from 0xff10 to 0xff23 are dedicated to the storage of data of the three oscillators, while register 0xff24 is dedicated to the storage of the value of the *output* knob (which represents the output volume) and registers 0xff25 and 0xff26 are used to determine if the three processed signals are meant to be stereo, mono or muted. This means that we have 20 registers (0xff24-0xff10) dedicated to the oscillator's data and 3 registers (0xff27-0xff24) dedicated to hardware output management.
so, if we can consider 0 as the first register and forget their hexadecimal nature without loss of information we get the following representation:

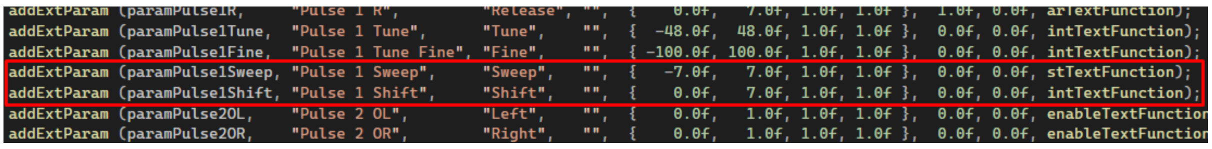
- registers going from 0 to 4 are dedicated to the first square wave
- registers going from 6 to 9 are dedicated to the second square wave
- registers going from 17 to 19 are dedicated to the noise channel.

Now that we introduced the registers, let’s address the chain of orders for this section of the code, which is the following:

PAPUEngine::runOscs → *PAPUEngine::writeReg* → *Gb_Apu::write_register*→ → the relative *Gb_Osc::write_register* with respect to what oscillator we are referring to

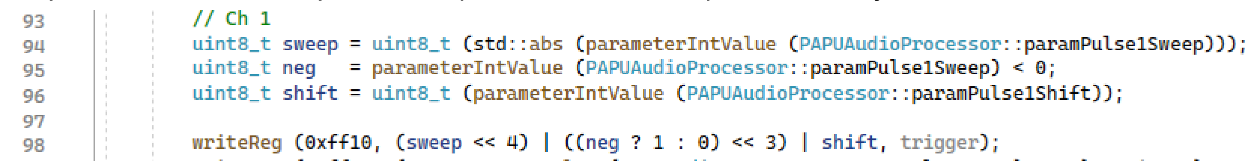
In order to explain this, let’s follow the process that makes the settings of the sweep and shift knobs reflect on the system output audio. All the knobs of the application have different range values which specify the behavior of their relative parameter, we can see how this works in *PAPUAudioProcessor::PAPUAudioProcessor()*, inside *PluginProcessor.cpp*.

Let's analyze for instance what happens when the user switches the knob from -54.7 ms to -46.9 ms. By taking a closer look at the swift knob we can see that it has 15 positions, going from -7 to +7, each one of these corresponds to a precise value in ms that can be seen in the *stTextFunction* function, in particular if we inspect this function we notice that -54.7 ms is related to the position -7, while -46.9 ms instead is linked to the position -6. When the knob gets rotated to the new value basically what is going on is a change in the *addExtParam* function from -7 to -6, which is then eventually reflected into an update on the *stTextFunction* string that the user sees on screen.



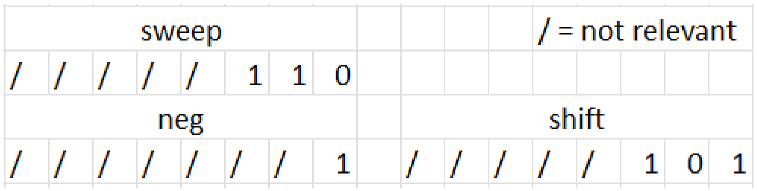
```
addExtParam (paramPulse1R,      "Pulse 1 R",          "Release", "",  {    0.0f,    7.0f, 1.0f, 1.0f },  1.0f, 0.0f, arTextFunction);
addExtParam (paramPulse1Tune,   "Pulse 1 Tune",       "Tune",    "",  {  -48.0f,   48.0f, 1.0f, 1.0f },  0.0f, 0.0f, intTextFunction);
addExtParam (paramPulse1Fine,   "Pulse 1 Tune Fine",  "Fine",    "",  { -100.0f,  100.0f, 1.0f, 1.0f },  0.0f, 0.0f, intTextFunction);
addExtParam (paramPulse1Sweep,  "Pulse 1 Sweep",      "Sweep",   "",  {   -7.0f,    7.0f, 1.0f, 1.0f },  0.0f, 0.0f, stTextFunction);
addExtParam (paramPulse1Shift,  "Pulse 1 Shift",      "Shift",   "",  {    0.0f,    7.0f, 1.0f, 1.0f },  0.0f, 0.0f, intTextFunction);
addExtParam (paramPulse2OL,     "Pulse 2 OL",         "Left",    "",  {    0.0f,    1.0f, 1.0f, 1.0f },  0.0f, 0.0f, enableTextFunctio
addExtParam (paramPulse2OR,     "Pulse 2 OR",         "Right",   "",  {    0.0f,    1.0f, 1.0f, 1.0f },  0.0f, 0.0f, enableTextFunctio
```

This switch of parameter value comes into play on the *PAPUEngine::runOscs* function, in line 94 the code calls *PAPUAudioProcessor::paramPulse1Sweep*, which purpose is to get the position value of the parameter paramPulse1Sweep, discussed just above.



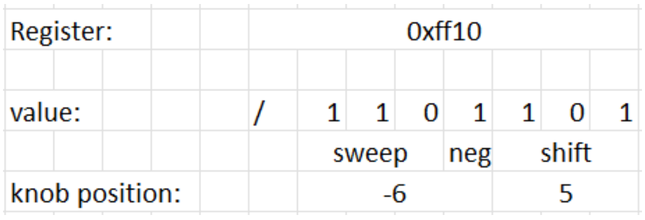
```
93        // Ch 1
94        uint8_t sweep = uint8_t (std::abs (parameterIntValue (PAPUAudioProcessor::paramPulse1Sweep)));
95        uint8_t neg   = parameterIntValue (PAPUAudioProcessor::paramPulse1Sweep) < 0;
96        uint8_t shift = uint8_t (parameterIntValue (PAPUAudioProcessor::paramPulse1Shift));
97
98        writeReg (0xff10, (sweep << 4) | ((neg ? 1 : 0) << 3) | shift, trigger);
```

After we get the value "-6.0f", we cast it as integer and proceed to take the absolute value, transforming it into "6", we then change from int to a digits binary representation by calling *uint8_t*. Finally the result is saved into the variable sweep. In line 95 we follow a similar process, saving the sign of the *paramPulse1Sweep* instead, justifying the use of the abs value of before. In line 96 instead we save the value of the shift knob, which has 8 possible positions and so it can be represented by only 3 bits.



sweep											/ = not relevant				
/	/	/	/	/	1	1	0								
neg								shift							
/	/	/	/	/	/	/	1	/	/	/	/	/	1	0	1

In line 98 we finally write on the register 0xff10 all the values by implementing a left shift rotation of the bits of the variable sweep and neg, compressing all the information in just one byte.



Register:	0xff10							
value:	/	1	1	0	1	1	0	1
		sweep			neg	shift		
knob position:		-6				5		

The function write_register that is called is PAPUEngine::writeReg, that calls the writeReg function inside Gb_Apu.cpp, which receives the register and its data and differentiates between parameter regulating registers (below 0xff24) and final output regulating registers (after 0xff23). Our sweep register is 0xff10, so it falls under the first category.
It is important in order to understand the following lines of code to add that there is an array called *oscs* containing the references of the different variables containing the oscillators.

```
29    oscs [0] = &square1;
30    oscs [1] = &square2;
31    oscs [2] = &wave;
32    oscs [3] = &noise;
```

It is also important to remember the decimal register subdivision explained at the beginning of the chapter, which correlates registers going from 0 to 4 to the first square wave. Now that we have made these premises we can address the following section of the code, still inside Gb_Apu::write_register:

```
165    int reg = addr - start_addr;
```

```
173    if ( addr < 0xff24 )
174    {
175        // oscillator
176        int index = reg / 5;
177        oscs [index]->write_register( reg - index * 5, data );
178    }
```

The code here is basically sorting between different parameter regulating registers, at line 165 we're setting the variable *reg* as a decimal number corresponding to the register numbers from 0 to 19 just mentioned above. After we've done that, in line 176 our *reg* variable gets divided by 5, forgetting the remainder of the division, and gets saved on the *index* variable. Therefore inside *index* we have 0 if we're referring to the first square wave, 1 if we're referring to the second one and 3 if we're addressing the noise channel instead, reflecting exactly the references contained in the *oscs* array. In line 177 we exploit this relationship and call write_register on the right oscillator, present in *oscs [index]*. The function *write_register* called in this line sends to the file *Gb_Oscs*, in which are present different implementations of this function based on the type of wave being addressed. To call the right write_register function we can exploit the arrow operator, in our case the *write_register* will be the one of the square wave. When making this call we pass the value (reg - index * 5), which is a number going from 0 to 4 representing the 5 different registers dedicated to the signal processing features of each single oscillator. In detail

- number 0 corresponds to the sweep/shift feature
- number 1 is represents the duty cycle knob
- number 2 is dedicated to the "attack" of the signal
- number 3 and 4 control the frequency of the signal (this comprehends semitone changes, hundreds of semitone changes, or updates on the value of the pitch wheel)

Everything can be clearer by looking directly at *Gb_Square::write_register* in the file Gb_Oscs.cpp. The function is divided in different *cases*, going from 0 to 5, which all correspond to the just mentioned effects. In our case we will fall in the case 0, since
(reg - index * 5) = 0 - 0 * 5 = 0. We then go ahead and decompress the information contained in the value of the register by right shifting some bits and by saving them into proper separate variables. We successfully transferred the GUI information of the sweep and of the shift knobs to our synthesizers.

```
163    case 0:
164        sweep_period = (value >> 4) & 7; // changed
165        sweep_shift = value & 7;
166        sweep_dir = value & 0x08;
167        break;
```