# On JUCE side:

To work with the Vocodec code as described below, you’ll need to update the presetKnobValues array appropriately from JUCE.

For things to be automatable in DAW, you could use JUCE’s AudioParameterFloat, AudioParameterInt, etc. classes (<https://docs.juce.com/master/tutorial_audio_parameter.html>). The base class of all these is AudioProcessorParameter. If you make the AudioProcessorEditor or some other high level UI component a AudioProcessorParameter::Listener, you should be able to update (and have the plugin host automatically update during automation) the presetKnobValues array in AudioProcessorParameter::Listener::parameterValueChanged(). I’ve not actually made an automatable plugin like this in JUCE, so I’m not sure this will work; JUCE isn’t that explicit about how it handles automation.

You can also try looking into AudioProcessorValueTreeState, which is newer and might be better. I’m less familiar with it but I think it works similarly and integrating it with Vocodec should be similar to above.

# Of course I’m open to questions and working through it together when you get to this stuff. - Matt

# 

# The Functions:

void noteOn(int key, int velocity)

This function will get called when a MIDI note on happens. The function is implement in sfx.c but you need to call it from the MIDI input in JUCE.

void noteOff(int key, int velocity)

Same as above

void pitchBend(int data)

Same as above -- data is the two 7-bit numbers added together with a bit shift to create a 14 bit number when the MIDI application calls it pitchBend((USB\_message[2]) + (USB\_message[3] << 7));

presetKnobValues[25]

This is an array of numbers from 0.0 to 1.0 that represents the current position of the parameter’s virtual knobs, and is updated when they change. The array is 25 elements long, and some presets only use the first few elements (right now none of them use all of the elements). The reason I say “virtual knobs” is because depending on which “page” of the 5 knobs you are looking at, it will set the presetKnobValues of that knob. For instance, if you are on page 2 of the knobs and you turn the third knob, it will edit presetKnobValues[7]. The array index is 7 because (element 2 + (knob page \* knob page size)). It’s element 2 because it’s zero-based counting. Knob page size is 5 because there are 5 knobs. Knob page is 1 because it’s zero based counting.

displayValues[25]

This is an array of numbers with a variable range. It’s filled by the sfx code, after reading the knob value and scaling it for the functionality. It’s what is printed on the OLED screen when a parameter changes.

buttonActionsSFX[ButtonB][ActionPress] (example)

This is an array of flags for each button, representing their state. When a button is pressed, the ActionPress flag is set to 1. When a button is released, the ActionReleased flag is set to 1. While a button is held down, the ActionContinuousPress flag is set to 1. When a button has been held down for a particular defined length, ActionHoldInstant gets set to 1.

**typedef** **enum** \_VocodecButton

{

*ButtonEdit* = 0,

*ButtonLeft*,

*ButtonRight*,

*ButtonDown*,

*ButtonUp*,

*ButtonA*,

*ButtonB*,

*ButtonC*,

*ButtonD*,

*ButtonE*,

*ExtraMessage*,

*ButtonNil*

} VocodecButton;

**typedef** **enum** \_ButtonAction

{

*ActionPress* = 0,

*ActionRelease*,

*ActionHoldInstant*,

*ActionHoldContinuous*,

*ActionNil*

} ButtonAction;

There is also a buttonActionsUI. It is also set to the same values as the buttonActionsSFX array when the buttons are read.

The reason for the two arrays is so that the sfx.c functions can read the SFX array and mark the flag back to zero when they are handled for audio purposes, and the ui.c functions can read the UI array and set those to zero when they are handled. For instance, sfx.c will pay attention to the C button to implement a freeze effect on the audio, and mark the one in the sfx version of the array as zero once the audio has been frozen, and the ui.c code will right “freeze” to the OLED display when it can, and then set the ui version of the array element to zero.

You’ll need to look at oled.c and implement most of the functions in there to draw to the OLED screen and turn on and off the separate colored LEDs. The OLED screen is white text on black. Oled.c also refers to gfx.c (adafruit’s graphics library, modified by translating it to c and slimming it down), and custom\_fonts.h. The actual “oledDrawFullBuffer” command is the only one that actually talks to the hardware, all the other ones prep the buffer with the proper pixel data. You could either do this by drawing a pixel bitmap (probably preferred, or by replacing these functions with just writing fonts to that screen position. The display font is EuphemiaCAS 8pt (for preset names) and 7pt (for everything else). I’ll put the font in this folder.

For instance, these functions include:

setLED\_A() etc

All the LED functions take a 0 or a 1 to turn them off and on.

OLED\_writePreset();

This function writes a the name of the preset to the first line of the OLED. You can look at the function in oled.c

OLED\_writeTuning();

Writes the name of the current tuning system to the second line of the OLED.

OLED\_writeEditScreen();

Writes the text of the Edit screen to the OLED -- probably need to change the Edit button to be a toggle rather than a continuous hold, since you can’t do that interaction with a mouse.

OLEDwriteString()

OLEDwriteInt();

OLEDwriteFloat();

void OLEDclearLine(OLEDLine line);

void OLEDclear();

Look up what these do in oled.c and make the display show the info necessary.

decrementPage();

Go down one knob page. Changes which parameters the 5 knobs control.

incrementPage();

Go up one knob page

-Jeff