DSP PAW

Guide & Algorithm Examples

Clyne Sullivan

October 8, 2023

Table of Contents

Introduction

This guide is meant to help new DSP PAW users get acquainted with the platform and software. It consists of a brief section on hardware setup and overview followed by a collection of algorithm examples to demonstrate how DSP PAW can be used implement DSP algorithms in C++ and how the user interface can be used to test and analyze an algorithm’s performance.

You should have some familiarity with microcontroller development as far as setting up the necessary software and a general understanding of how peripherals like the ADC and DAC work. Some experience with C++ (or at least C) is also expected.

The examples in this guide are focused on algorithm implementation and testing, and do not cover the theory behind the given algorithms. You have ideally either taken a course on digital signal processing or have enough knowledge in this field to build educational content off of this guide’s examples.

If you have questions, comments, or suggestions about this guide, you can open an issue on the project’s online repository at https://github.com/bitgloo/dsp-paw/issues. The online repository also has additional information on the software and hardware behind DSP PAW.

Setup Instructions

1. Gather required hardware

- **NUCLEO-L476RG development** board
  This is the development board that DSP PAW builds on top of. Other NUCLEO (or Arduino-compatible) boards are not supported at this time, but may receive support in the future.
- **DSP PAW add-on board**
  The custom Arduino-form-factor board with the circuitry necessary for DSP PAW to function. Not available for purchase yet, but all of the design files are available in the online repository.
- **Micro-B USB cable**
  Connects the add-on board to the computer for using the user interface.
- **Mini-B USB cable**
  Necessary for programming the NUCLEO development board, which may already come with this.
- **Female-to-male/female wires, 0.1"-pitch**
  For connecting input and output signals to the 0.1"-pitch header. For this guide’s examples you will need at least one female-to-female wire to connect the signal generator to the signal input.
- **3.5mm-jack cables, headphones, etc.** (optional)
  For connecting input and output signals to the 3.5mm jacks.

You will also need a computer running either Windows or a Linux-based operating system.

2. Download required software

Windows

Download and install the programs listed below. You should ensure that all of these programs are added to your PATH.

- git: https://git-scm.com/downloads
- The arm-none-eabi GCC toolchain:
  https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads
- OpenOCD: https://openocd.org/pages/getting-openocd.html
- make: https://gnuwin32.sourceforge.net/packages/make.htm

Linux

You will need the same programs listed above, though it would make more sense to go through your distribution's package manager to obtain them. On Debian-based systems, the following command should work:

```
sudo apt install git gcc-arm-none-eabi openocd make
```

3. Obtain and compile the microcontroller firmware

Open a terminal or command prompt in the folder or directory where you would like to keep the DSP PAW firmware files.

Use git to download the DSP PAW source package:

```
git clone https://code.bitgloo.com/bitgloo/dsp-paw.git
cd dsp-paw
```

Next, use git to fetch the project's submodules (i.e. third-party dependencies):

```
git submodule update --init --recursive
```

You can now enter the firmware directory and compile the source code:

```
cd firmware
make
```

This should produce a directory named "build" which contains the file "ch.hex". This is the compiled firmware file that will be programmed onto the microcontroller.

4. Program the microcontroller with the DSP PAW firmware

Plug the NUCLEO board into the computer's USB port.

With a terminal or command prompt open in the firmware directory, it should be possible to program the NUCLEO with a single command:

```
openocd -f interface/stlink.cfg -f target/stm32l4x.cfg -c "program
build/ch.hex verify reset exit"
```

Ensure that the command completes successfully.

5. Connecting the DSP PAW add-on board

You should now have a programmed NUCLEO board and an assembled DSP add-on board.

First, set jumper JP5 on the NUCLEO (next to the reset button) to E5V so that the add-on board powers the microcontroller.

Second, remove the SB57 jumper on the back side of the NUCLEO board. This is a jumper resistor which will require desoldering.

Finally, stack the add-on board onto the NUCLEO. The board aligns with the NUCLEO's Arduino header, and should make for an easy fit.

You can now plug a USB cable into the add-on board to power everything up. If set up correctly, the status LED should go white for a couple seconds, followed by a steady green blink.

6. Setting up the IDE

The source code for the IDE is included in the dsp-paw repository, kept inside the *gui* folder.

On Linux, you will need to compile the source code yourself. This requires GCC, make, and development libraries for SDL2.

For Windows, pre-compiled binaries will be made available on the DSP PAW repository website, under the Releases section. Otherwise, you may also compile the source code.

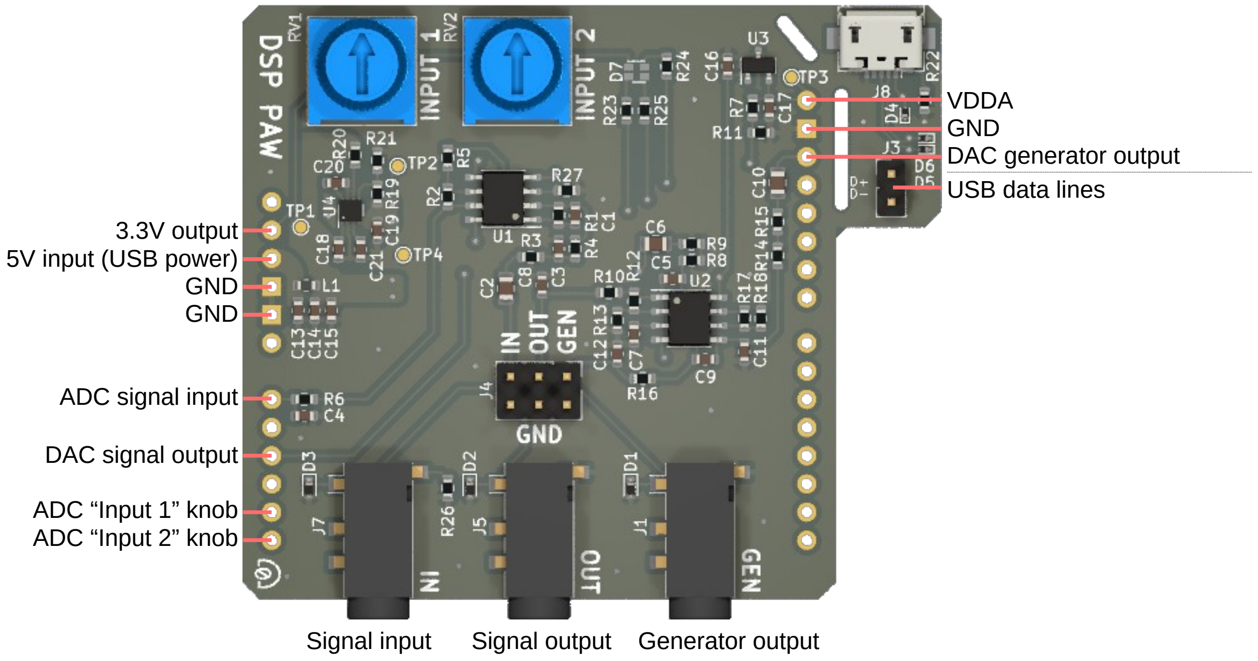
With the prerequisites installed, simply run *make* to compile the code. The result will be an executable named *stmdspgui*.

7. Testing the connection between the hardware and GUI

With the IDE running and the hardware connected to the computer through the add-on board USB port, you can test the USB connection by opening the "Device" menu and choosing "Connect". The GUI's log should say "Connected!" if successful.

If you have issues on Linux, confirm that your user has permissions to access the device (e.g. /dev/ttyACM0).

Hardware overview



Electrical and functional specifications

	Min	Typical	Max	Units
Signal input	-2.0[1]		+2.0[1]	Volts
Signal/generator output	-2.0[1]		+2.0[1]	Volts
VDDA		2.048		Volts
USB input power	3.6	5.0	5.5	Volts
Sampling rate	8	32[2]	96	kHz
Sample buffer size	100	4,096[2]	4,096	count
ADC/DAC resolution		12		bits
Processing latency	1[3]	128[2][3]	512[3]	milliseconds
Algorithm binary size			16,384	bytes
Algorithm stack size	10,000		15,360[5]	bytes

[1] Exceeding this specification may cause damage to the microcontroller.

[2] Default setting after power-on or reset.

[3] Determined by sample buffer size divided by sampling rate.

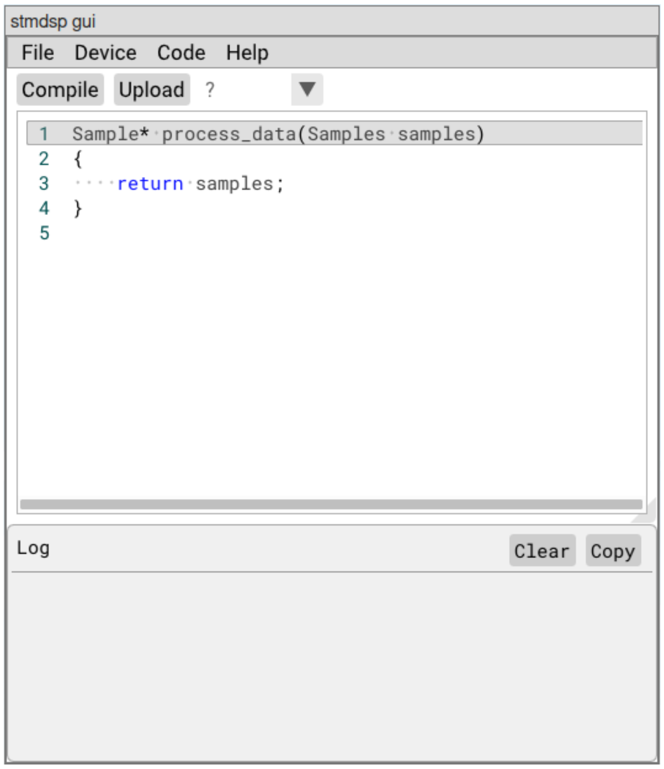
[5] Usable maximum is less than this due to firmware overhead. Exceeding the usable maximum will interrupt algorithm execution.

1. Pass-through and the Signal Generator

To begin working with DSP PAW, you will need a suitable input signal to apply your algorithm to. DSP PAW can supply this through its built-in signal generator, capable of producing signals based on either a list, formula, or audio file input. This example will go over creating input signals with the generator, using the default pass-through algorithm (which *passes* the input *through* to the output) and "Plot over time" feature to verify that the generator is working.

If you will be working with an external signal generator, you may still wish to read through this section to get a feel for how DSP PAW operates.

Overview of the algorithm design process



```
Sample* process_data(Samples samples)
{
    return samples;
}
```

Once you start up the GUI, you will see a window similar to the one pictured to the left. The code pane shows the pass-through algorithm by default.

Algorithms are executed through the `process_data` function, which takes the input sample buffer as an argument and returns a pointer to the output sample buffer. `Samples` is a type alias for an array of `Sample`, where the array size equals the buffer size set by the GUI (default: 4,096). `Sample` is a 16-bit unsigned integer.

The buffer size is made available through a constant named `SIZE`. This constant will be used when we need to iterate through the sample buffer.

Algorithms can be written, saved, and compiled without connecting the GUI to DSP PAW hardware. When you are ready to execute your algorithm on hardware, follow these steps:

1. Connect the hardware to the computer using the USB port on the DSP PAW add-on board.
2. Select "Connect" from the "Device" menu. The log will respond with the message "Connected!".

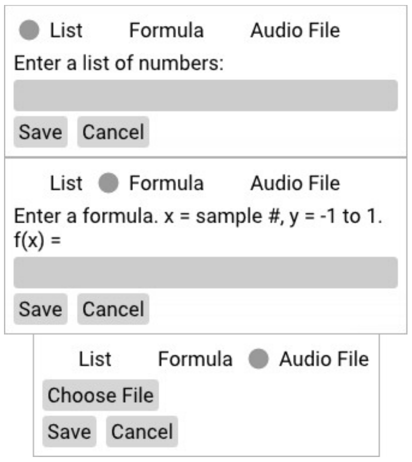
3. Compile your algorithm with the “Compile” button. The log will report if the compilation was successful or if there are errors in the code.
4. Upload your algorithm to the microcontroller using the “Upload” button.
5. Select “Start” from the “Device” menu to begin algorithm execution. The LED indicator on the add-on board will begin flashing blue. At this point, the menu option will change to “Stop” and can be used to stop algorithm execution.

Now that you have a feel for the algorithm design workflow, let’s look over how to prepare the signal generator and how the Plot over time window works so that we can put these features together for the example.

Preparing the signal generator

Configuring the signal generator requires connecting the DSP PAW hardware to the GUI as explained in steps one and two above. Once the hardware is connected, the “Load signal generator” and “Start signal generator” options under the “Device” menu will become available. The generator can be started and stopped independent of algorithm execution; however, the generator can only be loaded with data while no algorithm is running.

Connect your DSP PAW hardware to your computer, then choose the “Load signal generator” option. You will be presented with a small window with three data options: List, Formula, and Audio File.


● List Formula Audio File
Enter a list of numbers:
Save Cancel
List ● Formula Audio File
Enter a formula. x = sample #, y = -1 to 1.
f(x) =
Save Cancel
List Formula ● Audio File
Choose File
Save Cancel

List

“List” allows you to enter in a whitespace-separated list of numbers which will be copied into the generator sample buffer directly. This list can be hand-typed or could be pasted in from an external source like a spreadsheet.

Samples are 12-bit values, so the values in this list must be within the range of zero and 4,095. If there are not enough values to fill the sample buffer (sized at 4,096 samples by default), then the remaining space will be filled with “midpoint” values (2,048 or around 0V).

Formula

“Formula” allows you to enter a formula in the form y = x. Once saved, the formula will be calculated for each value of “x” between 0 and the current sample buffer size. The formula’s output “y” should be in the range of -1 to 1, and will be scaled to the zero to 4,095 range for saving into the sample buffer. For example, a formula of y = sin(x) will produce a sample buffer of a sine wave centered at 2,048 or 0V (at a frequency dependent on your sampling rate).

This feature uses the ExprTk library (https://www.partow.net/programming/exprtk/), which supports many mathematical functions and features.

The GUI also provides a few specialized functions. These are:

- random(L, H): Returns a random number between the range [L, H).
- square(X): Creates a square wave with a period of 2.0.
- triangle(X): Creates a triangle wave with a period of 2.0.
- pulse(L, X): Creates an impulse that returns 1 while X <= L or 0 otherwise.

Audio File

This is an advanced feature which supports streaming audio files to the signal generator while an algorithm is executing. The audio file must be 16-bit WAVE audio (.wav) with a single mono channel. The audio will be streamed to the generator at the sampling rate configured in the GUI, so matching the audio file to that may be desirable.

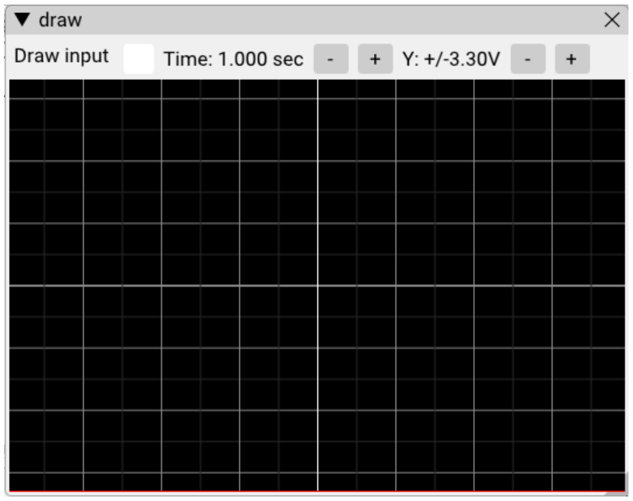
Once the audio has been played through the generator completely, it will loop back to the beginning and play again immediately.

Plot over time

The GUI is capable of visualizing the input and output signal buffers while your algorithm executes. This feature is enable through the "Plot over time" checkbox under the "Device" menu. When the visualizer is opened during algorithm execution, the output sample buffer will be continuously drawn. By checking the "Draw input" box, the input sample buffer can be drawn as a second layer.

The "Time" and "Y" scales can be adjusted to get a better view of the signal(s). You can also hover your mouse cursor over the signals to see their voltages.

The visualizer will retain its capture once algorithm execution is stopped, allowing you to inspect the signal(s) more closely.



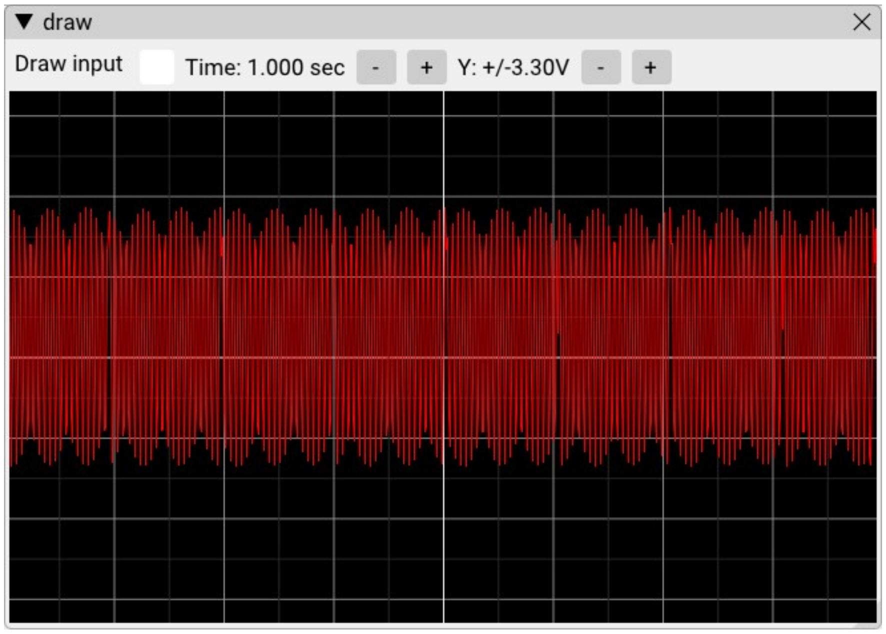
Example: Formula

For now, let's load a formula into the signal generator. The formula `0.5*sin(x/32)` will create a sine wave of an observable frequency with half of the supported amplitude; it will swing in the range of +/-1V.

After loading the formula into the generator, make sure the default algorithm is compiled and uploaded. Finally, open the Plot over time window and start the algorithm execution.

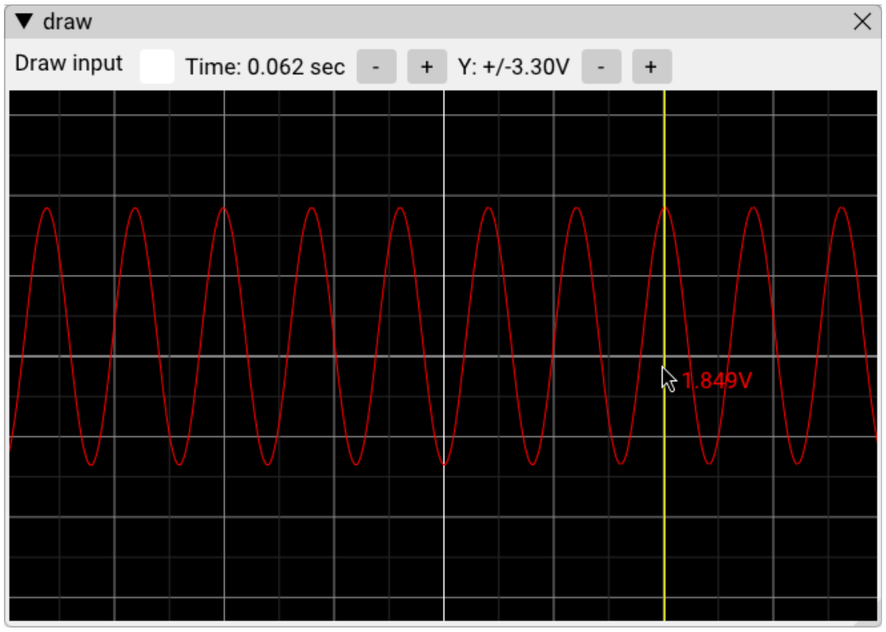
The visualizer should should a flat line that reads around 0V. Now, if you start the signal generator through the Device menu, you should see the sine wave appear. Feel free to move your cursor around to verify the +/- 1V peaks of the signal, or try adjusting the Time and Y scales to get a different look at the output.

Once you're finished, stop the algorithm execution and the generator output through the Device menu (in either order).



Above: Visualization of `0.5*sin(x/32)`.

Below: Zoomed-in capture of signal
(development screenshot: voltage levels are not accurate).



Signal clipping

Note that for List and Formula signals, the sample buffer is only initialized once. If your signal is periodic but does not line up with size of your buffer, you will see clips in the generator's output each time it rewinds to the beginning of the buffer.

2. User-controlled attenuation

Now that you are familiar with how to execute an algorithm, use the signal generator, and view the input and output signals, it's time to try writing an algorithm that actually modifies the input signal.

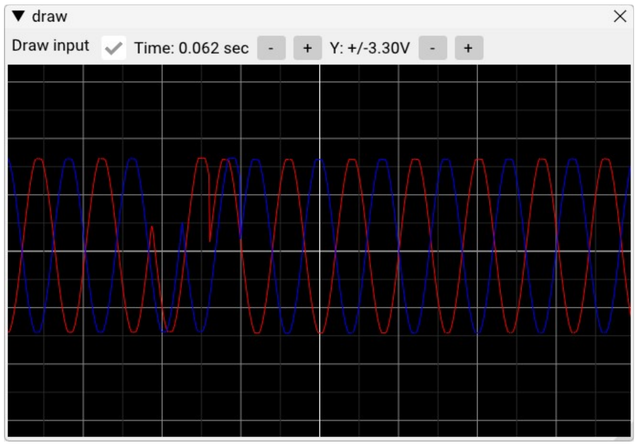
One of the most basic algorithms is the *attenuator*, which reduces the amplitude of the input signal by a given factor. An example involving a fixed attenuation factor will be shown first, followed by a more complex algorithm which utilizes one of the parameter knobs on the add-on board to adjust the attenuation factor while the algorithm is executing.

Below is one way to program an attenuator with a factor of 0.5, halving the amplitude of the incoming signal:

```
Sample* process_data(Samples samples)
{
    for (int i = 0; i < SIZE; ++i) {
        samples[i] = samples[i] / 2 + 1024;

    }

    return samples;
}
```

This algorithm uses a `for` loop to iterate through each sample in the `samples` array. For each sample, the sample's value is halved to apply our attenuation. However, if the algorithm only performed this division then we will have introduced an offset: the 0V sample value of 2,047 would become 1,023, or -1V. To fix this we need to add 1,024 to re-center the signal.

To test this algorithm, load the signal generator with a sine wave function (e.g. "sin(x/32)"). Then, compile and upload the algorithm, open the Plot over time window, and start executing the algorithm. Observe that the amplitude of the output signal is 50% of the view's span even though the supplied sine wave has a "100%" amplitude.



If we choose to overlay the input signal to see the change in amplitude, we run into a problem: the input and output signals appear to be the same! This is not a bug, but is in a fact a problem of our own creation. By reusing the input sample buffer as our output buffer, we overwrite the input data before the plot can display it. The solution would be to declare a separate output buffer, which we will do next with the parameter knob example.

Using a parameter knob

Let's use a parameter knob to generalize our algorithm so that we can control the attenuation factor ourselves. The two knobs on the add-on board can be read by your algorithm using the `param1()` and `param2()` functions. It's important to remember that these functions involve analog-to-digital conversions that take time to complete, so to keep your algorithm efficient you should only read the knob(s) once at the top of the `process_data` function.

Attenuation is a gain factor that is less than 1.0, so we will need a way to scale the knob reading (a value between zero and 4,095) to a range of 0.0 to 1.0. We can easily do this by dividing by 4,095:

```
const auto p = param1();
const float scale = p / 4095.f;
```

Next, we need to compute the offset necessary to keep our sample values centered at 2,048. In the fixed attenuation example, a factor of 0.5 (equal to a knob reading of 2,048) required an offset of 1,024 – half of what the knob's reading would be. So, the offset will be found by dividing the knob's reading by two.

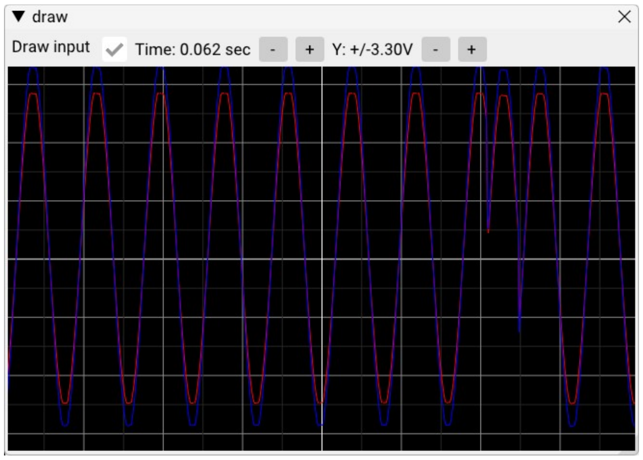
We also know that as the knob's reading decreases, reducing the gain factor, the offset needs to increase. This requires inverting the knob's reading, giving this final equation:

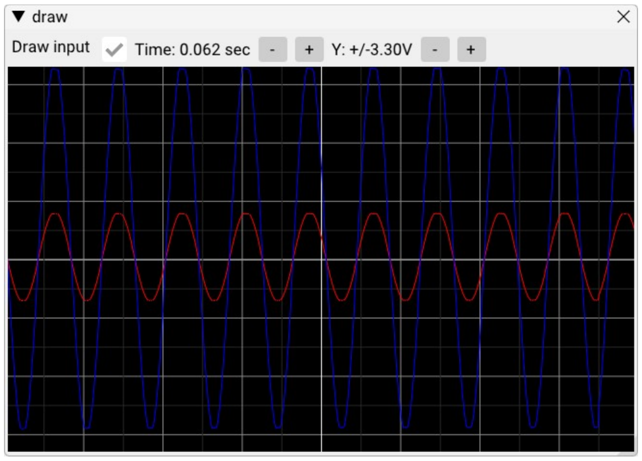
```
const auto offset = (4095 - p) / 2;
```

With our gain factor and offset calculated, we can now adapt the original attenuation algorithm to use them. We will also declare a dedicated output buffer so that we can visualize both the input and output signals. The result should look something like this:

```
Sample* process_data(Samples samples)
{
    static Samples output;

    const auto p = param1();
    const float scale = p / 4095.f;
    const auto offset = (4095 -   p) / 2;

    for (int i = 0; i < SIZE; ++i)
        output[i] = samples[i] * scale + offset;

    return output;
}
```

Compile and execute this algorithm on your DSP PAW hardware. Observe how the amplitude of the output waveform changes as you adjust the parameter knob:





3. FIR Filter: Moving Average

Finite Impulse Response (FIR) filters are very versatile and can be used for a variety of applications. These filters are commonly implemented through convolution, a kind of algorithm that can be written in code without much difficulty. This example will use convolution to implement a simple moving average filter.

Output buffer and filter definition

At a given point in time (i.e. a given index in the sample buffer), the moving average simply averages a window of input samples around that time-point to create the output sample for that time-point.

Since we will be creating new output sample values while still relying on the input values, it is essential for us to create a dedicated output sample buffer. This is done with a single declaration:

```
static Samples buffer;
```

Keep in mind that sample buffers take up a lot of our limited data memory -- you should not declare more than one additional `Samples` buffer in an algorithm.

We will also need to define our moving average kernel. A size of five should be sufficient for this example, meaning each value within the kernel will be 0.2 so that they sum to 1.0:

```
constexpr unsigned int filter_size = 5;
float filter[filter_size] = {
    0.2, 0.2, 0.2, 0.2, 0.2
};
```

Convolution implementation

The convolution algorithm will be made up of two `for` loops: one to iterate through each of the input samples, and another nested within the first to iterate through the kernel.

Since each average calculation requires neighbors both before and beyond itself in time, there are points at the beginning and end of our input sample buffer where we do not have enough neighbors to work with. The solution to this is “overlap-and-save”, where the algorithm is made to remember the last few input samples from its previous execution so that there are always enough samples to work with. This will be left out for now for simplicity, though that means we will see rough and inconsistent connections between each buffer of output samples.

In this case, our outer loop will iterate from the first sample up until `SIZE – (filter_size – 1)` so that there are always enough samples to work with.

```
for (int n = 0; n < SIZE - (filter_size - 1); n++) {
```

Within the loop, we will first zero the output sample's value and then begin the inner loop to iterate through the kernel:

```
buffer[n] = 0;
for (int k = 0; k < filter_size; k++) {
```

The inner loop takes the `k`-th input sample from the current sample buffer index and multiplies it by the kernel value.

```
    buffer[n] += samples[n + k] * filter[k];
}
```

And that's it! After these loops complete we just return the output sample buffer.

The complete algorithm is below:

```
Sample* process_data(Samples samples)
{
    static Samples buffer;

    constexpr unsigned int filter_size = 5;
    float filter[filter_size] = {
        0.2, 0.2, 0.2, 0.2, 0.2
    };

    for (int n = 0; n < SIZE - (filter_size - 1); n++) {
        buffer[n] = 0;
        for (int k = 0; k < filter_size; k++)
            buffer[n] += samples[n + k] * filter[k];
    }

    return buffer;
}
```

Noisy signal generation

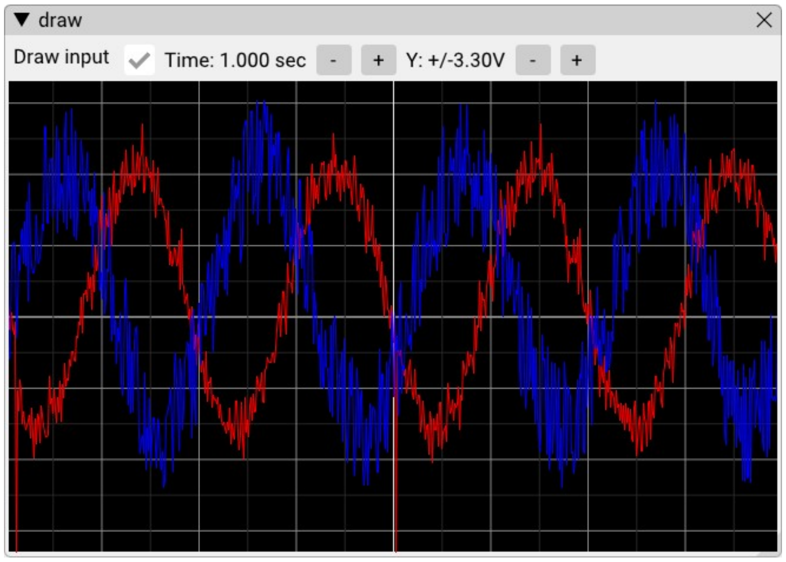
The moving average filter is effectively a low-pass filter: high-frequency changes in the sample data get averaged and smoothed out. To see this effect, we'll use the signal generator to create a sine wave with some added random noise.

Try this formula: `0.6*sin(2*3.1415*x/2048)+random(-0.3, 0.3)`

Sampling rate

We will also lower the sampling rate so that the distance in time between samples is larger, making the smoothing more noticeable. The sampling rate drop-down menu is next to the Compile and Upload buttons. The default is 32 kHz – change this to 8 kHz for this example.

Start the signal generator, begin executing the algorithm, and use Plot over time to view the input and output signals. The plot should look similar to the image below: the smoothing isn't the best it can be, but the amplitude of the added noise is noticeably lower in the red output signal.



4. Differentiator

Another easy algorithm to implement is a *differentiator*, which outputs the derivative of the input signal. This is accomplished by taking the difference between each pair of samples and using that for the output.

We will make one addition to this simple subtraction though. Consider that most signals will span across many samples: a triangle wave jumping between 0 and 4,095 every 100 samples will have a constant derivative of just +/- 400 or only +/- 0.2V. This might be difficult to observe, so we will add a scaling factor to amplify the derivative.

The gives us the below `for` loop:

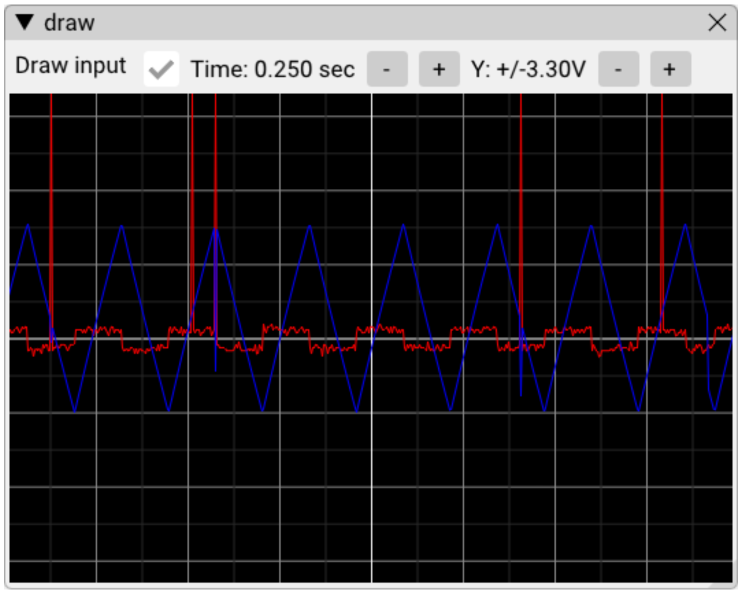
```
for (unsigned int i = 1; i < SIZE; i++) {
    output[i] = ((samples[i] - samples[i – 1]) // change between samples
                  * scaling_factor)            // amplification
                 + 2048;                       // keep 0V offset
}
```

Apart from the loop, we will need to define the scaling factor and an output sample buffer. We will also need to remember the last sample from the previous iteration so that we can calculate `output[0]`. The result is an algorithm that looks like this:

```
Sample* process_data(Samples samples)
{
    constexpr int scaling_factor = 25;
    static Samples output;
    static Sample prev = 2048;

    // Calculate the first output sample
    output[0] = 2048 + ((samples[0] - prev) * scaling_factor);

    for (unsigned int i = 1; i < SIZE; i++) {
        output[i] = ((samples[i] - samples[i - 1]) * scaling_factor)
                    + 2048;
    }

    // Remember last output sample for next output[0] calculation
    prev = samples[SIZE - 1];

    return output;
}
```

The signal generator can be used to create a triangle wave for testing. The output of our algorithm should like a square wave as a result, since the triangle wave's derivative jumps between positive and negative constant slopes.

Try this formula: `0.8*triangle(x/512)`. Compile the algorithm, upload it, and execute it with the time plot open. You should see something like below:



The square wave result (red) that we were looking for is nice and clear, though there are some glitches in the input signal (blue) that are being reflected in the output. This is most likely due to how the sample data from the signal generator was created; a formula with a different period may remove these glitches.

Conclusion

You’ve made it through the guide! It is still a work-in-progress, and although it has not covered all of DSP PAW’s features I hope you at least feel more prepared to design and test algorithms with this system.

If you have designed a new algorithm and would like to see it added to this guide, please reach out by creating an “issue” at https://github.com/bitgloo/dsp-paw/issues.