Synth 101 - Analysis of the JUCE Plugin PAPU

Federico Vescovi, CP:10935407, Matr n°:220651

Pio Francesco Calogero, CP:10939480, Matr n°:226226

Marco Pelazza, CP:10746821, Matr n°:232343

For the second homework the Synth 101 group has been asked to analyze a Synth Juce Plugin and we've chosen to take a look at an emulator of the Nintendo Gameboy audio hardware called PAPU. The plugin consists of two square waves and one noise channel connected in parallel. 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 in GUI management and in handling 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*.
The report has been subdivided in different sections:

- **Description of the GUI**
- **GUI block chain**
- **Audio block chain**
- **An example of processed sound**
- **Extra**

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 comprehends two on/off buttons that decide whether to send the signal to the left and right channels and to 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:** regulates the 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 a key is released
- **Tune:** represents a shift in semitones on played sounds
- **Fine:** controls a detune effect on the played sound
- **Sweep:** regulates the duration of notes during the "sweep" effect, this will be clearer in chapter 3.2 and 4

- **Shift:** regulates the frequency interval between consecutive notes in the “sweep” effect, this will be clearer in chapter 3.2 and 4

For the second square oscillator, the same thing stands as above, except for the “sweep” and “shift” knobs, that are not present here.
For the noise generation instead the possible settings are the following:

- **Attack:** sets the time it takes for the signal to rise
- **Release:** sets the time it takes for the sound to decay when a key is released
- **Steps:** indicates how many identical noise waves are superimposed. The generation of the noise is in fact an “additive one”, in the sense that a single noise signal is generated and then it is summed to other identical replicas
- **Shift:** controls the time shift between each individual noise wave mentioned above
- **Ratio:** controls the overall amount of amplification of the noise signal

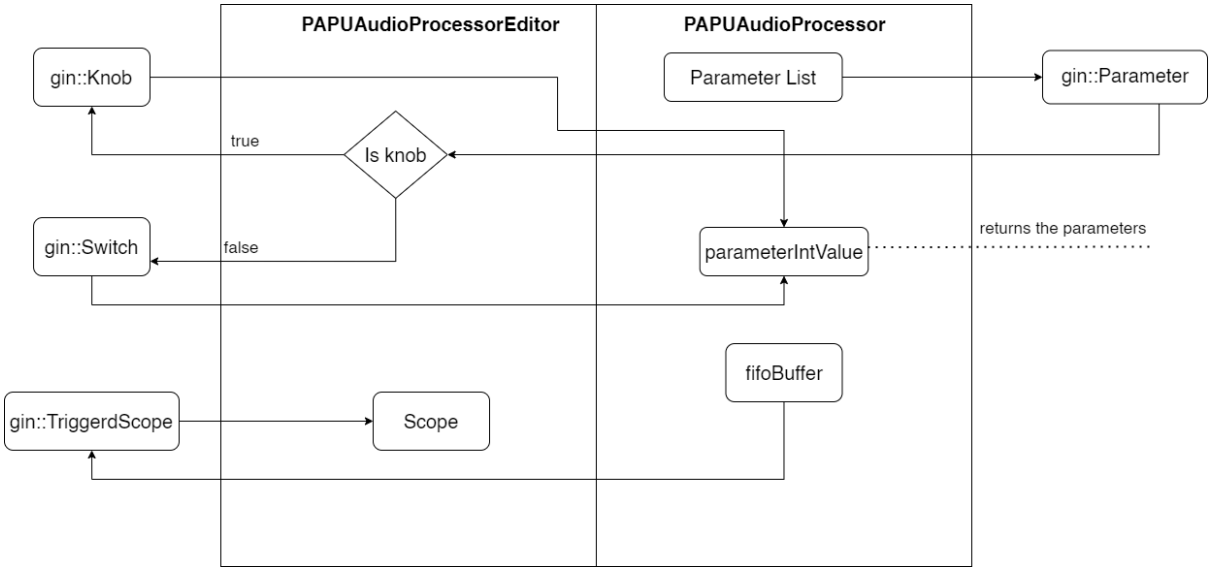
There are some additional parameters that instead affect every synth:

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

2 - GUI block chain

In this section we will address from a general point of view the sections of the code that are in charge of implementing the Graphical User Interface. On the next page you can find a diagram representing how the different code blocks interoperate. The GUI is created in the PAPUAudioProcessorEditor by an iteration on the *Parameter List* declared instead in the PAPUAudioProcessor. This list is set in such a way that each item contains the information to instantiate the class *Parameter,* implemented in the gin library. This class has methods that return its possible values and how it has to be represented in the GUI.
Thanks to the information retrieved by *Parameter*, a knob or a switch are created by using two classes, Knob and Switch, coming once again from gin. These classes inherit directly from the corresponding JUCE classes but have some extra features. The instant values of the parameters are then constantly read and returned by parameterIntValue, a method of the class Processor (parent of the PAPUAudioProcessor class).
The GUI also presents a time scope of the current generated signal, realized by the gin::TriggeredScope class. What the class does is basically to read from a mono FIFO

buffer, which has already been filled by reading from the stereo output buffer in the processBlock method of the PAPUAudioProcessor class.

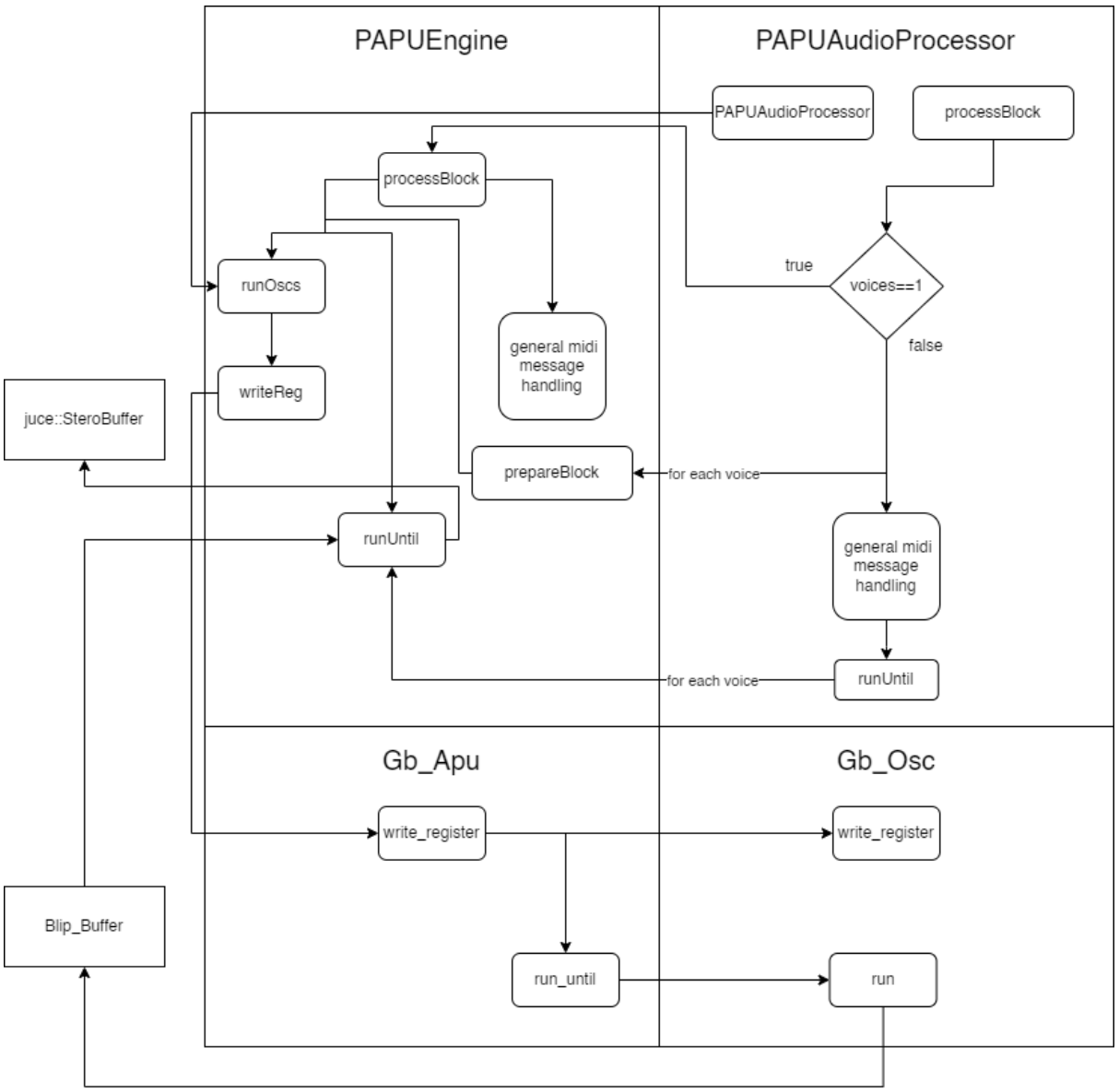


3 - Audio block chain

Now let’s move to the sound generation part of the plugin. In the next page you can find the overall audio block diagram of the plugin, which as you can see is subdivided in 4 classes and a structure: PAPUAudioProcsssor, PAPUEngine, Gb_Apu, Blip_Buffer and Gb_Osc. There are another bunch of secondary files**,** but they are less relevant and we felt that by adding them we would have made the diagram much less clear to comprehend.
Let's give a general description of what these sections of code do:

- PAPUAudioProcessor is responsible for general sound management, in particular when we have a multi voices kind of situation. Here we can find the standard JUCE class methods such as processBlock and prepareToPlay.
- PAPUEngine is instead in charge of single MIDI note information handling and of making various calls to the methods that make sure the current sound matches with the GUI parameters.
- Gb_Apu is a class, from the homonymous library, that covers more general audio aspects like overall output settings. Gb_Apu also acts as a bridge between PAPUEngine and Gb_Osc.
- Blip_Buffer is a class included in the library Gb_Apu. It is quite particular due to the fact that it has three channels, two for the standard stereo and one for the mono. The waveform samples are loaded in this buffer, which is then read from the runUntil method of PAPUEngine that uploads the samples in a JUCE StereoBuffer.
- Gb_Osc is a structure that handles the specific sound parameters for each of the possible oscillators in order for the sound to be consistent with what is shown in the GUI. This is also where the square and noise signals get generated, so we can say that it’s the core of the audio generation.

The plugin has a pretty complex structure, therefore will be able to cover only the blocks that we considered to be the most interesting.



3.1 - Voices and MIDI handling

One of the most important parameters that can be set by the user is *"voices"*. The value of it ranges from 1 to 8 and represents the number of keys that the user is allowed to play at the same time. This difference is not only enclosed in the player side but also in the implementation, in fact, as we can see in the above scheme, inside the PAPUAudioProcessor class there is a condition about the voices that makes the code flow take a substantially different route when the number is greater than one. In order to understand the reason for such deviation it is important to introduce the Array *papus*, contained inside the *PAPUAudioProcessor* class, which is basically an array of pointers to instances of *PAPUEngine*.

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

A brief clarification on this array is appropriate to better understand its purpose. Even though the highest number of voices in PAPU is 8, to prevent any kind of overflow, *papus* is

initialized with 16 elements, each member of the array, conceptually, represents a single pressed key and therefore a single synthesized note. Going back to the previous condition, let's analyze the case in which the value of voices is equal to one. If this happens, *PAPUEngine::processBlock* is executed, this method receives a MIDI message whenever a key is pressed/released and handles it by adding the new incoming notes to a queue of those that includes the current one to play as its last element. This *processBlock* also invokes *PAPUEngine::runOscs* and *PAPUEngine::runUntil* to respectively write and read from the *Blip_Buffer*.
If *"voices"* is instead greater than one, for each element of the papus array the *PAPUEngine::prepareBlock* is executed, this method corresponds to the *PAPUEngine::processBlock* one but without the MIDI message handling part. In this case the MIDI management is provided in *PAPUAudioProcessor::processBlock*, with the support of two other methods: *findFreeVoice* and *findVoiceForNote.* The first one is used when a noteOn message is received, a case in which it's necessary to find an instance of the *PAPUEngine* class contained in the *papus* array that hasn't a current note assigned. This state is identified by the -1 values set down to the *curentNote* attribute of the class *PAPUEngine*, then when we find such a voice, *PAPUEngine::handleMessage* is invoked and basically proceeds to complete the *prepareBlock* method to play a note. The second one, *findVoiceForNote,* is instead useful when a *noteOff* message is received, situation in which it is necessary to find an instance of *PAPUEngine* that corresponds to that precise note. This task is accomplished again by analyzing the *currentNote* attribute, when the correct note is found its MIDI value is set as -1. Other two messages that can be received are *pitchWheel* and *allNotesOff* and their management is pretty simple since it consists of the execution of the *PAPUEngine::handleMessage* for each papus element.

3.2 - Transferring GUI parameters to the actual sound

The block chain we're going to address in this section is the following:

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

The communication between these methods of the plugin is deeply rooted in a smart data storing and data management system. In detail, SOCALABS have 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 overall 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 can see that the code is subdivided in this way:

- 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.

In order to explain the block chain mentioned before, let's follow an example of the communication process by looking at how the settings of the sweep and shift knobs reflect on the system output audio. One important thing to say is that 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 sweep knob we can see that it has 15 positions, going from -7 to +7, each one corresponding to a precise value in ms that can be seen on screen when turning the knob. In our case we're simply switching from position -7 to position -6. This switch of parameter value comes into play when the *PAPUEngine::runOscs* function calls *PAPUAudioProcessor::paramPulse1Sweep*, asking for the current position value of the parameter paramPulse1Sweep, which for us is -6.

```
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 getting 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. We're going to assume the shift knob to be in position 5.

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, obtaining this kind of representation:

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

The function write_register that is called in this line is *PAPUEngine::writeReg*. This proceeds to call 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. The code then proceeds to identify the synth corresponding to the received register, in our case the first square one, and then to call the relative *Gb_Osc::write_register*. This last method is implemented several times in *Gb_Osc*, once for each kind of signal we can have (square, noise, ...), and it is responsible for deconstructing the information present in the different registers and for properly updating the sound parameters.
When making this call, after having identified that we're on the square signal case, we determine which of the 5 different registers dedicated to the signal processing features of the square oscillator has been modified. In detail:

- number 0 corresponds to the sweep/shift feature
- number 1 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, after semitone, hundreds of semitone, pitch wheel changes have been applied

When looking directly at *Gb_Square::write_register*, we can see that the function is subdivided in different *cases*, which span from 0 to 4, all of which correspond to the aforementioned sound information. In our case we will fall in the case 0, since 0xff10 is the first register for the first squarewave. 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 the proper separate variables. Since Gb_Osc is also the place in which the waveforms are created, we can say that we have successfully transferred the GUI information of the sweep and 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;
```

4 - An example of processed sound

For reasons of space, we cannot describe how all the different parameters of the GUI impact 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 mentioned before, *Gb_Oscs.cpp* is the file responsible for the actualization of the knob settings, one of which is the sweep/shift. This effect is enclosed in *Gb_Square::clock_sweep()*, and has three particular features which are important to mention:

- The duration of each note during the arpeggio is regulated by the value of the "sweep" knob, in particular positive values of this lead to a sweep towards higher frequencies, while for negative values of the knob, the sweep will be towards lower frequencies
- The duration of each note gets progressively shorter toward higher frequencies
- The frequency gap between consecutive notes is determined by the shift knob value.

In particular, it is interesting to dive a little bit into this last point.
The frequency gap, in the code called *offset*, is determined by the following line of code:

int *offset* = *sweep_freq* >> *sweep_shift*;

Where *sweep_freq* is a 11 bit binary variable containing the value of the current frequency being played and *sweep_shift* is instead the value of the shift knob, which spans between 0 to 7. This means that our gap is determined by a right shift of the frequency value and it is a number that can have from 4 to 11 bits. Later in the code such *offset* gets changed in sign if the original value of the sweep knob was negative. Finally, *offset* is summed to the current frequency, as we iterate this for every single successive sound sample we achieve a proper sound sweep.

4 - Extra

We felt free to add a little extra and replicate the first square oscillator in MATLAB, this came useful in preparing the presentation as this plot is more manageable that the little PAPU onscreen plot. This is both because of the higher resolution of the plot and because of the possibility of adding on screen notations, you can see the final plot below. The period is obviously depending on the frequency of the played note, while *PW* is instead the parameter of the GUI mentioned before, which regulates the duty cycle. The wave is basically a repetition of two exponentials in base 2. More on this can be found in the MATLAB file.