Plaquette Manual

Release 0.7.0

Sofian Audry

Jun 13, 2025

CONTENTS

1 Essentials **3**
- 1.1 Why Plaquette? . . . 3
  - 1.1.1 The Need for a New Standard . . . 3
  - 1.1.2 Meet Plaquette . . . 4
- 1.2 Features . . . 5
  - 1.2.1 Object-oriented . . . 5
  - 1.2.2 User-friendly . . . 5
  - 1.2.3 Signal-centric . . . 5
  - 1.2.4 Signal Filtering . . . 6
  - 1.2.5 Real-time . . . 6
  - 1.2.6 Arduino-compatible . . . 7
- 1.3 Reference . . . 8
  - 1.3.1 Base Units . . . 8
  - 1.3.2 Generators . . . 8
  - 1.3.3 Timing . . . 8
  - 1.3.4 Filters . . . 9
  - 1.3.5 Functions . . . 9
  - 1.3.6 Structure . . . 9
  - 1.3.7 Extra . . . 10

2 Guide **11**
- 2.1 Getting started . . . 11
  - 2.1.1 Step 1: Install Plaquette . . . 11
  - 2.1.2 Step 2: Your first Plaquette program . . . 12
  - 2.1.3 Step 3 : Experiment! . . . 14
  - 2.1.4 Learning More . . . 18
- 2.2 Inputs and Outputs . . . 19
  - 2.2.1 Digital vs Analog . . . 19
  - 2.2.2 Digital Inputs and Outputs . . . 20
  - 2.2.3 Analog Outputs and Inputs . . . 21
  - 2.2.4 Using Units as Their Own Values . . . 21
  - 2.2.5 The Piping Operator (>>) . . . 22
  - 2.2.6 Dealing with Noisy Signals: Debouncing and Smoothing . . . 23
  - 2.2.7 Mapping Values to Different Ranges . . . 24
  - 2.2.8 Making Decisions with Conditions . . . 24
  - 2.2.9 Modes for Inputs and Outputs . . . 25
  - 2.2.10 Conclusion . . . 27
- 2.3 Generating Waveforms . . . 27
  - 2.3.1 Visualizing Waves with the Serial Plotter . . . 27
  - 2.3.2 Types of Waves . . . 28

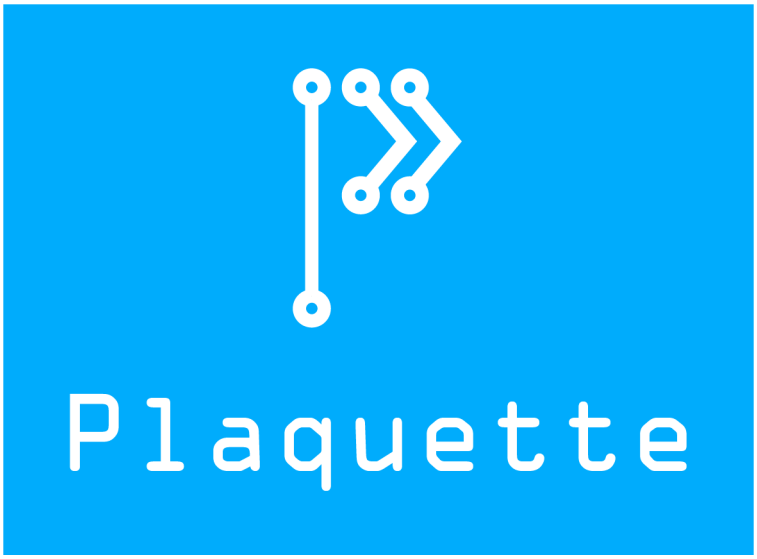
2.3.3 Wave Properties . . . 30
2.3.4 Wave Addition . . . 32
2.3.5 Modulation . . . 33
2.3.6 Adding Noise with randomFloat() . . . 33
2.3.7 Timing Functions . . . 35
2.3.8 Phase Shifting with shiftBy() . . . 35
2.3.9 Conclusion . . . 36
2.4 Working with Time . . . 36
2.4.1 Measuring Absolute Time with seconds() . . . 37
2.4.2 Timing Units . . . 37
2.4.3 Keeping Track of Time with Chronometer . . . 38
2.4.4 Scheduling with Alarm . . . 38
2.4.5 Triggering Periodic Events with Metronome . . . 39
2.4.6 Creating Smooth Transitions with Ramp . . . 40
2.4.7 Combining Timing Units . . . 46
2.4.8 Conclusion . . . 46
2.5 Regularizing Signals . . . 47
2.5.1 Direct Input-to-Output . . . 47
2.5.2 Getting the Full Range of a Signal . . . 48
2.5.3 Reacting to Signal Changes . . . 49
2.5.4 Adapting to Changing Conditions . . . 50
2.5.5 Normalizing Signals to Spot Extreme Values . . . 50
2.5.6 Detecting Peaks . . . 51
2.5.7 Conclusion . . . 53
2.6 Managing Events . . . 53
2.6.1 Supported Events . . . 53
2.6.2 Reacting to an Event . . . 54
2.6.3 Managing Multiple Events . . . 55
2.6.4 Coordinating Parallel Events with Metronomes . . . 55
2.6.5 Creating On-the-fly Callbacks . . . 56
2.6.6 Conclusion . . . 56
2.7 Advanced Usage . . . 57
2.7.1 Smoothing Arbitrary Signals . . . 57
2.7.2 Vanilla Coding Style . . . 57
2.7.3 Using Plaquette as an External Library . . . 58
2.7.4 Synchronizing Groups of Units with Secondary Engines . . . 58

3 Reference **63**
3.1 Base Units . . . 63
3.1.1 AnalogIn . . . 63
3.1.2 AnalogOut . . . 65
3.1.3 DigitalIn . . . 67
3.1.4 DigitalOut . . . 69
3.2 Generators . . . 72
3.2.1 SineWave . . . 72
3.2.2 SquareWave . . . 75
3.2.3 TriangleWave . . . 80
3.3 Timing . . . 83
3.3.1 Alarm . . . 83
3.3.2 Chronometer . . . 86
3.3.3 Metronome . . . 87
3.3.4 Ramp . . . 90
3.4 Filters . . . 94
3.4.1 MinMaxScaler . . . 94

3.4.2 Normalizer . . . 97
3.4.3 PeakDetector . . . 100
3.4.4 Smoother . . . 104
3.5 Functions . . . 106
3.5.1 mapFloat() . . . 106
3.5.2 mapFrom01() . . . 108
3.5.3 mapTo01() . . . 109
3.5.4 randomFloat() . . . 110
3.5.5 seconds() . . . 111
3.5.6 wrap() . . . 112
3.6 Structure . . . 113
3.6.1 Engine . . . 113
3.6.2 begin() . . . 116
3.6.3 step() . . . 117
3.6.4 [] (arrays) . . . 118
3.6.5 . (dot) . . . 119
3.6.6 >> (pipe) . . . 119
3.7 Extra . . . 120
3.7.1 Easings . . . 120
3.7.2 ContinuousServoOut . . . 121
3.7.3 ServoOut . . . 123
3.7.4 StreamIn . . . 124
3.7.5 StreamOut . . . 125

4 Related Info **129**
4.1 Credits . . . 129
4.1.1 Core Developers . . . 129
4.1.2 Contributors . . . 129
4.1.3 Partners . . . 129
4.1.4 Funding . . . 129
4.1.5 Inspiration . . . 130
4.2 License . . . 130

Index **131**



Plaquette is an object-oriented, user-friendly, signal-centric programming framework for **creative physical computing**. It promotes **expressiveness** over technical details while remaining fully compatible with Arduino, thus allowing **both novice and experienced** creative practitioners to **intuitively** design meaningful physical interactive systems.

Whether you are a beginner working with physical computing for the first time, an intermediate user familiar with creative signal-based softwares (eg. Max, Ossia Score, PureData, SuperCollider, TouchDesigner, etc.), or a seasoned Arduino creative coder: Plaquette is the perfect tool for your creative embedded and IoT projects.

Plaquette allows you to:

- React to multiple sensors and actuators in real-time without interruption.
- Automatically calibrate sensors to design stable interactions in changing environments.
- Design rich interactive behaviors by seamlessly combining powerful effects.

Quick links:

Discover the features	*Get started*
Watch video tutorials	*Filter your signals*
GitHub repository	PDF manual

CHAPTER

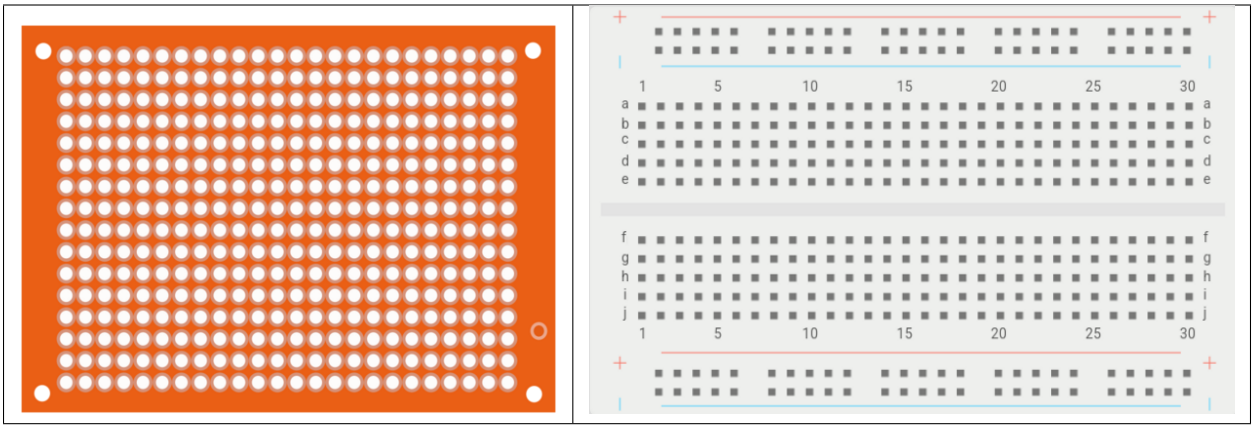
ONE

ESSENTIALS

1.1 Why Plaquette?

Plaquette is a groundbreaking creative coding framework designed to empower creative practitioners by simplifying the way they work with real-time signals in tangible computing applications. By bridging the gap between low-level electronics and high-level creative expression, Plaquette enables creators to focus on what matters most: bringing their visions to life.

Note: Plaquette is a French noun pronounced pla-kett which refers to prototyping plates or boards ("plaquette de prototypage") commonly used in designing electronic projects.



1.1.1 The Need for a New Standard

Media artists, interactive designers, digital luthiers, and electronic musicians constantly engage with real-time signals. However, when working with tangible computing systems such as embedded sensors, robotics, connected objects, and electronic music instruments, available tools such as Arduino are often very low-level and lack expressivity. Creative practitioners thus struggle to implement their vision directly using such platforms.

Consider the following case of learning how to work with a simple lignt sensor (eg. photoresistor) connected to an Arduino board on analog pin 0. The code reads as follows:

```
int value = analogRead(A0);
```

The value that is read is a raw 10-bit value returned by the Arduino board's Analog to Digital Converter (ADC), an integer between 0 and 1023. But how is this value intuitively useful for an artist who wants to use this value creatively?

For example, what if one wants to react to a flash of light? Well, one solution would be to look at the value and compare it to a threshold:

```
if (value > 716) {
  ...
}
```

There are two problems with this approach.

Firstly, while it might work under certain lighting conditions, it will likely stop working if these conditions change, forcing us to make adjustments to the threshold value by hand.

Secondly, and perhaps more importantly, this piece of code does not really *express* what we are after. As creative practitioners, we don't care whether the light signal is above 716 or 456 or whatnot: what we really want to know to detect a flash of light is whether the light signal is *significantly high compared to ambient light*.

What this example shows is that the way we are teaching and learning about sensor data is inefficient for creative applications. In other words: **raw digital data lacks expressiveness**.

Continuing with our example, consider how one would take the input value and directly reroute it to an analog (PWM) output on pin 9:

```
analogWrite(9, value / 4);
```

Why do we need to perform that division by 4? That's because while the ADC gives us 10-bit values (1024 possibilities), the PWM only supports 8 bits (256 possibilities) forcing us to divide the incoming value by 4 (2 bits). But again, why is this detail important to know for an artist, designer, or musician? And what exactly does it have to do with our expressive intention?

1.1.2 Meet Plaquette

As a way to address these issues, Plaquette offers a general-purpose, standard interface for simple, real-time signal processing tailored for media artists.

Plaquette's key objectives are:

1. **Empowering creators to focus on the creative aspects of their work**, rather than getting lost in irrelevant numerical details, which supports a smoother learning process.
2. **Providing accessible tools for creative practitioners** that capture high-level concepts such as *"normalizing"* and *"detecting peaks"*, without requiring deep technical knowledge of complex techniques like Fast Fourier Transforms or Chebyshev filtering.
3. **Facilitating teamwork and interoperability** by promoting an intuitive, cross-platform approach to real-time signals, such as by keeping all signals consistently scaled between 0 and 1 for easier integration across different applications.

Plaquette achieves these goals by embracing the following core principles:

- **Ease of use:** Offering a carefully selected set of functionalities that address the most common challenges faced by creators—keeping things simple while solving 95% of typical use cases.
- **Real-time performance:** Enabling smooth, uninterrupted interactions to ensure responsiveness in dynamic environments.
- **Signal-oriented approach:** Focusing on meaningful signal manipulation rather than dealing with arbitrary numerical values such as 255, 1024, 716, or 42.

- **Robustness:** Adapting to changes in the sensory context without breaking down, ensuring reliability in unpredictable and evolving environments such as art installations, live performances, and public art.
- **Interoperability and extensibility:** Leveraging an object-oriented architecture that seamlessly integrates with the Arduino ecosystem, ensuring compatibility and future scalability.

1.2 Features

Plaquette is an *object-oriented*, *user-friendly*, *signal-centric* framework that facilitates *signal filtering* in *real-time*. It is fully *compatible with Arduino*.

1.2.1 Object-oriented

Plaquette is designed using input, output, and filtering units that are easily interchangeable in a plug-and-play fashion. Units are created using expressive code.

For example, the code `DigitalOut led` creates a new digital output object that can be used to control an LED.

Arduino	Plaquette
Create digital output to control an LED:	
`pinMode(12, OUTPUT);`	`DigitalOut led(12);`
Create digital input push-button:	
`pinMode(2, INPUT);`	`DigitalIn button(2);`

1.2.2 User-friendly

Plaquette allows users to quickly design interactive systems using an expressive language that abstracts low-level functions. This allows both beginners and experts to create truly expressive code. For example, switching our LED object on or off can be achieved by calling: `led.on()`. Find out more about Plaquette's base units by following *this link*.

Arduino	Plaquette
Turn LED on:	
`digitalWrite(12, HIGH);`	`led.on();`
Check if button is pushed:	
`if (digitalRead(2) == HIGH)`	`if (button.isOn())`

1.2.3 Signal-centric

Plaquette helps designers manipulate real-time signals from inputs to outputs. In Plaquette, signals are represented either as `true/false` conditions (in the case of digital binary signals such as those coming from a button or switch), or as floating-point numbers in the [0, 1] range (ie. 0% to 100%) (in the case of analog signals such as those emitted by a light sensor, microphone, or potentiometer.) Because of this, there is no more need for users to perform counter-intuitive conversions on integer values.

Arduino	Plaquette
Check if button is released:	
`if (digitalRead(2) != HIGH)`	`if (!button)`
Check if sensor value is higher than 70%:	
`if (analogRead(A0) >= 716)`	`if (sensor >= 0.7)`

1.2.4 Signal Filtering

Plaquette provides simple yet powerful data filtering tools for debouncing, smoothing, and normalizing data. Removing noise in input signals can be as simple as calling a function such as `debounce()` or `smooth()`. Rather than guessing the right threshold for triggering an event based on input sensor input, one can use auto-normalizing *filters* such as *MinMaxScaler* and *Normalizer*.

Signals in Plaquette can easily flow between units, in a similar fashion to modern data-flow software such as Max, Pure Data, and TouchDesigner. While this can be achieved using function calls, Plaquette provides a special **piping operator** (>>) which allows data to be sent from one unit to another.

Arduino	Plaquette
Set LED to ON when button is pressed:	
`digitalWrite(12, digitalRead(2)/4);`	`button >> led;`
Set LED to ON when input sensor is high:	
`digitalWrite(12, (analogRead(A0) >= 716 ? HIGH : LOW));`	`(sensor >= 0.7) >> led;`

Read *Regularizing Signals* to see how you can take full advantage of Plaquette's signal filtering features.

1.2.5 Real-time

Plaquette avoids blocking processes such as Arduino's (in)famous `delay()` by providing a set of *timing units* as well as time-based *signal generators*. As such, the processing loop is never interrupted, allowing interactive and generative processes to flow smoothly.

Plaquette forbids the use of blocking functions such as Arduino's delay() and delayMicroseconds(). Rather, it invites programmers to adopt a frame-by-frame approach to coding similar to Processing.

Compare the following attempt to make an LED blink when pressing a button in Arduino, versus Plaquette's real-time approach:

Arduino

```
int buttonPin = 2;
int ledPin = 12;

void setup() {
  pinMode(buttonPin, OUTPUT);
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // Button is checked once per second.
  if (digitalRead(buttonPin) == HIGH) {
    digitalWrite(ledPin, HIGH);
    delay(500); // do nothing for 500ms
    digitalWrite(ledPin, LOW);
    delay(500); // do nothing for 500ms
  }
}
```

Plaquette

```
DigitalIn button(2);
DigitalOut led(12);

// Square wave 1 second period.
SquareWave oscillator(1.0);

void begin() {}

void step() {
  // Button is checked at all time.
  if (button)
    oscillator >> led;
}
```

1.2.6 Arduino-compatible

Plaquette is installed as an Arduino library and provides a replacement for the core Arduino functionalities while remaining fully compatible with Arduino code. Seasoned Arduino users should consult the *Advanced Usage* section for some tips on how to integrate Plaquette into their existing code.

The following example uses Plaquette to control a blinking LED that slows down with each button push, using Arduino's constrain() to keep the LED oscillation period within a certain range and Serial object to reset the counter to a random integer value using random().

```
#include <Plaquette.h>

DigitalIn button(2, INTERNAL_PULLUP); // button input

DigitalOut led(LED_BUILTIN); // LED output

SquareWave oscillator(1.0); // square oscillator

int currentPeriod = 0; // oscillator period counter

void begin() {
   button.debounce(); // debounce button
}

void step() {
   if (Serial.read() == 'R') // reset counter
     currentPeriod = random(1, 10);

   if (button.rose()) // true when value rises (ie. button is pushed)
     currentPeriod = constrain(currentPeriod+1, 1, 10); // increment

   oscillator.period(currentPeriod); // set period
   oscillator >> led; // send signal to LED
}
```

Danger: Plaquette needs the main processing loop to run continuously without interruption to work correctly. Users should thus **avoid using blocking processes** such as Arduino's delay() and delayMicroseconds() and functions in their code when using Plaquette.

Warning: Many of the core Arduino functions work with integer types such as `int` or `long` rather than floating-point types such as `float`. Plaquette provides alternative *functions* which should be used instead.

In particular, please use:

- *mapFloat()* instead of map()
- *randomFloat()* instead of random()
- *seconds()* instead of millis()

Warning: Plaquette is still at an experimental stage of development. If you have any issues or questions, please contact the developers, or file a bug in our issue tracker.

1.3 Reference

1.3.1 Base Units

- *AnalogIn* Reads analog input values between 0 and 1. Typically used for sensors that output a range of values, such as potentiometers or light sensors.
- *AnalogOut* Writes analog output values between 0 and 1. Useful for controlling devices like dimmable LEDs or motor controllers.
- *DigitalIn* Reads digital input values as boolean true/false. Commonly used with buttons, switches, or other on/off signals.
- *DigitalOut* Writes digital output values as boolean true/false. Often used to control relays, LEDs, or other binary devices.

1.3.2 Generators

- *SineWave* Generates a smooth, periodic sine wave signal. Commonly used for oscillatory motion or smooth transitions.
- *SquareWave* Generates a periodic square wave signal, alternating between high and low states. Ideal for toggling signals like blinking LEDs.
- *TriangleWave* Produces a periodic triangle wave signal with a linear rise and fall. Useful for generating smoother oscillatory patterns than square waves.

1.3.3 Timing

- *Alarm* Triggers an event after a specified time duration. Can be used to schedule delays or time-based actions.
- *Chronometer* Measures elapsed time since the start or reset. Useful for timing events or profiling system performance.
- *Metronome* Produces a periodic tick at specified intervals. Often used in rhythmic applications such as sound or light pulses.
- *Ramp* Generates a linear ramp signal over time. Commonly used for smooth parameter transitions like fading lights or scaling values.

1.3.4 Filters

- *MinMaxScaler* Scales signals to fit within a specified minimum and maximum range. Essential for normalizing input signals from diverse sources.
- *Normalizer* Adjusts signals to have a zero mean and unit variance. Useful in signal processing pipelines where consistent scaling is required.
- *PeakDetector* Detects peaks (local maxima) in input signals, allowing for event-based processing such as edge detection.
- *Smoother* Reduces noise and fluctuations in input signals using smoothing algorithms like exponential moving averages.

1.3.5 Functions

- *mapFloat()* Maps a float value from one range to another. Useful for adapting input ranges to the desired output domain.
- *mapFrom01()* Maps a float value from the normalized [0,1] range to a custom range, such as [-10, 10].
- *mapTo01()* Maps a float value from a custom range to the normalized [0,1] range, simplifying calculations for normalized operations.
- *randomFloat()* Generates a random float between 0 and 1, ideal for simulations or procedural generation.
- *seconds()* Returns the current time in seconds since the program started, enabling precise time tracking.
- *wrap()* Wraps a value within a specified range, making it cyclic. Commonly used for angles or periodic parameters.

1.3.6 Structure

- *Engine* A control structure managing an ensemble of units, handling their initialization, update, and timing, ensuring they remain synchronized.
- *begin()* Initializes the system, similar to Arduino's *setup()* function. Sets up necessary configurations and prepares units for operation.
- *step()* Repeatedly called during the program's execution, akin to Arduino's *loop()* function. Drives the execution of the main logic.
- *[] (arrays)* Allows the creation of arrays of Plaquette units for batch operations. Facilitates efficient processing of multiple units simultaneously.
- *. (dot)* Provides access to an object's methods and data, enabling intuitive object-oriented programming with Plaquette units.
- *>> (pipe)* Sends data across units from left to right, creating a streamlined and intuitive flow of information between connected units.

1.3.7 Extra

- *Easings* Provides easing functions for smooth and natural transitions between values. Commonly used in animations and motion design.
- *ContinuousServoOut* Controls a continuous rotation servo motor by setting its speed and direction. Ideal for robotics or mechanical motion control.
- *ServoOut* Controls a standard servo motor by setting its angle. Useful for applications like robotic arms or pan-tilt systems.
- *StreamIn* Streams input data continuously, allowing real-time signal processing from external devices.
- *StreamOut* Streams output data continuously, enabling real-time control of external actuators or visualizations.

CHAPTER

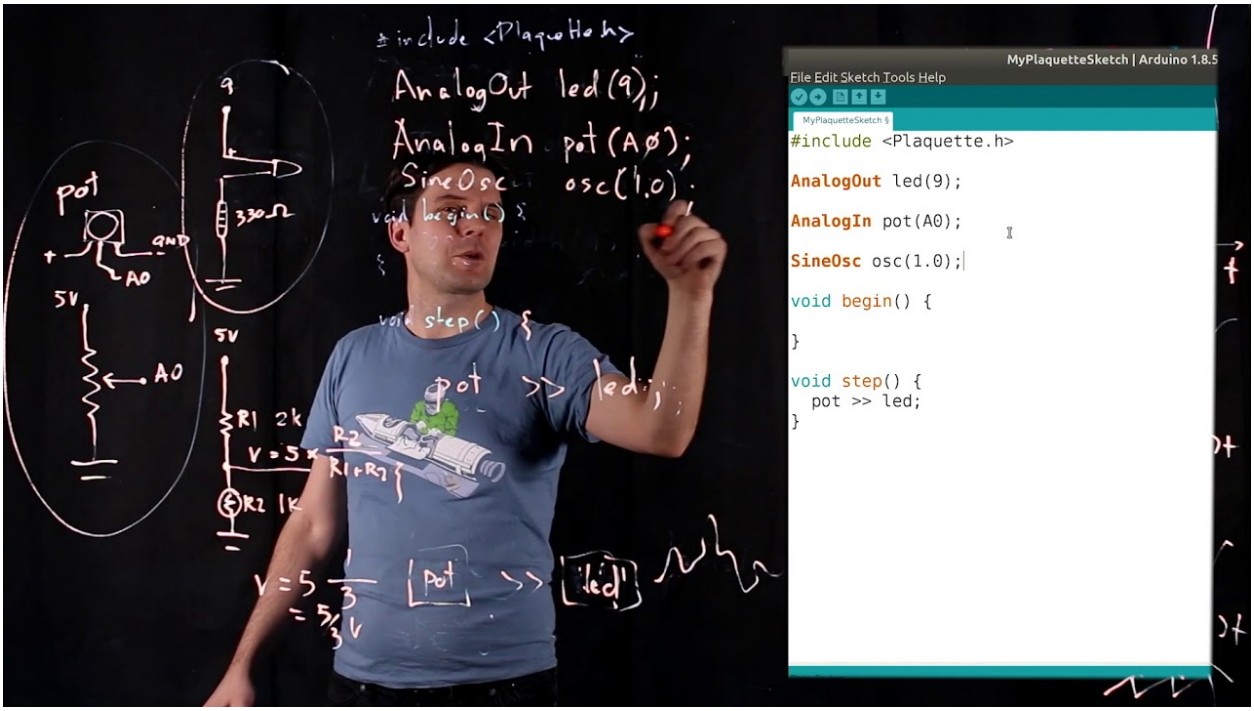
TWO

GUIDE

2.1 Getting started

This short introduction will guide you through the first steps of using Plaquette.

We also recommend watching our introductory video tutorial series.



2.1.1 Step 1: Install Plaquette

If you do not have Arduino installed on your machine you need to download and install the Arduino IDE for your platform.

Once Arduino is installed, please install Plaquette as an Arduino library following these instructions.

2.1.2 Step 2: Your first Plaquette program

We will begin by creating a simple program that will make the built-in LED on your microcontroller blink.

Create a new sketch

Create a new empty sketch by selecting **File > New**.

IMPORTANT: New Arduino sketches are initialized with some "slug" starting code. Make sure to erase the content of the sketch before beginning. You can use **Edit > Select All** and then click **Del** or **Backspace**.

Include library

Include the Plaquette library by typing:

```
#include <Plaquette.h>
```

Create an output unit

Now, we will create a new unit that will allow us to control the built-in LED:

```
DigitalOut myLed(13);
```

In this statement, `DigitalOut` is the **type** of unit that we are creating. There also exists other types of units, which will be described later. `DigitalOut` is a type of software unit that can represent one of the many hardware pins for digital output on the Arduino board. One way to think about this is that the `DigitalOut` is a "virtual" version of the Arduino pin. These can be set to one of two states: ("on/off", "high/low", "1/0").

The word `myLed` is a **name** for the object we are creating.

Finally, `13` is a **parameter** of the object `myLed` that specifies the hardware *pin* that it corresponds to on the board. In English, the statement would thus read as: "Create a unit named `myLed` of type `DigitalOut` on pin `13`."

Tip: Most Arduino boards have a pin connected to an on-board LED in series with a resistor and on most boards, this LED is connected to digital pin `13`. The constant `LED_BUILTIN` is the number of the pin to which the on-board LED is connected.

Create an input unit

We will now create another unit that will generate a signal which will be sent to the LED to make it blink. To this effect, we will use the `SquareWave` unit type which generates a square wave oscillating between "on/high/one" and "off/low/zero" at a regular period of 2.0 seconds:

```
SquareWave myWave(2.0);
```

Create the begin() function

Each Plaquette sketch necessitates the declaration of two functions: `begin()` and `step()`.

Function `begin()` is called only once at the beginning of the sketch (just like the setup() function in Arduino). For our first program, we do not need to perform any special configuration at startup so we will leave the `begin()` function empty:

```
void begin() {}
```

Create the step() function

The `step()` function is called repetitively and indefinitely during the course of the program (like the loop() function in Arduino).

Here, we will send the signal generated by the `myWave` input unit to the `myLed` output unit. We will do this by using Plaquette's special >> operator:

```
void step() {
  myWave >> myLed;
}
```

In plain English, the statement `myWave >> myLed` reads as: "Take the value generated by `myWave` and put it in `myLed`."

Upload sketch

Upload your sketch to the Arduino board. You should see the LED on the board **blinking once every two seconds at a regular pace**.

Et voilà!

Full code

```
#include <Plaquette.h>

DigitalOut myLed(13);

SquareWave myWave(2.0);

void begin() {}

void step() {
  myWave >> myLed;
}
```

2.1.3 Step 3 : Experiment!

So far so good. Let's see if we can push this a bit further.

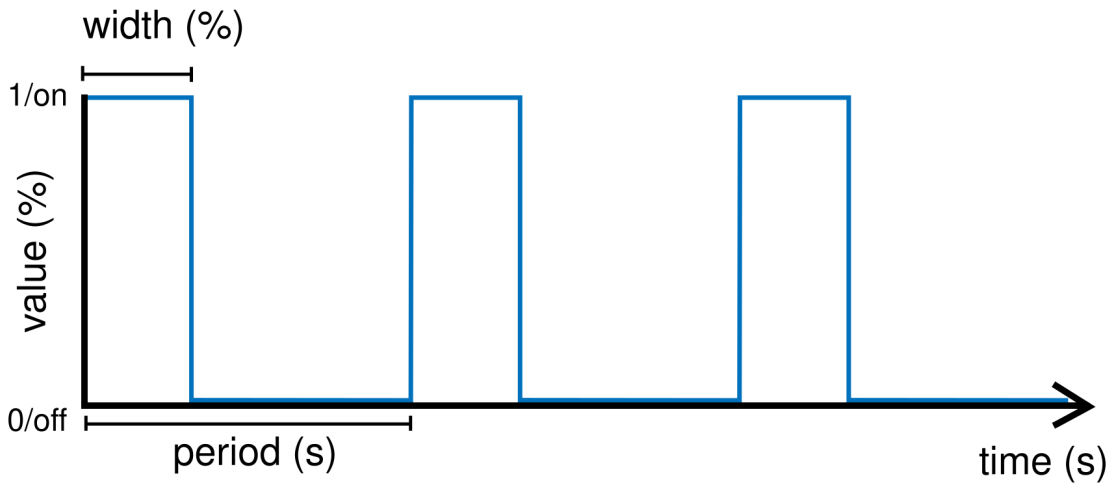
Change initial parameters of a unit

The `SquareWave` unit type provides two parameters when it is created that allows you to configure the oscillator's behavior. Both are optional: if not specified, they will take default values.

```
SquareWave myWave(period, width);
```

- `period` can be any positive number representing the period of oscillation (in seconds)
- `width` can be any number between 0.0 (0%) and 1.0 (100%), and represents the proportion of the period during which the signal is "high" (ie. "on duty") (default: 0.5)

Note: We call this step the **construction** or **instantiation** of the object `myWave`.



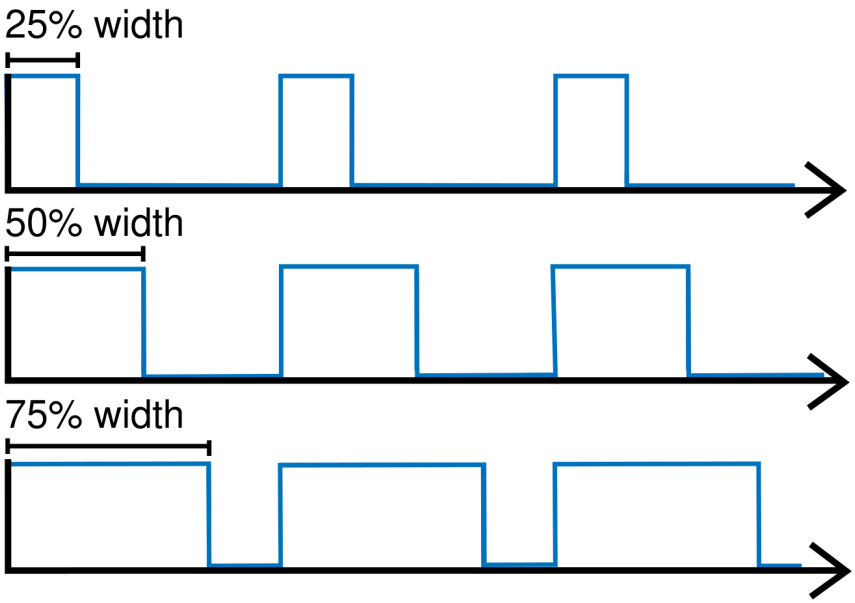
Try changing the first parameter (period) in the square oscillator unit to change the period of oscillation.

- `SquareWave myWave(1.0);` for a period of one second
- `SquareWave myWave(2.5);` for a period of 2.5 seconds
- `SquareWave myWave(10.0);` for a period of 10 seconds
- `SquareWave myWave(0.5);` for a period of half a second (500 milliseconds)

Important: Don't forget to re-upload the sketch after each change.

Now try adding a second parameter (width) to control the oscillator's width. For a fixed period, try changing the duty cycle to different percentages between 0.0 and 1.0.

- `SquareWave myWave(2.0, 0.5);` for a width of 50% (default)
- `SquareWave myWave(2.0, 0.25);` for a width of 25%
- `SquareWave myWave(2.0, 0.75);` for a width of 75%
- `SquareWave myWave(2.0, 0.9);` for a width of 90%



Change parameters of a unit during runtime

What if we wanted to change the parameters of the oscillator during runtime rather than just at the beginning? The `SquareWave` unit type allows real-time modification of its parameters by calling one of its functions using the . *(dot)* operator.

For example, to change the period, simply call the following inside the `step()` function:

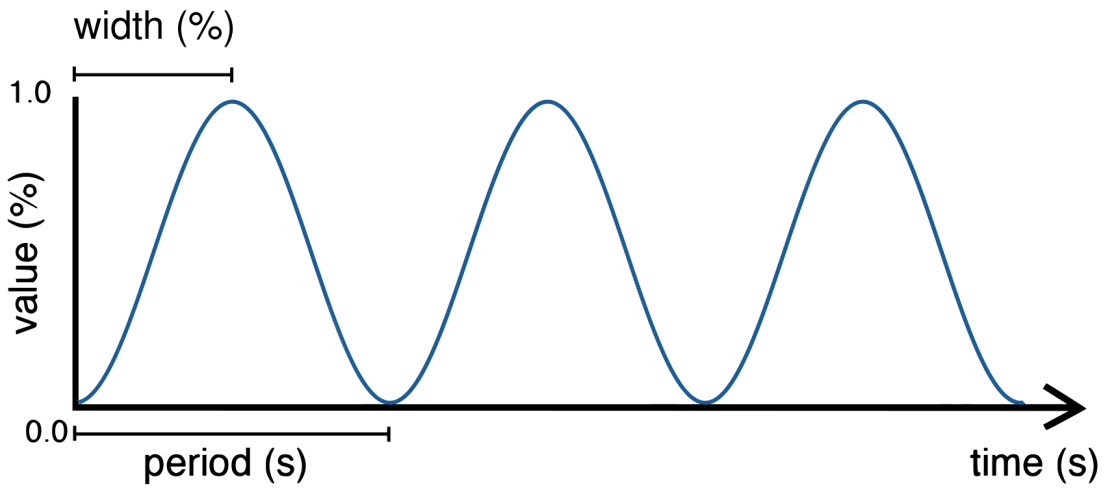
```
void step() {
  myWave.period(newPeriod);
  myWave >> myLed;
}
```

Of course, to accomplish our goal, we need a way to *change* the value `newPeriod` during runtime. We can accomplish this in many different ways, but let's try something simple: we will use another wave to *modulate* our wave's period.

For this, we will be using another kind of source called a `SineWave` and will use its outputs to change the period of `myWave`.

```
SineWave myModulator(20.0);
```

This wave will oscillate smoothly from 0 to 1 every 20 seconds.




```
void step() {
  myWave.period(myModulator);
  myWave >> myLed;
}
```

Upload the sketch and you should see the LED blinking as before, with the difference that the blinking speed will now change from blinking very fast (in fact, infinitely fast, with a period of zero seconds!) to very slow (period of 20 seconds).

Tip: If you want to visualize the values of both waves on your computer, you can print them on the serial port one after the other, separated by a space. Add the following code to your `step()` function:

```
print(myWave);
print(" ");
println(myModulator);
```

Then, launch the Arduino Serial Plotter by selecting in in **Tools > Serial Plotter**.

Now try modulating the width of `myWave` instead of its period:

```
myWave.width(myModulator);
```

Use a button

Now let's try to do some very simple interactivity by using a simple switch or button. For this we will be using the internal pull-up resistor available on Arduino boards for a very simple circuit. One leg of the button should be connected to ground (GND) while the other should be connected to digital pin 2.

Tip: If you do not have a button or switch, you can just use two electric wires: one connected to ground (GND) and the other one to digital pin 2. When you want to press the button, simply touch the wires together to close the circuit.

Declare the button unit with the other units at the top of your sketch:

```
DigitalIn myButton(2, INTERNAL_PULLUP);
```

You will notice that the type of this unit (*DigitalIn*) resembles that of our LED-controlling unit (*DigitalOut*). This is because both units have something in common: they have only two states: either on or off, high or low, true or false, one or zero, hence the adjective `Digital`. However, while the LED is considered an output or actuator (`Out`) our button is rather an input or sensor (`In`).

Note: If you are curious, you might also want to know that there is an *AnalogIn* and an *AnalogOut* types which support sensors and actuators that work with continuous values between 0 and 1 (0% to 100%).

Now, let's use this button as a way to control whether the LED blinks or not. For this, we will need to use the value of the button as part of a **condition** for an if…else statement.

```
void step() {
  if (myButton)
    myWave >> myLed;
  else
    0 >> myLed;
}
```

Full code

```
#include <Plaquette.h>

DigitalOut myLed(13);

SquareWave myWave(2.0);

SineWave myModulator(20.0);

DigitalIn myButton(2, INTERNAL_PULLUP);

void begin() {}

void step() {
  myWave.period(myModulator);

  if (myButton)
    myWave >> myLed;
```

(continues on next page)

(continued from previous page)

```
  else
    0 >> myLed;
}
```

2.1.4 Learning More

Built-in Examples

You will find more examples here or directly from the Arduino software in **File > Examples > Plaquette** including:

- Using analog inputs such as a photocells or potentiometers
- Using analog outputs
- Serial input and output
- Using wave generators
- Time management
- Ramps
- Basic filtering (smoothing, re-scaling)
- Peak detection
- Event-driven programming
- Controlling servomotors

The Plaquette Reference

The online reference can be accessed *here* or directly from the sidebar of the Plaquette website. It provides detailed technical documentation for every available unit and function in Plaquette. This reference serves as a go-to resource for understanding the specifics of each component, including their parameters, methods, and behavior.

Here are the key sections of the reference:

- *Base Units*: Introduces foundational units like *DigitalIn*, *DigitalOut*, *AnalogIn*, and *AnalogOut*. These are the building blocks for interfacing with hardware pins.
- *Generators*: Covers the signal generators *SquareWave*, *TriangleWave*, and *SineWave*. These are used to create various oscillating or periodic signals.
- *Timing*: Focuses on units related to time management, like *Metronome* for periodic events, *Alarm* for duration-based functionality, and *Ramp* for smooth transitions.
- *Filters*: Discusses tools for *smoothing*, *scaling*, or *normalizing* signals, as well as *detecting peaks*.
- *Functions*: Explains helper functions for tasks like value mapping, signal transformations, and conversions.
- *Structure*: Describes core structural functions and operators.
- *Extra*: Contains miscellaneous units and features.

What's Next?

With the basics covered, you are now ready to dive deeper into Plaquette's capabilities. Explore the rest of the guide to learn about specific features and advanced techniques:

- *Inputs and Outputs*: Learn how to use Plaquette to handle a variety of inputs and outputs, including analog, digital, and specialized sensors or actuators.
- *Generating Waveforms*: Understand the different types of wave generators available and how they can be used for oscillatory or periodic behavior.
- *Working with Time*: Delve into Plaquette's timing management units to handle scheduling and time-based logic in your projects.
- *Regularizing Signals*: Discover methods for automatically scaling and normalizing signals amd respond to peaks.
- *Managing Events*: Trigger actions, schedule events and manage parallel loops using event-driven programming.

2.2 Inputs and Outputs

When working with Plaquette, **inputs** and **outputs** allow you to interact with the physical world. Whether you are reading a sensor's value or controlling an actuator, Plaquette makes this process intuitive and efficient. This section introduces the basic concepts of digital and analog inputs and outputs, presents Plaquette's unique syntax for efficient and expressive code, explains how to clean noisy data, shows how to make decisions based on input values, and describes the different configuration modes of input and output units.

Let's explore these ideas step by step.

2.2.1 Digital vs Analog

Before diving into code, let's first clarify the difference between **digital** and **analog** signals.

- **Digital** signals represent binary states: ON vs OFF, HIGH vs LOW, 1 vs 0, true vs false. Examples of digital inputs include buttons and presence sensors, while digital outputs might control LEDs or relays.
- **Analog** signals represent a continuous range of values. Think of a dimmer switch (potentiometer) or a light sensor that measures brightness levels. Analog outputs control devices such as LEDs with variable brightness and DC motors where the speed can vary.

Warning: On many microcontrollers, analog outputs are generated using **Pulse Width Modulation (PWM)** rather than a true **Digital-to-Analog Converter (DAC)**. PWM rapidly switches the output pin between HIGH and LOW, creating the illusion of a continuous analog voltage when averaged over time. While this works for controlling brightness in LEDs or speed in motors, it may not be suitable for applications requiring a steady, smooth signal.

For more information, visit the official Arduino documentation on Pulse Width Modulation.

2.2.2 Digital Inputs and Outputs

Let's look at the basics of digital signals by working with LEDs and buttons.

Digital Outputs

A *DigitalOut* unit controls devices that can be turned ON or OFF, such as LEDs or relays. Here's an example of how to control an LED:

```
#include <Plaquette.h>

DigitalOut led(13); // LED connected to pin 13

void begin() {
  led.on(); // Turn the LED on initially
}

void step() {
  if (seconds() > 10.0)
    led.off(); // Turn the LED off after 10 seconds
}
```

Digital Inputs

A *DigitalIn* unit reads binary states from devices like buttons or switches. The easiest way to connect a button or switch is to use the **internal pull-up** resistor on Arduino boards. One leg of the button should be connected to ground (GND) while the other should be connected to a digital pin.

```
#include <Plaquette.h>

DigitalIn button(2, INTERNAL_PULLUP); // Button connected to pin 2
DigitalOut led(13);                   // LED connected to pin 13

void begin() {}

void step() {
  if (button.isOn()) {  // Check if the button is pressed
    led.on();
  } else {
    led.off();
  }
}
```

2.2.3 Analog Outputs and Inputs

Next, we will explore analog signals, which allow for finer control and more detailed readings.

Analog Outputs

An *AnalogOut* unit controls devices like LEDs or motors with continuous values. Here's an example of dimming an LED. The cathode (short leg) of the LED should be connected to ground, while the anode (long leg) should be connected to a 300 Ω resistor, which in turn should be connected to an analog / PWM pin (eg. pin 9).

```
#include <Plaquette.h>

AnalogOut led(9); // LED connected to pin 9

void begin() {
  led.put(0); // Set LED brightness to 0%
}

void step() {
  led.put( seconds() / 10 ); // Will reach 100% after 10 seconds
}
```

Analog Inputs

An *AnalogIn* unit reads continuous values from sensors, such as potentiometers, light, or temperature sensors.

Let's use a potentiometer to control an LED's brightness. For this circuit, the center pin of the potentiometer should be connected to analog input pin (`A0`), the left pin to ground (GND) and the right pin to +5V (Vcc).

```
#include <Plaquette.h>

AnalogIn dimmer(A0); // Potentiometer on analog pin A0
AnalogOut led(9);    // LED on pin 9

void begin() {}

void step() {
  led.put(dimmer.get()); // Map the potentiometer value directly to LED brightness
}
```

2.2.4 Using Units as Their Own Values

Plaquette offers an elegant shortcut: you don't need to explicitly call `isOn()` or `get()` for digital or analog inputs. Instead, you can use the input or output unit itself in lieu of the value it contains. This makes your code cleaner and easier to read.

Here's the same LED and button example, rewritten with this feature:

```
#include <Plaquette.h>

DigitalIn button(2, INTERNAL_PULLUP);
```

(continues on next page)

(continued from previous page)

```
DigitalOut led(13);

void begin() {}

void step() {
  if (button) {  // No need for button.isOn() : just use button as its own value
    led.on();
  } else {
    led.off();
  }
}
```

For analog inputs, this works similarly. Instead of calling `dimmer.get()`, you can use the `dimmer` unit directly:

```
#include <Plaquette.h>

AnalogIn dimmer(A0);
AnalogOut led(9);

void begin() {}

void step() {
  led.put(dimmer); // No need for dimmer.get() : just use dimmer as its own value
}
```

These simplifications make your code more expressive and emphasize the logic over the syntax.

2.2.5 The Piping Operator (>>)

In Plaquette, the >> operator allows you to directly send or "pipe" the value of one unit to another. This makes it incredibly simple to map inputs to outputs without extra variables or function calls.

Let's revisit the potentiometer and LED example using the piping operator:

```
#include <Plaquette.h>

AnalogIn dimmer(A0);
AnalogOut led(9);

void begin() {}

void step() {
  dimmer >> led; // Directly pipe the potentiometer value to the LED
}
```

This operator improves code readability and emphasizes the relationship between inputs and outputs.

Note: The piping operator (>>) allows to expressively connect input, output, and filtering units in a similar fashion to data-flow environments such as Max, Pure Data, and TouchDesigner. The operator is directly inspired from the ChucK operator (=>) in programming languge ChucK.

2.2.6 Dealing with Noisy Signals: Debouncing and Smoothing

In real-world applications, signals can be messy. Buttons can produce electrical noise when pressed, and analog sensors might give fluctuating readings. Plaquette provides tools to handle these issues: **debouncing** for digital signals and **smoothing** for analog ones.

Debouncing

Debouncing ensures that a button press is recorded cleanly, ignoring any noise. Here's how to debounce a button:

```
#include <Plaquette.h>

DigitalIn button(2, INTERNAL_PULLUP); // Button with pull-up resistor
DigitalOut led(13);                   // LED on pin 13

void begin() {
  button.debounce(); // Debounce the button
}

void step() {
  if (button.rose()) {  // Detect a clean press
    led.toggle();       // Toggle the LED state
  }
}
```

Smoothing

For analog signals, smoothing helps stabilize noisy data.

Here's how you can smooth a light sensor (photoresistor). For this circuit, you will need to create a simple voltage divider circuit. Connect the photoresistor between the ground (GND) and the analog input pin (`A0`). Then connect a fixed resistor with value matching your photoresistor between analog input pin and +5V (Vcc). For example, for a 1k Ω - 10k Ω photoresistor you could use a fixed resistor of about 5.5k Ω).

```
#include <Plaquette.h>

AnalogIn lightSensor(A0);
AnalogOut led(9);

void begin() {
  lightSensor.smooth(); // Apply default smoothing
}

void step() {
  lightSensor >> led;
}
```

You can adjust the level of smoothing and deboucing by indicating a parameter representing the time window (in seconds) over which the value is averaged. Experiment with different smoothing values to see the result:

- `lightSensor.smooth()` : Default smoothing window (100ms)
- `lightSensor.smooth(1.0)` : Smooth over one second

- `lightSensor.smooth(10.0)` : Smooth over 10 seconds
- `lightSensor.smooth(0.01)` : Smooth over 10ms

2.2.7 Mapping Values to Different Ranges

Sometimes, the output of a sensor doesn't match the range needed for an actuator. Plaquette provides a simple **mapping function** `mapTo(low, high)` which maps the analog input value to a specified range which is very useful for scaling sensor readings.

Example: Controlling the blinking frequency of an LED based on the value of a light sensor.

```
#include <Plaquette.h>

AnalogIn lightSensor(A0);
DigitalOut led(13);
SquareWave wave(1.0);

void begin() {}

void step() {
  // Map sensor value to frequency in range 1-10 Hz
  wave.frequency( lightSensor.mapTo(1, 10) );
  // Control LED with wave.
  wave >> led;
}
```

2.2.8 Making Decisions with Conditions

Interactive systems often need to respond to changes in input. Plaquette provides convenient methods like `rose()`, `fell()`, and `changed()` for detecting transitions in digital signals.

Digital Conditions

Here's an example of toggling an LED when a button is pressed:

```
#include <Plaquette.h>

DigitalIn button(2, INTERNAL_PULLUP);
DigitalOut led(13);

void begin() {}

void step() {
  if (button.rose()) { // Detect the moment the button is pressed
    led.toggle();      // Toggle the LED state
  }
}
```

Analog Conditions

Analog conditions are useful when you want to trigger actions based on a threshold. For instance, turning on an LED when the light level drops below 30% (0.3):

```
#include <Plaquette.h>

AnalogIn lightSensor(A0);
DigitalOut led(13);

void begin() {}

void step() {
  if (lightSensor < 0.3) {
    led.on();  // Turn on LED in low light
  } else {
    led.off(); // Turn off LED in bright light
  }
}
```

2.2.9 Modes for Inputs and Outputs

All input and output units in Plaquette support different modes, which allow you to adapt to various circuit configurations. You may already be familiar with the `INTERNAL_PULLUP` mode from *DigitalIn*, which provides a simple way to connect a button input. Let's explore how modes affect *DigitalIn*, *AnalogIn*, *DigitalOut*, and *AnalogOut* units.

Understanding these modes helps you design stable and efficient circuits, whether you're reading inputs or driving outputs. Choose the mode that best fits your hardware setup and application requirements.

DigitalIn Modes: DIRECT, INVERTED, and INTERNAL_PULLUP

The *DigitalIn* unit supports three primary modes:

- **DIRECT** (default): The unit is ON when the input pin is HIGH (e.g., 5V). This mode is used for buttons with pull-down resistors, which keep the pin LOW (OFF) when the button is not pressed and allow it to go HIGH (ON) when the button is pressed. Pull-down resistors typically have values around 10k Ω.

  Example: Button connected between pin 2 and 5V with a pull-down resistor to ground:

  ```
  DigitalIn button(2, DIRECT);
  DigitalOut led(13);

  void step() {
    if (button) {
      led.on();
    } else {
      led.off();
    }
  }
  ```

- **INVERTED**: The unit is ON when the input pin is LOW (e.g., GND). This is useful for buttons with pull-up resistors, which keep the pin HIGH when the button is not pressed and allow it to go LOW when the button is pressed. The `INTERNAL_PULLUP` mode activates an internal pull-up resistor, simplifying the circuit.

Example: Button connected between pin 2 and ground with a pull-down resistor to +5V (Vcc):

```
DigitalIn button(2, INVERTED);
```

- **INTERNAL_PULLUP**: As in mode `INVERTED` the unit is ON when the input pin is LOW (e.g., GND). Makes use of the internal pull-up resistor on the board, therefore removing the need to add a pull-up resistor.

  Example: Button connected between pin 2 and ground (no need for an extra pull-up resistor):

```
DigitalIn button(2, INTERNAL_PULLUP);
```

AnalogIn Modes: DIRECT and INVERTED

The *AnalogIn* unit also supports `DIRECT` and `INVERTED` modes, which determine how the sensor's voltage is interpreted:

- **DIRECT** (default): Reads the raw analog value, normalized to a range of [0.0, 1.0]. This mode is suitable for sensors like photoresistors, where increasing light decreases resistance, resulting in higher voltage and a higher normalized value.

  Example: Using a photoresistor in direct mode:

```
AnalogIn lightSensor(A0, DIRECT);
AnalogOut led(9);

void begin() {}

void step() {
  lightSensor >> led;
}
```

- **INVERTED**: Flips the normalized value, so high input voltage results in a low output value and vice versa. This is useful when you want the sensor to behave oppositely without changing your logic.

  Example: Inverted photoresistor reading:

```
AnalogIn lightSensor(A0, INVERTED);
```

DigitalOut and AnalogOut Modes: DIRECT and INVERTED

The *DigitalOut* and *AnalogOut* units control the flow of current and can operate in two modes:

- **DIRECT** (default): The pin provides current when ON, suitable for devices like LEDs connected between the pin and ground.

  Example: LED in direct mode. Connect the LED anode (long leg) to pin 9 and the cathode (short leg) to ground, with a 330 Ω in series.

```
AnalogOut led(9, DIRECT);
SineWave wave(1.0);

void begin() {}

void step() {
  wave >> led;
}
```

- **INVERTED**: The pin emits zero volts (GND) when ON so the current "sinks" to the pin. Suitable for digital outputs connected between a positive voltage and the pin.

  Example: LED in inverted mode. Connect the LED anode (long leg) to +5V (Vcc) and the cathode to pin 9, with a 330 Ω resistor in series.

```
AnalogOut led(9, INVERTED);
```

2.2.10 Conclusion

Understanding inputs and outputs is crucial for building interactive projects. With Plaquette's simplified syntax, tools for handling noisy signals, and powerful mapping and conditional features, you can quickly create dynamic and engaging systems. Next, we'll explore how to use Plaquette's timing and signal generation features to add even more complexity and creativity to your projects.

2.3 Generating Waveforms

In this section, we will explore waves (also called oscillators), essential tools for creating dynamic and expressive media. Oscillators generate repeating waveforms, which can control various outputs such as LEDs or motors. We will also learn how to visualize signals and shape different kinds of waveforms. We will then introduce combining different waves together, either by adding them or through modulation. Finally, we will look at how to use randomness to generate noisy waveforms that feel more natural.

Note: To follow along with the examples, set up a simple circuit:

- A **potentiometer** connected to `A0` to control proprties dynamically.
- A **button** connected to pin `2` with an internal pull-up resistor to trigger actions.
- An **LED** connected to pin 9 (PWM capable) through a 330 Ω resistor.

2.3.1 Visualizing Waves with the Serial Plotter

In this section, we will use **serial communication** to send data from our Arduino board to our PC so as to visualize the waves in real time. The `print()` and `println()` functions allow you to send data to the serial, which is invaluable for debugging and visualizing data. They will provide a way to graphically observe how wave properties like amplitude, phase, or frequency affect the output.

Single Signal

To visualize the data, open the Serial Plotter in the Arduino IDE. The Serial Plotter can graphically display waveforms by interpreting each printed value as a separate line on the graph, making it an invaluable tool to visualize signals such as sensor values and waveforms.

Example: Print the value of the potentiometer:

```
#include <Plaquette.h>

AnalogIn pot(A0); // The potentiometer

```

(continues on next page)

(continued from previous page)

```
void begin() {}

void step() {
  println(pot); // Print the potentiometer value and ends the line
}
```

Multiple Signals

For multiple signals, print their values separated by spaces in a single line, followed by a newline using `println()`.

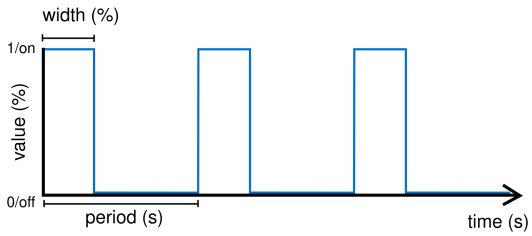
Example: Print the value of the potentiometer and a sine wave:

```
#include <Plaquette.h>

AnalogIn pot(A0);   // Potentiometer input
SineWave wave(2.0); // Sine wave with period of 2 seconds

void begin() {}

void step() {
  print(wave);  // Print wave value
  print(" ");   // Print white space
  println(pot); // Print the potentiometer value and ends the line
}
```

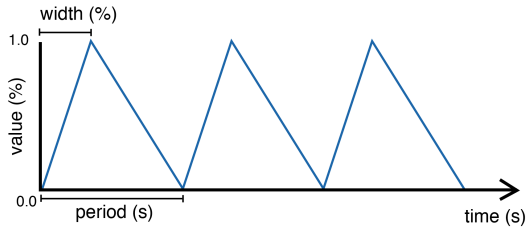
2.3.2 Types of Waves

Plaquette provides 3 types of waves:

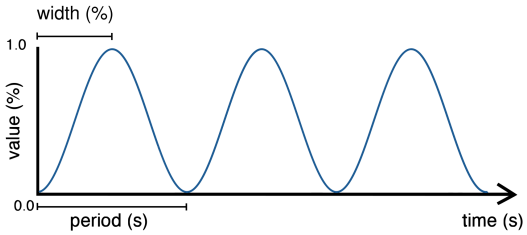
- *SquareWave*: Alternates between two levels with sharp transitions. Useful for creating rhythmic on-off patterns such as blinking LEDs or simple tone generators for buzzers. Possesses some properties of digital units.



- *TriangleWave*: Smoothly transitions between two levels in a linear fashion. By varying the width of the wave, you can create a **sawtooth wave** (width = 0) or an **inverted sawtooth wave** (width = 1). This is ideal for simulating ramping motions or gradual changes in brightness.



- *SineWave*: Produces a sinusoidal waveform for smoother modulation. Commonly used for creating natural, flowing transitions, such as smooth dimming or speed control.



You can visualize these waves on the Serial Plotter by streaming their values.

Example: Display different waves for comparison:

```
#include <Plaquette.h>

// Three wave types.
SquareWave square(1.0);
TriangleWave triangle(1.0);
SineWave sine(1.0);

void begin() {}

void step() {
  // Print all wave values separated by spaces
  print(square); print(" ");
  print(triangle); print(" ");
  println(sine);
}
```

2.3.3 Wave Properties

Oscillators are defined by their **period**, **width**, **frequency**, **amplitude**, and **phase**. Let us explore these properties and their corresponding functions:

- **period**(): Sets the duration of one cycle in seconds.
- **width**(): Controls the balance between the rising and falling portions of the wave cycle (in range [0, 1]). For each wave type, this property has a specific effect:
  - For *SquareWave*, it adjusts the duty cycle (the ratio of ON to OFF time).
  - For *TriangleWave*, it skews the wave towards a sawtooth (width = 0) or inverted sawtooth (width = 1).
  - For *SineWave*, it shifts the inflection points of the wave, altering its symmetry.
- **frequency**(): Inverse of period; sets the cycles per second (Hz).
- **bpm**(): Alternative way to set the frequency using beats per minute (BPM).
- **phase**(): Sets the initial point in the wave cycle (as % of period) (in range [0, 1]).
- **amplitude**(): Sets the peak level of the wave (as % of max) (in range [0, 1]);

Initializing Properties

The period and width of a waveform can be initialized when the unit is created.

Example: Assign period and width when creating the unit:

```
#include <Plaquette.h>

TriangleWave wave1;           // period = 1 sec (default), width = 0.5 (default)
TriangleWave wave2(2.0);      // period = 2 sec, width = 0.5 (default)
TriangleWave wave3(3.0, 0.1); // period = 3 sec, width = 0.1
```

Other properties are typically initialized in the `begin()` to build a specific waveform. It is also common to initialize period and width in the same way for more expressive code.

Example: Assign some properties of a wave at program startup:

```
#include <Plaquette.h>

TriangleWave wave;

void begin() {
  wave.frequency(2); // 2 Hz
  wave.width(0.9); // width 90%
  wave.phase(0.1); // dephased by 10% of period
  wave.amplitude(0.5); // 50% amplitude
}

void step() {
  println(wave); // Print wave value
}
```

Changing Properties During Runtime

Properties can also be changed in real-time in the `step()` function to create interactive or evolutive effects.

Example: Control the width of the waves using the potentiometer:

```
#include <Plaquette.h>

AnalogIn pot(A0); // Potentiometer input

SquareWave square(1.0);
TriangleWave triangle(1.0);
SineWave sine(1.0);

void begin() {}

void step() {
  // Assign new width value.
  square.width(pot);
  triangle.width(pot);
  sine.width(pot);
  // Print all wave values separated by spaces
  print(square); print(" ");
  print(triangle); print(" ");
  println(sine);
}
```

Example: Control the period of the waves using the potentiometer. Necessitates remapping potentiometer value to appropriate ranges.

```
#include <Plaquette.h>

AnalogIn pot(A0); // Potentiometer input

SquareWave square(1.0);
TriangleWave triangle(1.0);
SineWave sine(1.0);

void begin() {}

void step() {
  // Read new period value.
  float newPeriod = pot.mapTo(0.5, 5); // Map to 0.5-5 seconds period
  // Assign new period value.
  square.period(newPeriod);
  triangle.period(newPeriod);
  sine.period(newPeriod);
  // Print all wave values separated by spaces
  print(square); print(" ");
  print(triangle); print(" ");
  println(sine);
}
```

Try using the potentiometer to control different wave properties and visualize the result using the Serial Plotter.

Accessors and Mutators

All properties in wave units have two variants:

- A **mutator** variant allowing to change the value of the property. Example: `wave.period(3.0);`.
- An **accessor** read-only variant that returns the current value of the property. Example: `float x = wave.period();`

Tip: This naming convention is a standard in Plaquette and you will find it in other units as well.

Example: Increase the wave's period by one second each time the button is pressed:

```
#include <Plaquette.h>

DigitalIn button(2, INTERNAL_PULLUP); // Button input

TriangleWave wave(1.0); // Wave with initial 1 second period

void begin() {}

void step() {
  if (button.rose()) {
    wave.period( wave.period() + 1 ); // Set period to current period plus one
  }
  println(wave); // Print wave value
}
```

2.3.4 Wave Addition

Adding waves together allows for the creation of complex and dynamic waveforms. By superimposing multiple signals, you can simulate natural phenomena, generate rhythmic patterns, or create rich textures for artistic applications. In Plaquette, wave addition is as simple as computing the average value of different waves.

One compelling example of wave addition is simulating a **heartbeat**. A heartbeat typically has two peaks: a stronger primary beat followed by a softer secondary beat. This can be achieved by adding two waves with different amplitudes and timings.

Example: Heartbeat simulation. This example uses two *SineWave* units: one for the primary beat one for the secondary beat. The `bpm()` function sets the frequency of the waves in beats per minute.

```
#include <Plaquette.h>

SineWave primary;   // Main heartbeat wave
SineWave secondary; // Secondary beat
AnalogOut led(9);   // LED for visualizing the heartbeat

void begin() {
  primary.bpm(80); // Set primary beat to 80 beats per minute
  secondary.bpm(2*primary.bpm()); // Set secondary beat to twice primary BPM
  secondary.amplitude(0.8); // Secondary beat is less strong
}
```

(continues on next page)

(continued from previous page)

```
void step() {
  float heartBeat = (primary + secondary) / 2; // Combine and normalize waves
  led.put(heartBeat);  // Drive LED with combined signal
  println(heartBeat);  // Stream the combined wave for visualization
}
```

In this simulation, the `primary` sine wave provides the dominant rhythm, while the `secondary` sine wave introduces a softer, complementary pulse. The resulting waveform mimics the double-thump pattern of a human heartbeat.

Try experimenting with different wave types, amplitudes, and frequencies to see how the combined waveform changes. Try adding a third wave, making sure you divide the result by 3 intead of 2. Wave addition opens up endless possibilities for creating expressive and engaging outputs.

2.3.5 Modulation

Modulation involves using one oscillator to influence the properties of another, creating rich and dynamic effects. For example, a slower wave (also called a **Low-Frequency Oscillator (LFO)**) can modulate the frequency, phase, period, amplitude, or width of a faster wave.

Example: Modulate the frequency of a sine wave with a triangle wave:

```
#include <Plaquette.h>

TriangleWave modulator(10.0); // LFO (10 seconds period)
SineWave sine;    // Main wave
AnalogOut led(9); // LED output

void begin() {}

void step() {
  sine.frequency(modulator.mapTo(1.0, 10.0)); // Modulate frequency between 1 and 10 Hz
  sine >> led; // Drive LED with modulated sine wave
  println(sine); // Stream the modulated wave
}
```

2.3.6 Adding Noise with randomFloat()

While oscillators are incredibly useful for generating regular and predictable waveforms, there are times when you may want to introduce randomness to add a sense of natural variation or lifelike behavior. Plaquette provides the `randomFloat()` function, which is a powerful tool for generating random values.

> **Warning:** Avoid using Arduino's random() function as it returns integer numbers instead of floating-point numbers.

The `randomFloat()` function can be used in several ways:

- `randomFloat()` generates a random float between 0.0 and 1.0.
- `randomFloat(max)` generates a random float between 0.0 and `max`.
- `randomFloat(min, max)` generates a random float between `min` and `max`.

These random values can be used to add noise directly to a signal.

Example: Add noise to a sine wave.

```
#include <Plaquette.h>

SineWave wave(1.0); // Base waveform
AnalogOut led(9);   // LED output

void begin() {}

void step() {
  float noise = randomFloat(-0.1, 0.1); // Generate noise value in [-0.1, 0.1]
  float noisyWave = wave + noise; // Compute sine value + noise
  noisyWave >> led;   // Drive LED with noisy sine wave
  println(noisyWave); // Stream the noisy sine wave
}
```

These random values can also be used to modify properties such as amplitude, frequency, width, or phase.

Example: Update the wave's period according to a random walk. The potentiometer controls the amount of noise.

```
#include <Plaquette.h>

AnalogIn pot(A0);   // Potentiometer input
SineWave wave(1.0); // Wave with initial period of 1 second
AnalogOut led(9);   // LED output

void begin() {}

void step() {
  float noise = randomFloat(-pot, pot); // Generate noise according to potentiometer␣
↪value
  wave.period( wave.period() + noise ); // Add noise to period
  wave >> led;   // Drive LED with noisy sine wave
  println(wave); // Stream the sine wave
}
```

Example: Introduce randomness to the frequency of a triangle wave. Frequency updated on each push of the button.

```
#include <Plaquette.h>

DigitalIn button(2, INTERNAL_PULLUP); // Button input
TriangleWave wave; // Wave with default properties
AnalogOut led(9);  // LED output

void begin() {
  button.debounce(); // Debounce button
  wave.frequency(5.0); // Start at 5 Hz
}

void step() {
  if (button.rose()) {
    wave.frequency(randomFloat(4.0, 6.0)); // Random frequency between 4 and 6 Hz
  }
```

(continues on next page)

(continued from previous page)

```
  println(wave); // Stream the wave for visualization
}
```

Randomness can also be combined with modulation to create highly dynamic and expressive behaviors. Experiment with adding random noise to various properties and observe the effects using the Serial Plotter. Try to simulate a natural phenomena like a flickering flame or a lightning bolt.

2.3.7 Timing Functions

Oscillators offer various timing functions to control their behavior:

- **start()**: Starts/restarts the oscillator.
- **stop()**: Stops it and resets it.
- **pause()**: Pauses the wave at its current point.
- **resume()**: Resumes from the paused point.
- **togglePause()**: Toggles between paused and running states.
- **isRunning()**: Returns whether the oscillator is active.
- **setTime()**: Sets the current phase of the oscillator based on absolute time (in seconds).

Example: Use the button to start and stop the wave:

```
#include <Plaquette.h>

DigitalIn button(2, INTERNAL_PULLUP); // Button input
SineWave sine;    // Wave with default properties
AnalogOut led(9); // LED output

void begin() {
  sine.frequency(2.0); // Initialize frequency to 2 Hz
}

void step() {
  if (button.rose()) {
    sine.togglePause(); // Pause or resume the wave
  }
  sine >> led; // Drive LED with sine wave
  println(sine); // Stream the wave for visualization
}
```

2.3.8 Phase Shifting with shiftBy()

The `shiftBy()` function allows you to offset the phase of an oscillator relative to its current position and returns the value of the dephased wave. This is useful for creating complex, synchronized patterns.

Example: Shift the phase of a sine wave:

```
#include <Plaquette.h>

SineWave wave(5.0); // Sine wave with 5 seconds period
```

(continues on next page)

(continued from previous page)

```
void begin() {}

void step() {
  // Print shifted values separated by white spaces.
  print(wave); print(" "); // 0% shift
  print(wave.shiftBy(0.25)); print(" "); // 25% shift
  print(wave.shiftBy(0.5)); print(" "); // 50% shift
  println(wave.shiftBy(0.75)); // 75% shift
}
```

2.3.9 Conclusion

Oscillators are powerful tools for creating dynamic, expressive systems. By combining their waveforms, timing functions, and phase-shifting capabilities, you can achieve intricate and synchronized behaviors. Modulation and randomness add another layer of complexity, enabling you to create engaging and responsive media systems. Explore these features in Plaquette and see how waves can bring your projects to life.

2.4 Working with Time

In interactive systems, **timing** plays a crucial role in controlling the flow of events. Whether you are counting time, triggering events, or generating periodic signals, Plaquette provides powerful tools to manage time effectively. In this section, we will explore timing functions and units, such as *seconds()*, *Chronometer*, *Alarm*, *Metronome*, and *Ramp*.

Timing is the backbone of interactive systems. By understanding and leveraging Plaquette's timing tools, you can create precise, dynamic, creative projects that respond in real-time to various inputs and conditions.

Note: To follow along with the examples, set up a simple circuit:

- A **potentiometer** connected to `A0` to control proprties dynamically.
- A **button** connected to pin `2` with an internal pull-up resistor to trigger actions.
- An **LED** connected to pin 9 (PWM capable) through a 330 Ω resistor.

It is strongly advised to use the Serial Plotter to visualize the signals.

Warning: The value returned by timing units and functions are approximations. They are good enough for most creative applications. They should, however, not be used as a substitute for a real-time clock in applications requiring high precision, especially over long periods of time. For example, on an Arduino Uno, a drift may be experienced of up to 10 seconds per hour, or 5 minutes per day.

2.4.1 Measuring Absolute Time with seconds()

The most fundamental timing functionality in Plaquette is the *seconds()* function. As its name suggests, it simply returns the elapsed time in seconds *since the program started running*. This is very useful for measuring durations or triggering time-based events.

Example: Turn LED off after 3 seconds, then on again after 10 seconds.

```
#include <Plaquette.h>

DigitalOut led(LED_BUILTIN);

void begin() {
  led.on();
}

void step() {
  if (seconds() >= 3) { // After 3 seconds: turn LED off
    led.off();
  } else if (seconds() >= 10) { // After 10 seconds: turn LED on
    led.on();
  }
}
```

While *seconds()* provides a simple and effective way to measure time, it is inherently limited in scope. It measures only the elapsed time since the program started, functioning as a continuously increasing global counter that cannot be reset or adapted for specific events. This makes it inflexible when we need to measure time between arbitrary points or manage multiple independent timing events. For precise control and event-specific timing operations such as starting, stopping, resetting, or tracking multiple durations simultaneously, we need more refined timing instruments.

2.4.2 Timing Units

Plaquette offers a core set of specialized units to simplify common timing tasks:

- *Chronometer*: Measures elapsed time between events
- *Alarm*: Activates after a specific duration
- *Metronome*: Generates periodic pulses
- *Ramp*: Creates smooth transitions

Danger: Timing units deal with time and events without interrupting the main processing loop. Users should avoid blocking processes such as delay() and delayMicroseconds() and when using Plaquette.

Let us dive into these units and see what each one of them has to offer.

2.4.3 Keeping Track of Time with Chronometer

While *seconds()* can only give you the time since the start of the program, the *Chronometer* unit allows you to measure the time elapsed since it was started, like a real-life stopwatch. It is your basic building block for creating responsive systems where timing matters.

Chronometers are particularly useful for scenarios where the duration of an action determines its outcome. For instance, measuring how long a button is pressed can enable a system to interpret short and long presses differently.

Example: Changes LED intensity depending on how long button was pressed.

```
#include <Plaquette.h>

DigitalIn button(2, INTERNAL_PULLUP); // Button input
AnalogOut led(9); // LED output
Chronometer chrono; // Chronometer measuring button press duration

void begin() {
  button.debounce(); // Debounce button
  led.off();
}

void step() {
  if (button.rose()) {
    chrono.start(); // Start the timer when button is pressed
  }

  else if (button.fell()) {
    // Converts chronometer time to LED intensity over a range of 10 seconds
    float ledValue = mapTo01(chrono, 0, 10); // Maps from 0-10 seconds to [0, 1] range
    ledValue >> led;
    chrono.stop(); // Stops/resets the timer when button is released
  }

  println(chrono); // Prints value of chrono for visualization.
}
```

The *Chronometer* is great for counting time. In many scenarios, however, you want to know whether you waited for a certain amount of time. The *Alarm* unit provides a convenient way to do so.

2.4.4 Scheduling with Alarm

Like a real-world alarm-clock, the *Alarm* unit starts "buzzing" after a predefined time. This **digital unit** is initialized with a certain duration. It outputs 0/false until it reaches its timeout; then, it starts "ringing" and outputs 1/true until it is stopped or restarted.

Once triggered, it can be stopped by calling its `stop()` function, or restarted by calling `start()`, making the unit ideal for implementing delayed responses or timed sequences.

Alarms can help manage actions that require specific timing, such as turning off a light after a certain duration or triggering an animation. Their flexibility makes them a powerful tool in time- based designs.

Example: Starts blinking an LED when we reach the alarm's timeout. Pushing the button restarts the alarm, increasing its duration by 50% each time.

```
#include <Plaquette.h>

DigitalOut led(LED_BUILTIN); // LED on built-in pin
DigitalIn button(2, INTERNAL_PULLUP); // Button input

SquareWave blink(0.5); // Wave to blink LED when alarm is buzzing

Alarm alarm(2.0); // Alarm with 2s duration

void begin() {
  button.debounce(); // Debounce button
}

void step() {
  // Button: restart.
  if (button.rose()) {  // Button pressed event
    led.off();          // Turn off LED
    alarm.duration( alarm.duration() * 1.5 ); // Increase duration by 50%.
    alarm.start();      // Start alarm
  }

  // Alarm buzzing: blink LED.
  if (alarm) {    // Check if alarm is buzzing
    blink >> led; // Blink LED
  }

  println(alarm.progress()); // % progress of the alarm (for visualization)
}
```

2.4.5 Triggering Periodic Events with Metronome

While the *Alarm* unit is great for dealing with one-time events, there are many cases where an action needs to be triggered periodically. For such use cases, Plaquette provides the *Metronome* unit which sends a periodic pulse or "bang". In other words, it acts like an *Alarm* that gets restarted as soon as it starts buzzing. It also bears some resemblance with *wave units*.

Periodic actions are at the core of interactive systems, whether you are blinking an LED or synchronizing motor movements. The *Metronome* provides a straightforward way to create these kinds of repetitions.

Example: Blink an LED using a Metronome:

```
#include <Plaquette.h>

DigitalOut led(LED_BUILTIN); // LED on built-in pin
Metronome metro(1.0); // Metronome with period of 1 second

void begin() {}

void step() {
  if (metro) { // The unit will be true for a single frame every time it triggers
    led.toggle(); // Toggle LED on each pulse
  }
}
```

Metronome units can be used as a way to trigger different actions in parallel.

Example: Use multiple *Metronome* units to control different actions. One metronome toggles LED visibility, while another slower metronome accelerates blinking speed at each tick.

```
#include <Plaquette.h>

DigitalOut led(LED_BUILTIN); // LED on built-in pin
SquareWave blink(1.0); // Wave to blink the LED
Metronome metroToggle(2.0); // Metronome to toggle visibility
Metronome metroAccelerate(10.0); // Metronome to accelerate blink

boolean visible = true; // Flag to keep track of visibility

void begin() {}

void step() {
  // Toggle visibility.
  if (metroToggle) {
    visible = !visible; // Invert boolean value
  }

  // Accelerate blink.
  if (metroAccelerate) {
    blink.frequency( blink.frequency() * 2 ); // Double frequency
  }

  // Activate LED depending on visibility status.
  if (visible)
    blink >> led;
  else
    led.off();
}
```

2.4.6 Creating Smooth Transitions with Ramp

Ramps are a cornerstone of creative expression. Unlike oscillators like *TriangleWave* and *SineWave*, which generate periodic signals, ramps interpolate from one value to another over a specific duration or at a specific speed. The *Ramp* unit in Plaquette provides a flexible and powerful way to animate visual elements such as LEDs or physical components such as motors in a natural manner, allowing the creation of rich, dynamic, evolving experiences.

Tip: We strongly recommend to use the Serial Plotter to visualize the ramp values in the following examples.

Basic Usage

Like *Alarm* units, ramps can be restarted by calling their `start()` function. By default, they will ramp between 0 and 1.

Example: Gradually increases an LED brightness over 5 seconds every time a button is pressed.

```
#include <Plaquette.h>

DigitalIn button(2, INTERNAL_PULLUP); // Button input
AnalogOut led(9); // LED output
Ramp ramp(5.0);   // Ramp with 5 seconds duration

void begin() {
  button.debounce(); // Debounce button
  ramp.start(); // Initial ramp startup
}

void step() {
  if (button.rose()) {
    ramp.start(); // Restart ramp
  }

  ramp >> led; // Use ramp value to control LED brightness
  println(ramp); // Visualize ramp value with the Serial Plotter
}
```

Try changing the behavior of the ramp to rather go from 1 to 0 by calling the `fromTo()` function and see how that changes the behavior of the ramp:

```
void begin() {
  ramp.fromTo(1.0, 0.0); // Ramp from one to zero
  ramp.start();
}
```

Flexible Ranges

Ramps are not restricted to the range [0, 1]. You can define any starting and ending values, making ramps very useful for various applications such as changing properties of waves, controlling the angle of a servo motor, adjusting the color of a RGB LED, etc.

Example: Gradually increases an LED brightness over a 5 seconds period every time a button is pressed. The potentiometer sets the maximum LED value to attain.

```
#include <Plaquette.h>

DigitalIn button(2, INTERNAL_PULLUP); // Button input
AnalogIn pot(A0); // Potentiometer input
AnalogOut led(9); // LED output
Ramp ramp(5.0); // Ramp with 5 seconds duration

void begin() {
  button.debounce(); // Debounce button
}
```

(continues on next page)

(continued from previous page)

```
void step() {
  if (button.rose()) {
    ramp.to(pot); // Set ramp goal to value of potentiometer
    ramp.start(); // Restart ramp
  }

  ramp >> led; // Use ramp value to control LED brightness
  println(ramp); // Visualize ramp value with the Serial Plotter
}
```

Try adjusting the potentiometer to different positions and then pressing the button to see the effect.

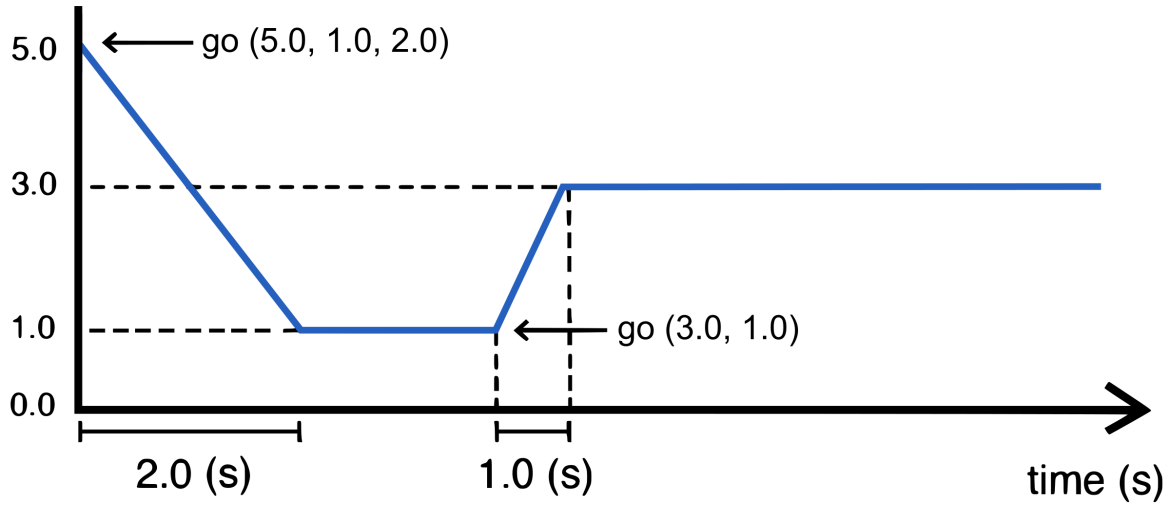
Notice how we are using function `to()` to set the goal of the ramp. The starting value is left unchanged at zero (default value). To change the starting value while preserving the goal value, use function `from()` instead. See what happens if you change the call `ramp.to(pot)` to use `from()` intead:

```
ramp.from(pot); // Set ramp goal to value of potentiometer
```

Dynamic Control with go()

A common scenario in creative applications is to respond to events by changing a value such as the position of a servomotor, the color of a RGB LED, or the volume of a sound. Ramps are often used in these cases to create smooth transitions instead of abrupt changes.

The `go()` function provides a simple way to immediately launch a ramp from one value to another, or simply from the current value towards a new goal.



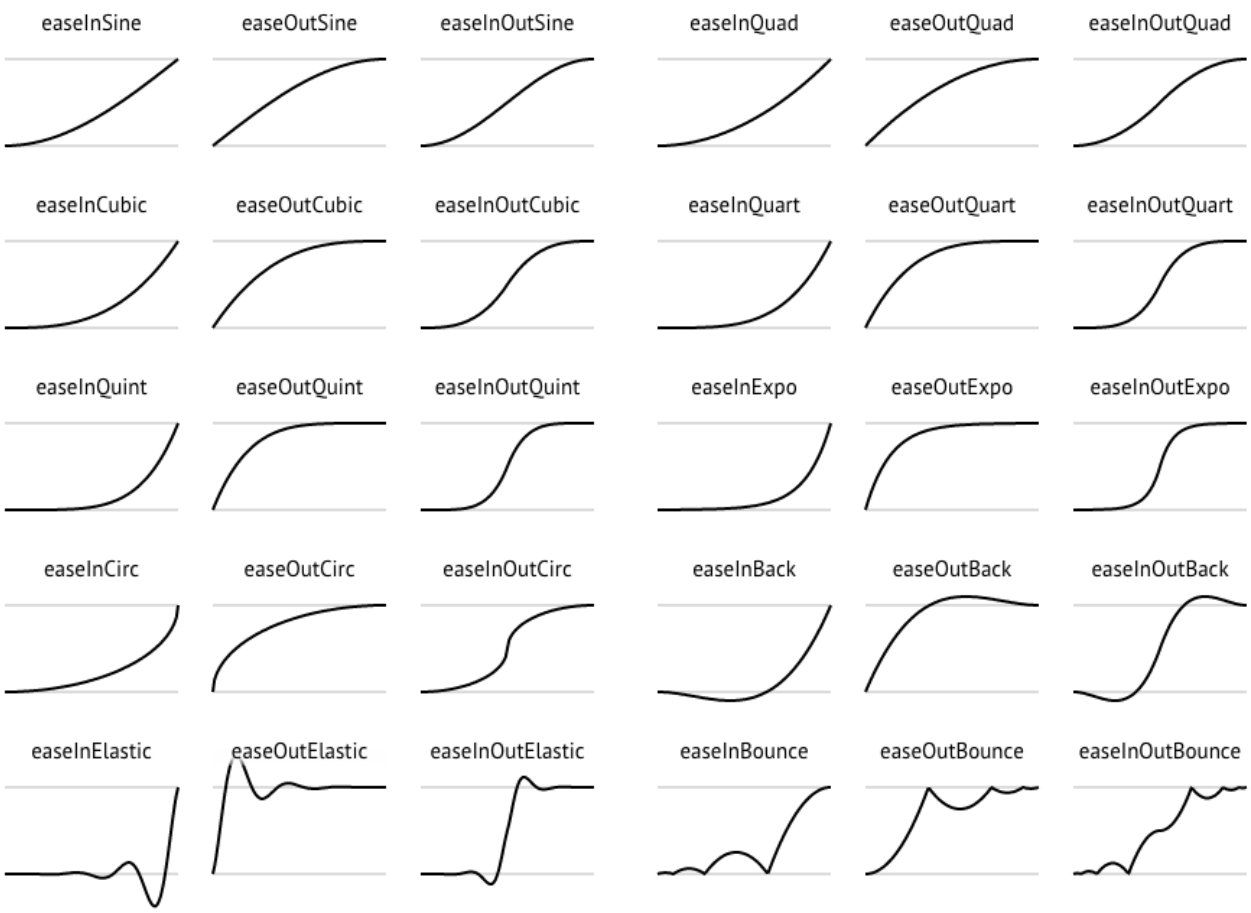
Example: Control blinking frequency using a button. Each time the button is pushed, a new frequency is chosen randomly and the ramp smoothly goes to the new frequency.

```
#include <Plaquette.h>

DigitalIn button(2, INTERNAL_PULLUP); // Button input
AnalogOut led(9);  // LED output
Ramp ramp(5.0);    // Ramp with 5 seconds duration
TriangleWave wave; // Oscillator

void begin() {
  wave.width(1.0);   // Sawtooth wave
  wave.bpm(100);     // Initial BPM
  button.debounce(); // Debounce button
}

void step() {
  if (button.rose()) {
    // Set target BPM to random value
    float targetBpm = randomFloat(60, 200);
    ramp.go(targetBpm); // Launch ramp
  }

  wave.bpm(ramp); // Use ramp value to adjust BPM of wave

  wave >> led;   // Oscillate LED
  println(ramp); // Visualize ramp value with the Serial Plotter
}
```

Note: Ramps provide multiple ways to call `go()` depending on the desired behavior, including specifying starting value and duration on the spot. For more details, please consult the *Ramp unit's reference*.

Generating Expressive Effects with Easing Functions

Ramp supports *easing function*, providing many different ways to generate expressive effects. Easing functions add acceleration or deceleration effects to ramp transitions, making them feel more natural and lifelike.



Example: Use easing to create a smooth LED fade repeatedly:

```
#include <Plaquette.h>

AnalogOut led(9); // LED output
Ramp ramp(3.0); // Ramp with 3 seconds duration

void begin() {
  ramp.easing(easeInOutQuad); // Apply an easing function
  ramp.start();
}

void step() {
  if (ramp.isFinished())
    ramp.start(); // Restart the ramp with the easing effect
  }

  ramp >> led;   // Use the ramp's value to control the LED brightness
  println(ramp); // Visualize ramp value with the Serial Plotter
}
```

Try experimenting with different easing functions and observe the results on the LED and using the Serial Plotter. Easing can transform mechanical transitions into expressive animations, giving your projects character.

Operational Modes: Duration vs Speed

By default, ramps transition between two values over a definite duration. However, there are many scenarios where this is not the appropriate behavior. For example, one might want to move a servomotor at a specific angular speed: ramping over 10 degrees should take much less time than a 90 degrees transition.

Ramps accomodate these different use cases by providing two modes of operation:

- In **duration mode** (default) the ramp transitions between values over a fixed number of seconds.
- In **speed mode** the ramp moves at a constant rate, defined in value change per second.

Example: Compare duration and speed modes. Ramp values can be visualized using the Serial Plotter.

```
#include <Plaquette.h>

DigitalIn button(2, INTERNAL_PULLUP); // Button input
Ramp rampDuration; // Ramp operating in duration mode
Ramp rampSpeed;    // Ramp operating in speed mode

void begin() {
  rampDuration.duration(5.0); // Duration: 5 seconds
  rampSpeed.speed(5.0); // Rate of change: 5 per second
  button.debounce(); // Debounce button
}

void step() {
  if (button.rose()) {
    // Both ramps go to random target value.
    float targetValue = randomFloat(-20, 20);
    rampDuration.go(targetValue);
    rampSpeed.go(targetValue);
  }

  // Visualize and compare ramps with the Serial Plotter
  print(rampWithDuration);
  print(" ");
  println(rampWithSpeed);
}
```

Tip: To switch between modes, you can simply call the `duration(value)` or `speed(value)` functions with a target duration or speed (recommended). Alternatively, you can change mode by calling `mode(RAMP_DURATION)` or `mode(RAMP_SPEED)`, in which case the duration or speed will be computed based on the ramp's current properties (ie. duration/speed, starting, and target values).

2.4.7 Combining Timing Units

Plaquette allows you to combine different timing units to achieve complex behaviors while keeping your workflow clear and intuitive. For instance, you can use a *Metronome* to repeatedly trigger a *Ramp* or synchronize multiple timing units.

Example: Use a Metronome to trigger a Ramp at regular intervals:

```
#include <Plaquette.h>

Metronome metro(10.0); // Trigger every 10 seconds
Ramp ramp(3.0);   // Ramp with 3 seconds duration
AnalogOut led(9); // LED output

void begin() {}

void step() {
  if (metro) {
    ramp.start(); // Start the ramp each time the metronome triggers
  }

  ramp >> led; // Use the ramp's value to control the LED brightness
  println(ramp); // Stream the ramp's value for visualization
}
```

Combining timing units unlocks an even greater range of creative possibilities. Use these tools to design intricate behaviors, smooth transitions, and expressive animations in your projects.

2.4.8 Conclusion

Timing is an essential aspect of creating interactive and dynamic systems, and Plaquette provides an intuitive set of tools to make this process seamless. From measuring durations with the *Chronometer*, to triggering events with the *Alarm*, generating rhythmic patterns with the *Metronome*, and creating smooth transitions with the *Ramp*, each timing unit offers unique possibilities.

The flexibility of these tools allows for countless creative applications, whether you are developing reactive systems, synchronizing events, or designing natural and expressive transitions. By combining these units, you can build intricate behaviors that bring your projects to life.

2.5 Regularizing Signals

Plaquette provides expressive, automated, and robust ways to deal with signals for interactive design using **regularization filters** such as smoothing, min-max scaling, and normalization.

2.5.1 Direct Input-to-Output

Let's review briefly how to handle raw *input and output* signals in Plaquette. We will be using an analog sensor such as a photoresistor for this example.

Note: In order to build this circuit, you will need to create a simple voltage divider circuit. Connect the photoresistor between the ground (GND) and the analog input pin (`A0`). Then connect a fixed resistor with value matching your photoresistor between analog input pin and +5V (Vcc). For example, for a 1k Ω - 10k Ω photoresistor you could use a fixed resistor of about 5.5k Ω).

Here is a simple Arduino code that allows one to change the value of an output LED using an input photocell:

```
// The photocell analog pin.
int photoCellPin = A0;

// The output analog LED pin.
int ledPin = 9;

void setup() {
  // Initialize pins.
  pinMode(photoCellPin, INPUT);
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // Read value from photocell (between 0 and 1023).
  int value = analogRead(photoCellPin);

  // Write value to LED (between 0 and 255).
  analogWrite(ledPin, value / 4);
}
```

As explained in *Why Plaquette?* section, this simple code is made complicated by the fact that the programmer needs to remember low-level information concerning the ranges of raw number values (1023, 255, ...) Furthermore, this code fails to adapt to changing conditions such as the range of the ambient light.

Let's see how Plaquette can help us to create more expressive code by using inputs and outputs signals rather than meaningless raw numbers.

To begin, we will re-implement the example above using Plaquette units.

First, let's define our input photocell on pin `A0` using an *AnalogIn* unit:

```
AnalogIn photoCell(A0);
```

Then, let's add an output analog LED on pin *9* using an *AnalogOut* unit:

```
AnalogOut led(9);
```

If we want to directly control the value of the LED from the value of the photocell, all we need to do is to send the photocell's value to the led. The easiest way to do so is by using the >> operator:

```
photoCell >> led;
```

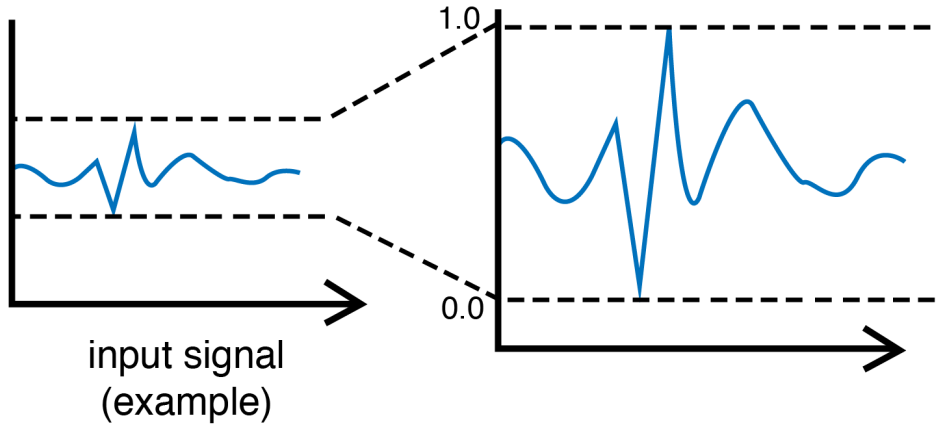
The complete Plaquette code will look like this:

```
#include <Plaquette.h> // include the Plaquette library

// Create input unit for photocell.
AnalogIn photoCell(A0);

// Create output unit for LED.
AnalogOut led(9);

void begin() {}

// Define frame-by-frame operations.
void step() {
  // Send photocell value directly to the LED.
  photoCell >> led;
}
```

2.5.2 Getting the Full Range of a Signal

If we run this program, we will likely notice that the LED brightness will not span the full range from 0% to 100%. That's because depending on ambient lighting conditions, the photocell's values will not move across the full spectrum of possibility. For instance, in the dark, the photocell might range from 10% to 50%, while in full daylight, it might range between 70% and 95%.

In order to resolve this issue, we need to **regularize** the photocell's signal. We can do so using a filtering unit such as a *MinMaxScaler*. This unit automatically keeps track of the minimum and maximum values of the incoming signal over time (for example, 10% and 50%) and remaps them into a new interval of [0, 1] (ie., 0% to 100%).



To use this approach, create the unit:

```
MinMaxScaler regularizer;
```

... and then *insert it* in the pipeline between the incoming photocell signal and the output LED:

```
photoCell >> regularizer >> led;
```

The above expression will do the following, in order:

1. Read the raw photocell value using the `photoCell` unit.
2. Send that raw value from the `photoCell` unit to the `regularizer` unit.
3. The `regularizer` unit updates itself if the value is a new extreme value (minimum or maximum).
4. The `regularizer` then remaps the raw photocell value to the full range of [0, 1] and sends it to the `led` unit.
5. The `led` unit takes the input value in [0, 1] and applies it to the intensity of the LED.

2.5.3 Reacting to Signal Changes

Remember our example from *ealier*, where we were trying to detect high-valued signals using arbitrary numbers?

```
if (value > 716)
  // do something
```

Suppose that instead of directly controlling the LED value based on the photocell's value, we instead want to use sudden changes in the photocell's value to trigger the on/off state of the LED? In other words, we would like to work with the **peaks** in the incoming signal (such as when someone points a light source towards the photocell).

One way to do so would be to pick a threshold in the regularized signal above which we would react to the light source. Let's say that we will react when the signal goes above 70%. The code of the `step()` function now becomes:

```
void step() {
  photoCell >> regularizer;
  if (regularizer > 0.7)
    1 >> led;
  else
    0 >> led;
}
```

... which can be more compactly rewritten by sending directly the conditional expression `(regularizer > 0.7)` to the output LED:

```
void step() {
  photoCell >> regularizer;
  (regularizer > 0.7) >> led;
}
```

2.5.4 Adapting to Changing Conditions

So far so good. The number 0.7 is still a bit of an arbitrary, hand-picked number, but it makes more sense than 716 because it refers to a more human-understandable concept (70% instead of 716 / 1023). However, this approach will still be sensitive to changes in the ambient light, and behave differently under different light conditions (for example, it might work as expected in the morning, but work less well in the late afternoon when the sun starts to go down.)

One thing we could do would be to make sure that our regularization unit adapts to changing conditions. In order to do this, rather than having our MinMaxScaler remap values depending on every single incoming value, we can have it adapt over a **time window**. This will allow our regularizer to slowly forget what it has learned, and reprogram itself after a certain amount of time has passed.

This can be accomplished by calling the `timeWindow(seconds)` function inside the `begin()` function:

```
void begin() {
  // Allow regularizer to adapt over an approximate period of 1 hour (3600 s).
  regularizer.timeWindow(3600.0f);
}
```

2.5.5 Normalizing Signals to Spot Extreme Values

The MinMaxScaler is a very useful unit for making sure signals stay within a [0, 1] range. However, it is not always the best for signal detection since it only accounts for extreme values (minimum and maximum), which makes it sensitive to rare events. Someone switching the lights on and off again rapidly might completely ruin the show.

A better alternative is the *Normalizer* unit, which regularizes incoming signals by normalizing them around a target **mean** by taking into account **standard deviation**. Once the data is normalized, extreme **outlier** values can be more easily and robustly detected based on how much they diverge from the mean.

Let's replace our MinMaxScaler by a Normalizer unit:

```
Normalizer regularizer;
```

... and use the `isHighOutlier()` function to find values that are higher than usual:

```
void step() {
  photoCell >> regularizer;
  regularizer.isHighOutlier(photoCell) >> led;
}
```

Tip: By default, the `isHighOutlier()` function detects values that are more than 1.5 deviations from the mean. The function can be made more or less sensitive by adjusting the number of deviations (typically between 1.0 and 3.0). For example, `isHighOutlier(value, 1.2)` will be more sensitive, `isHighOutlier(value, 2.5)` will be less sensitive, and `isHighOutlier(value, 3.0)` will only respond to rarely-occuring extremes. While these numbers (1.2, 1.5, 2.5, etc.) still need to be hand-picked, they are much more robust than our 716 and even to our 0.7 number from earlier.

Here is a complete version of the code:

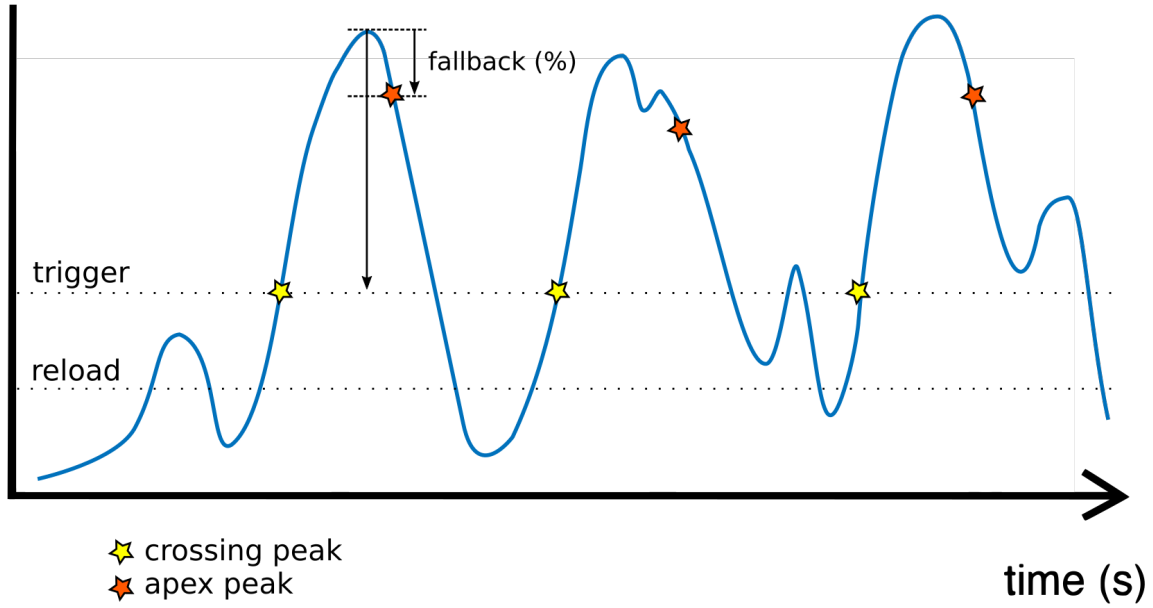
```
#include <Plaquette.h> // include the Plaquette library

// Create input unit for photocell.
AnalogIn photoCell(A0);

// Create output unit for LED.
AnalogOut led(9);

// Create regularization object.
Normalizer regularizer;

// Initialize everything.
void begin() {
  // Allow regularizer to adapt over an approximate period of 1 hour (3600 s).
  regularizer.timeWindow(3600.0f);
}

// Define frame-by-frame operations.
void step() {
  // Update regularizer with raw signal value.
  photoCell >> regularizer;

  // Detect outliers and send the value (1=true=outlier, 0=false=no outlier)
  // directly to the LED.
  regularizer.isHighOutlier(photoCell) >> led;
}
```

2.5.6 Detecting Peaks

The outlier detection method is useful to find extreme values. However, it also comes with an important limitation. The `isHighOutlier()` and `isOutlierLow()` methods return `true` *as long as* the received value is considered to be an outlier, making these methods unsuitable for triggering instantanous events, such as toggling the status of an LED, starting a sound event, activating a motor, etc.

The *PeakDetector* unit addresses this limitation. It is best used in combination with a Normalizer unit. We will use the default mode of the PeakDetector (`PEAK_MAX`): for a peak to be detected. In this mode, the signal will need to (1) cross a *trigger threshold* value (`triggerThreshold`); (2) reach its *apex* (max); and (3) *fall back* by a certain proportion (%)

between the threshold and the apex (controlled by the `fallbackTolerance` parameter).



Building on the previous section for outlier detection, we will assign the PeakDetector's `triggerThreshold` to the value above which a value is considered to be a high outlier, which can be obtained by calling the Normalizer's function `highOutlierThreshold()`:

```
PeakDetector detector(normalizer.highOutlierThreshold());
```

Tip: As for the `isHighOutlier()` function, the `highOutlierThreshold()` function is set to return, by default, a threshold that is 1.5 standard deviations from the mean. The function can be made more or less sensitive by adjusting the number of deviations. For example, `highOutlierThreshold(1.2)` will be more sensitive, while `highOutlierThreshold(2.5)` will be less sensitive.

Finally, let's rewrite the `step()` function with our new peak detector, so that only when a **peak** is detected will the LED change state:

```
void step() {
  // Signal is normalized and sent to peak detector.
  sensor >> normalizer >> detector;

  // Toggle LED when peak detector triggers.
  if (detector)
    led.toggle();
}
```

The PeakDetector unit offers many options to fine-tune the peak detection process. Please read the *full documentation of the unit* for details.

2.5.7 Conclusion

The Plaquette library simplifies signal processing for interactive design by abstracting low-level details and offering intuitive regularization tools like *MinMaxScaler* and *Normalizer*. Combined with *PeakDetector* opens the way to deploy precise event-driven behaviors.

Plaquette's ability to adapt to changing conditions ensures dynamic, robust systems while keeping code concise and expressive. By leveraging its modular architecture, users can streamline signal handling, improve scalability, and focus on innovation in signal-driven creative applications.

2.6 Managing Events

Plaquette supports event-driven programming, allowing you to execute specific actions automatically when an event occurs instead of constantly checking for changes with conditional statements such as `if (button.changed())`. In Plaquette, this is achieved by **events** and function **callbacks**.

An **event** is an instantaneous situation that is triggered by a unit under specific conditions, such as the push of a button, the tick of a metronome, the end of a timer, or a peak detection.

A **callback** is a custom function that is registered with a source event: when the event is triggered, the registered callback is automatically called.

Note: In programming, a **callback function** is like giving someone instructions on what to do when a specific event happens. For example, imagine you are baking cookies and you set a timer. Instead of constantly watching the oven, you set the timer to "call you back" (in other words, alert you) when time is up, so you can take the cookies out.

This approach offers several advantages:

- **Modularity**: Encapsulates behavior in reusable functions.
- **Expressiveness**: Focuses on declaring "what happens when".
- **Efficiency**: Reacts to changes without continuous polling.
- **Scalability**: Makes it easy to manage complex systems with multiple events.

Let us explore how this works with practical examples.

2.6.1 Supported Events

Event:	`onRise()`	`onFall()`	`onChange()`	`onBang()`	`onFinish()`	`onUpdate()`
Activation:	Value rises	Value falls	Value changes	Unit fires	Time out	New value
Alarm	✓	✓	✓		✓	
DigitalIn	✓	✓	✓			
Metronome				✓		
PeakDetector				✓		
Ramp					✓	
SquareWave	✓	✓	✓			
StreamIn						✓

2.6.2 Reacting to an Event

Let us take an example where we want to react to the push of a button by switching an LED on and off.

First, let us create the units we will be working with:

```
#include <Plaquette.h>

DigitalOut led(LED_BUILTIN); // LED connected to built-in pin
DigitalIn button(2, INTERNAL_PULLUP); // Button connected to pin 2
```

In order to react to an event, we first need to create a callback function which will be called when the event will happen:

```
// Callback function to toggle the LED.
void toggleLed() {
  led.toggle();
}
```

Then, we need to register our callback to an event. In this case, we will register our function `toggleLed()` to the `onRise()` event of our button unit, which will trigger at the instant the button is pressed.

```
void begin() {
  button.debounce(); // Enable debouncing to avoid multiple events

  // Register callbacks for button events.
  button.onRise(toggleLed); // Toggle the LED on button press
}
```

In this case, since the callback will take care of all the logic, the `step()` function can be left empty!

```
void step() {} // Nothing to do here!
```

Here is the final code for this example:

```
#include <Plaquette.h>

DigitalOut led(LED_BUILTIN); // LED connected to built-in pin
DigitalIn button(2, INTERNAL_PULLUP); // Button connected to pin 2

// Callback function to toggle the LED.
void toggleLed() {
  led.toggle();
}

void begin() {
  button.debounce(); // Enable debouncing to avoid multiple events

  // Register callbacks for button events.
  button.onRise(toggleLed); // Toggle the LED on button press
}

void step() {} // Nothing to do here
```

Now, try changing `onRise()` to `onFall()` or to `onChange()`. How does that affect the interaction between the button and the LED?

2.6.3 Managing Multiple Events

It is possible to register multiple callbacks with the same event. Likewise, a single callback can be registered with many events.

Example: Launch both `toggleLed()` and `printButton()` on button press, registering `printButton()` to both press and release events.

```
#include <Plaquette.h>

DigitalOut led(LED_BUILTIN); // LED connected to built-in pin
DigitalIn button(2, INTERNAL_PULLUP); // Button connected to pin 2

// Callback function to toggle the LED.
void toggleLed() {
  led.toggle();
}

// Callback function to print button state.
void printButton() {
  print("Button ");
  println(button ? "pressed" : "released");
}

void begin() {
  button.debounce(); // Enable debouncing to avoid multiple events

  // Register callbacks for button events.
  button.onRise(toggleLed); // Toggle the LED on button press

  button.onRise(printButton); // Print button state
  button.onFall(printButton); // Same here
}

void step() {} // Nothing to do here
```

2.6.4 Coordinating Parallel Events with Metronomes

There are many applications for which things happen concurrently at different pace, making one wish there could be multiple looping functions similar to `step()` running in parallel at different rates. This is easy to achieve in Plaquette using event-driven coding. Metronomes tick at a specific period, generating "bang" events which can trigger callbacks by registering them to the `onBang()` event.

In this example, two metronomes control two LEDs, one digital and one analog, each at a different interval. A ramp is used to fade the analog LED.

```
#include <Plaquette.h>

DigitalOut led1(LED_BUILTIN); // First LED (digital) connected to built-in pin
AnalogOut led2(9);            // Second LED (PWM) connected to pin 9
Metronome metro1(1.0);        // Metronome with a 1 second period
Metronome metro2(2.0);        // Metronome with a 2 seconds period
Ramp rampLed(0.5);            // Short ramp to control LED 2
```

(continues on next page)

(continued from previous page)

```
// Function to toggle the first LED.
void pingLed1() {
  led1.toggle();
}

// Function to start the ramp on second LED.
void pingLed2() {
  ramp.start();
}

void begin() {
  // Register callbacks for the metronomes.
  metro1.onBang(pingLed1); // Toggle LED 1 every second
  metro2.onBang(pingLed2); // Fade in LED 2 every 2 seconds
}

void step() {
  ramp >> led2; // Ramp second LED from 100% to 0%
}
```

2.6.5 Creating On-the-fly Callbacks

For simple, localized actions, you can define callback functions directly inline using an **anonymous function** (also called **lambda function**) which can be created with the following syntax:

```
[]() {
  // Function content goes here.
}
```

It allows you to write concise code without defining separate named functions and is thus especially useful for short, self-contained actions, keeping the code clean and readable.

For example, we could rewrite the callback registration from the example above in a shorter way, like this:

```
void begin() {
  // Register callbacks for the metronomes.
  metro1.onBang([]() { led1.toggle(); }); // Toggle LED 1 every second
  metro2.onBang([]() { ramp.start(); });  // Fade in LED 2 every 2 seconds
}
```

2.6.6 Conclusion

Event-driven programming in Plaquette simplifies the process of reacting to changes and scheduling actions, allowing you to write modular, expressive, and efficient code. By using callbacks and event sources like buttons and metronomes, you can manage complex behaviors that happen concurrently and at different rates.

2.7 Advanced Usage

2.7.1 Smoothing Arbitrary Signals

While the *AnalogIn* unit in Plaquette provide a convenient built-in `smooth()` function for removing noise, there are many cases where you may need to smooth signals coming from other sources such as specialized sensors. The *Smoother* unit provides a highly flexible smoothing solution for such use cases, allowing seamless integration into any signal pipeline. It works using an exponential moving average, acting as a low-pass filter to stabilize fast variations.

Here is an example of using the *Smoother* to smoothen a DHT 22 temperature and humidity sensor using the external DHT sensor library:

```
#include <Plaquette.h>
#include <DHT.h> // External specialized library

DHT dht(2, DHT22); // DHT 22 sensor connected to pin 2

// Create a Smoother with a 10-second time window.
Smoother temperatureSmoother(10.0);

// Stream out for debugging (e.g., to Serial Monitor).
StreamOut serialOut(Serial);

void begin() {
  dht.begin();  // Initialize the DHT sensor
}

void step() {
  // Read temperature in Celsius.
  float rawTemperature = dht.readTemperature();

  // Smooth the temperature and send it to the Serial.
  rawTemperature >> temperatureSmoother >> serialOut;
}
```

2.7.2 Vanilla Coding Style

You can avoid Plaquette's `>>` operator or auto-conversion of units to values (eg., `if (input)`, `input >> output`) in favor of a more conventional programming style by simply using the `get()` and `put()` functions of Plaquette units.

The `get()` method returns the current value of the unit:

```
float get()
```

The `put()` method sends a value to the unit and then returns the current value of the unit (the same that would be returned by `get()`):

```
float put(float value)
```

Additionally, digital input units such as *DigitalIn*, *Metronome* have a `boolean isOn()` method that works for boolean `true/false` values, while digital output units such as *DigitalOut* have a `boolean putOn(boolean value)` method.

Here are some examples of how to adopt a classic object-oriented functions style instead of the Plaquette style.

Plaquette Style	Object-Oriented Style
`input >> output;`	`output.put(input.get());`
`digitalInput >> digitalOutput;`	`digitalOutput.putOn(digitalInput.isOn());`
`(2 * input) >> output;`	`output.put(2 * input.get());`
`!digitalInput >> digitalOutput;`	`digitalOutput.putOn(!digitalInput.isOn());`
`if (digitalInput)`	`if (digitalInput.isOn())`
`if (input < 0.4)`	`if (input.get() < 0.4)`
`input >> filter >> output;`	`output.put(filter.put(input.get()));`

2.7.3 Using Plaquette as an External Library

Seasoned Arduino coders might want to avoid rewriting their code using Plaquette's builtin `begin()` and `step()` functions, or they may want to include Plaquette's self-updating loop in a timer interrupt function. It is possible to do so by including the file `PlaquetteLib.h` instead of `Plaquette.h`.

After this step, you is then responsible for calling `Plaquette.begin()` at the beginning of the `setup()` function, and also to call `Plaquette.step()` at the beginning of the `loop()` function, or inside the interrupt.

Here is an example of our blinking code rewritten by using this feature:

```
#include <PlaquetteLib.h>

using namespace pq;

DigitalOut myLed(13);

SquareWave myWave(2.0, 0.5);

void setup() {
  Plaquette.begin();
}

void loop() {
  Plaquette.step();
  myWave >> myLed;
}
```

2.7.4 Synchronizing Groups of Units with Secondary Engines

Have you wondered how units such as waves, inputs, outputs, or ramps are automatically initialized and updated in Plaquette? This is done thanks to an *Engine*, a control structure that acts like the **conductor of an orchestra**. It contains an ensemble of **units** and manages their initialization and updates. Every time the engine "ticks", it updates all of its units, making sure they stay synchronized.

By default, all units are added to a built-in engine called the **primary engine**. This engine is simply called `Plaquette` and is mainly used when working with *Plaquette as an external library*. In the default mode, when one declares the `void begin()` and `void step()` functions, the primary engine's `Plaquette.begin()` and `Plaquette.step()` functions are automatically called.

There are many contexts where more than one engine are necessary:

- **Multi-tasking** Engines allow you to take full advantage of **timer interrupts**, **threads** and/or **multiple processor cores** to run different unit ensembles in parallel, possibly running with different frequencies and priorities.

- **Grouping** Engines can be used to better organize your code by creating **groups of units** and possibly run them at different frequencies.
- **Switching** On computationally-intensive applications with lots of units, you may want to **switch between multiple ensembles of units** to avoid running them all at the same time.
- **Saving Energy** Lowering the update frequency of units using engines allows for more energy-efficient applications.

In these cases, you can create **secondary engines** that each control their own group of units at their own refresh rate. You can step them in a timer interrupt, in a task running on another core, or even from a *Metronome* unit.

To create an engine, simply declare it:

```
Engine secondaryEngine;
```

When you create a unit, you can now add it to your new engine by adding the engine as the last argument of the unit's constructor:

```
// Ramp starting at default value.
Ramp ramp(secondaryEngine);

// Alarm unit with 10s duration.
Alarm alarm(10.0, secondaryEngine);

// Square wave unit with period of 1s and 20% width.
SineWave wave(1.0, 0.2, secondaryEngine);
```

Example: Fast vs Slow Control

In this example we will control a blinking LED with a pushbutton: every time we push the button, the LED will blink faster and faster. The button should be polled quite frequently to ensure it is debounced properly. However, there is no need to update the LED as fast as possible, so we can save some precious computation steps by updating it less often (about 25 frames per second would be plenty for such visual feedback).

- The **slow engine** (running at 25 fps) will control the LED with the square wave.
- The **fast engine** (running at 1000 Hz) will monitor the button.
- The **primary engine** (running as fast as possible) will use *Metronome* units to trigger the slow and fast engines.

```
#include <Plaquette.h>

// The engines.
Engine slowEngine;
Engine fastEngine;

// Metronomes (belong to primary engine).
Metronome slowMetro(0.04);  // 25 fps
Metronome fastMetro(0.001); // 1000 Hz

DigitalIn button(2, INTERNAL_PULLUP, fastEngine); // Button (operates on fast engine).

// Oscillator and LED (can operate more slowly to save on computation).
SquareWave squareWave(1.0, slowEngine);
DigitalOut led(LED_BUILTIN, slowEngine);
```

(continues on next page)

(continued from previous page)

```
float ledFrequency = 1.0; // Oscillation frequency.

void begin() {
  // Initialize engines.
  slowEngine.begin();
  fastEngine.begin();
  button.debounce(); // Debounce button.
  // Attach metronomes to engine step functions.
  slowMetro.onBang(slowEngineStep);
  fastMetro.onBang(fastEngineStep);
}

void step() {}

void slowEngineStep() {
  slowEngine.step(); // Update slow engine.
  // Adjust frequency and send to LED.
  squareWave.frequency(ledFrequency);
  squareWave >> led;
}

void fastEngineStep() {
  fastEngine.step(); // Update fast engine.
  // On button press, increase square wave frequency.
  if (button.rose())
    ledFrequency++;
}
```

Example: ESP32 Dual Core

In this example, we will take full advantage of the ESP32's dual core architecture:

- **Core 1 (default)** will run the **primary engine** to update a simple indicator LED.
- **Core 0** will run a **secondary engine** to animate Neopixel LEDs using a waveform.

```
#include <Plaquette.h>
#include <Adafruit_NeoPixel.h>

// Pin where the Neopixel strip is connected
#define STRIP_PIN  5
#define NUM_LEDS  16

// The NeoPixel LED strip.
Adafruit_NeoPixel strip(NUM_LEDS, STRIP_PIN, NEO_GRB + NEO_KHZ800); // Neopixels.

SquareWave indicatorLedBlink(1.0, 0.2); // Blinking oscillator.
DigitalOut indicatorLed(LED_BUILTIN);   // Indicator LED.

Engine ledEngine; // Secondary engine for controlling NeoPixels.
SineWave ledWave(1.0, ledEngine);  // Waveform for NeoPixels.
```

(continues on next page)

(continued from previous page)

```
void begin() {
  xTaskCreatePinnedToCore( // Launch LED engine on Core 0.
    [] (void* param) {
      ledEngine.begin(); // Start LED engine.

      // Initialize LED strip.
      strip.begin();
      strip.show();

      // Main loop.
      while (true) {
        ledEngine.step(); // Step engine.
        stepLeds();       // Update LEDs.
        vTaskDelay(1);    // Important! Prevents IDLE on Core 0 (1 tick = ~1ms).
      }
    },
    "LED Engine", 2048, nullptr, 1, nullptr,
    0 // Core 0
  );
}

// Primary engine step function.
void step() {
  indicatorLedBlink >> indicatorLed; // Blink.
}

// Step function for the LED engine.
void stepLeds() {
  // Update LED strip according to LED wave.
  for (int i = 0; i < NUM_LEDS; i++) {
    float phaseShift = mapTo01(i, 0, NUM_LEDS-1);       // LED position to % phase shift.
    float shiftedLedWave = ledWave.shiftBy(phaseShift); // Shifted wave value in [0, 1].
    int level = int(mapFrom01(shiftedLedWave, 0, 255)); // Brightness level in [0, 255].
    strip.setPixelColor(i, strip.Color(level, 0, 0));   // Red channel only.
  }
  strip.show(); // Display LEDs.
}
```

Multiple engines give you more control and better performance, especially on multi-core platforms or in time-sensitive applications like LED control, audio, or robotics.

CHAPTER

THREE

REFERENCE

3.1 Base Units

Basic input-output units.

3.1.1 AnalogIn

An analog (ie. continuous) input unit that returns values between 0 and 1 (ie. 0% and 100%).

The unit is assigned to a specific `pin` on the board.

The `mode` specifies the behavior of the component attached to the pin:

- in `DIRECT` mode (default) the value is expressed as a percentage of the reference voltage (Vref, typically 5V)
- in `INVERTED` mode the value is inverted (ie. 0V corresponds to 100% while 2.5V corresponds to 50%).

> **Warning:** If the analog input pin is **not connected** to anything, the value returned by `get()` will fluctuate based on a number of factors (e.g. the values of the other analog inputs, how close your hand is to the board, etc.).

Example

Control an LED using a potentiometer.

```
#include <Plaquette.h>

AnalogIn potentiometer(A0);

AnalogOut led(9);

SineWave oscillator;

void begin() {}

void step() {
  // The analog input controls the frequency of the LED's oscillation.
  oscillator.frequency(potentiometer.mapTo(2.0, 10.0));
  oscillator >> led;
}
```

Reference

class **AnalogIn** : public Unit, public PinConfig, public Smoothable

A generic class representing a simple analog input.

Public Functions

AnalogIn(uint8_t pin, *Engine* &engine = *Engine*::*primary*())

Constructor.

Parameters

- **pin** – the pin number
- **mode** – the mode (DIRECT or INVERTED)

inline virtual float **get**()

Returns value in [0, 1].

virtual float **mapTo**(float toLow, float toHigh)

Maps value to new range.

float **read**() const

Direcly reads value from the pin (bypasses mode, smoothing, and engine).

int **rawRead**() const

Directly reads raw value from the pin (bypasses mode, smoothing, and engine).

inline uint8_t **pin**() const

Returns the pin this component is attached to.

inline uint8_t **mode**() const

Returns the mode of the component.

inline virtual void **mode**(uint8_t mode)

Changes the mode of the component.

inline virtual void **smooth**(float smoothTime = PLAQUETTE_DEFAULT_SMOOTH_WINDOW)

Apply smoothing to object.

inline virtual void **noSmooth**()

Remove smoothing.

inline virtual void **cutoff**(float hz)

Changes the smoothing window cutoff frequency (expressed in Hz).

inline float **cutoff**() const

Returns the smoothing window cutoff frequency (expressed in Hz).

See Also

- *AnalogOut*
- *DigitalIn*

3.1.2 AnalogOut

An analog (ie. continuous) output unit that converts a value between 0 and 1 (ie. 0% and 100%) into an analog voltage on one of the analog output pins.

The unit is assigned to a specific `pin` on the board.

The `mode` specifies the behavior of the component attached to the pin:

- in `DIRECT` mode (default) the pin acts as the source of current and the value is expressed as a percentage of the maximum voltage (Vcc, typically 5V)
- in `INVERTED` mode the source of current is external (Vcc)

Example

```
AnalogOut led(9);

void begin() {
  led.put(0.5);
}

void step() {
  // The LED value is changed randomly by a tiny amount (random walk).
  // Mutliplying by samplePeriod() makes sure the rate of change stays stable.
  (led + randomFloat(-0.1, 0.1) * samplePeriod()) >> led;
}
```

Important: On most Arduino boards analog outputs rely on Pulse Width Modulation (PWM). After a call to `put(value)`, the pin will generate a steady square wave of the specified duty cycle until the next call to `put()` on the same pin. The frequency of the PWM signal on most pins is approximately 490 Hz. On the Uno and similar boards, pins 5 and 6 have a frequency of approximately 980 Hz.

Note: On most Arduino boards (those with the ATmega168 or ATmega328P), this functionality works on pins 3, 5, 6, 9, 10, and 11. On the Arduino Mega, it works on pins 2 - 13 and 44 - 46. Older Arduino boards with an ATmega8 only support `AnalogOut` on pins 9, 10, and 11. The Arduino DUE supports analog output on pins 2 through 13, plus pins DAC0 and DAC1. Unlike the PWM pins, DAC0 and DAC1 are Digital to Analog converters, and act as true analog outputs.

Reference

class **AnalogOut** : public AnalogSource, public PinConfig

A generic class representing a simple PWM output.

Public Functions

AnalogOut(uint8_t pin, *Engine* &engine = *Engine*::*primary*())

Constructor with default mode DIRECT.

Parameters

pin – the pin number

AnalogOut(uint8_t pin, uint8_t mode, *Engine* &engine = *Engine*::*primary*())

Constructor.

Parameters

- **pin** – the pin number
- **mode** – the mode (DIRECT or INVERTED)

virtual float **put**(float value)

Pushes value into the component and returns its (possibly filtered) value.

inline virtual void **invert**()

Inverts value by calling `put(1-get())` (eg. 0.2 becomes 0.8).

void **write**(float value)

Direclty writes value in [0, 1] to the pin (bypasses mode and engine).

void **rawWrite**(int value)

Direclty writes raw value to the pin (bypasses mode and engine).

inline virtual float **get**()

Returns value in [0, 1].

inline uint8_t **pin**() const

Returns the pin this component is attached to.

inline uint8_t **mode**() const

Returns the mode of the component.

inline virtual void **mode**(uint8_t mode)

Changes the mode of the component.

See Also

- *AnalogIn*
- *DigitalOut*

3.1.3 DigitalIn

A digital (ie. binary) input unit that can be either "on" or "off".

The unit is assigned to a specific `pin` on the board.

The `mode` specifies the behavior of the component attached to the pin:

- in `DIRECT` mode (default) the unit will be "on" when the voltage on the pin is high (Vref, typically 5V)
- in `INVERTED` mode the unit will be "on" when the voltage on the pin is low (GND)
- in `INTERNAL_PULLUP` mode the internal pullup resistor is used, simplifying usage of switches and buttons

Debouncing

Some digital inputs such as push-buttons often generate spurious open/close transitions when pressed, due to mechanical and physical issues: these transitions called "bouncing" may be read as multiple presses in a very short time, fooling the program.

The DigitalIn object features debouncing capabilities which can prevent this kind of problems. Debouncing can be achieved using different modes: stable (default) (`DEBOUNCE_STABLE`), lock-out (`DEBOUNCE_LOCK_OUT`) and prompt-detect (`DEBOUNCE_PROMPT_DETECT`). For more information please refer to the documentation of the Bounce2 Arduino Library.

Example

Turns on and off a light emitting diode (LED) connected to digital pin 13, when pressing a pushbutton attached to digital pin 2. Pushbutton should be wired by connecting one side to pin 2 and the other to ground.

```
#include <Plaquette.h>

DigitalIn button(2, INTERNAL_PULLUP);

DigitalOut led(13);

void begin() {
  button.debounce(); // debounce button
}

void step() {
  // Toggle the LED each time the button is pressed.
  if (button.rose())
    led.toggle();
}
```

Reference

class **DigitalIn** : public DigitalSource, public PinConfig, public Debounceable

A generic class representing a simple digital input.

Public Functions

DigitalIn(uint8_t pin, *Engine* &engine = *Engine*::*primary*())

Constructor.

Parameters

- **pin** – the pin number
- **mode** – the mode (DIRECT, INVERTED, or INTERNAL_PULLUP)

virtual void **mode**(uint8_t mode)

Changes the mode of the component.

float **read**() const

Directly reads value from the pin as 1 or 0 (bypasses mode, debounce, and engine).

int **rawRead**() const

Directly reads raw value from the pin as HIGH or LOW (bypasses mode, debounce, and engine).

inline virtual bool **isOn**()

Returns true iff the input is "on".

inline virtual bool **rose**()

Returns true if the value rose.

inline virtual bool **fell**()

Returns true if the value fell.

inline virtual bool **changed**()

Returns true if the value changed.

inline virtual int8_t **changeState**()

Difference between current and previous value of the unit.

inline virtual void **onRise**(EventCallback callback)

Registers event callback on rise event.

inline virtual void **onFall**(EventCallback callback)

Registers event callback on fall event.

inline virtual void **onChange**(EventCallback callback)

Registers event callback on change event.

inline virtual bool **isOff**()

Returns true iff the input is "off".

inline virtual int **getInt**()

Returns value as integer (0 or 1).

inline virtual float **get**()

Returns value as float (either 0.0 or 1.0).

inline uint8_t **pin**() const
: Returns the pin this component is attached to.

inline uint8_t **mode**() const
: Returns the mode of the component.

inline virtual void **debounce**(float debounceTime = PLAQUETTE_DEFAULT_DEBOUNCE_WINDOW)
: Apply smoothing to object.

inline virtual void **noDebounce**()
: Remove smoothing.

inline uint8_t **debounceMode**() const
: Returns the debounce mode.

inline void **debounceMode**(uint8_t mode)
: Sets debounce mode.

Parameters
mode – the debounce mode (DEBOUNCE_DEFAULT, DEBOUNCE_LOCK_OUT or DEBOUNCE_PROMPT_DETECT)

See Also

- *AnalogIn*
- *DigitalOut*
- Bounce2 Arduino Library

3.1.4 DigitalOut

A digital (ie. binary) output unit that can be switched "on" or "off".

The unit is assigned to a specific `pin` on the board.

The `mode` specifies the behavior of the component attached to the pin:

- in `DIRECT` mode (default) the pin acts as the source of current and the component is "on" when the pin is "high" (Vcc, typically 5V)
- in `INVERTED` mode the source of current is external (Vcc) and the component is "on" when the pin is "low" (GND)

Example

Switches off an LED connected in "sink" mode after a timeout.

```
#include <Plaquette.h>

DigitalOut led(13, INVERTED);

void begin() {
  led.on();
}
```

(continues on next page)

(continued from previous page)

```
void step() {
  // Switch the LED off after 5 seconds.
  if (seconds() > 5)
    led.off();
}
```

Reference

class **DigitalOut** : public DigitalSource, public PinConfig

A generic class representing a simple digital output.

Public Functions

DigitalOut(uint8_t pin, *Engine* &engine = *Engine*::*primary*())

Constructor with default mode DIRECT.

Parameters
pin – the pin number

DigitalOut(uint8_t pin, uint8_t mode, *Engine* &engine = *Engine*::*primary*())

Constructor.

Parameters

- **pin** – the pin number
- **mode** – the mode (DIRECT or INVERTED)

virtual void **mode**(uint8_t mode)

Changes the mode of the component.

void **write**(bool value)

Directly writes value to the pin (bypasses mode and engine).

void **write**(float value)

Directly writes value in [0, 1] to the pin (bypasses mode and engine).

void **rawWrite**(int value)

Directly writes HIGH or LOW value to the pin (bypasses mode and engine).

inline virtual bool **isOn**()

Returns true iff the input is “on”.

inline virtual bool **rose**()

Returns true if the value rose.

inline virtual bool **fell**()

Returns true if the value fell.

inline virtual bool **changed**()

Returns true if the value changed.

inline virtual bool **toggle**()

Switches between on and off.

inline virtual int8_t **changeState**()

Difference between current and previous value of the unit.

inline virtual void **onRise**(EventCallback callback)

Registers event callback on rise event.

inline virtual void **onFall**(EventCallback callback)

Registers event callback on fall event.

inline virtual void **onChange**(EventCallback callback)

Registers event callback on change event.

inline virtual bool **isOff**()

Returns true iff the input is "off".

inline virtual int **getInt**()

Returns value as integer (0 or 1).

inline virtual float **get**()

Returns value as float (either 0.0 or 1.0).

inline virtual bool **on**()

Sets output to "on" (ie. true, 1).

inline virtual bool **off**()

Sets output to "off" (ie. false, 0).

inline virtual float **put**(float value)

Pushes value into the unit.

Parameters

value – the value sent to the unit

Returns

the new value of the unit

inline uint8_t **pin**() const

Returns the pin this component is attached to.

inline uint8_t **mode**() const

Returns the mode of the component.

See Also

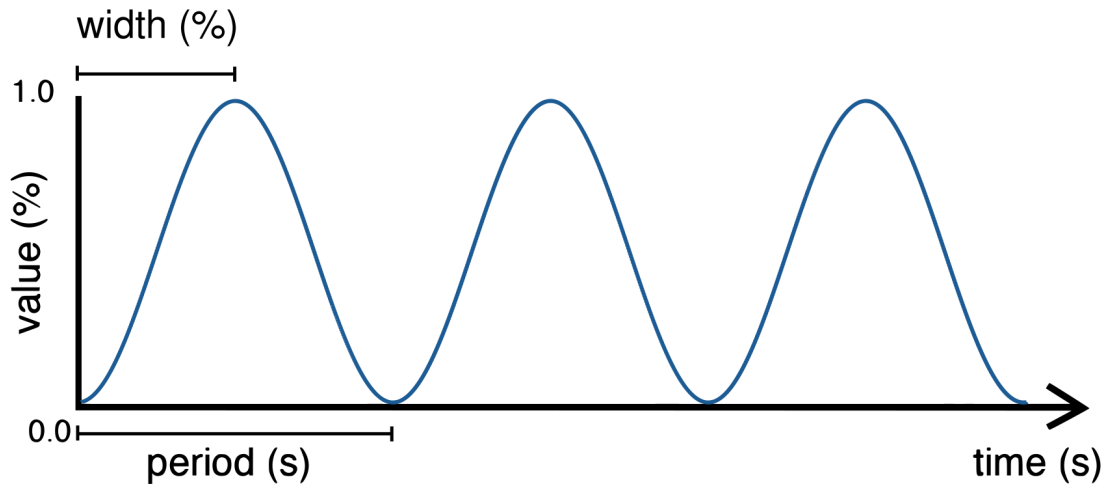
- *AnalogOut*
- *DigitalIn*

3.2 Generators

Source units that generate different kinds of signals.

3.2.1 SineWave

A source unit that can generate a sinusoid or sine wave. The signal is remapped to oscillate between 0 and 1 (rather than -1 and 1 as the traditional sine wave function). It can be tuned by adjusting parameters such as `period`, `frequency`, `amplitude`, or `width`.



Tip: The `width` parameter is included for consistency with triangle and square wave units. It controls when the sine wave reaches its peak within a cycle. A width value of 0.5 (default) yields a standard symmetric sine wave. Lower values shift the peak earlier (left-skewed), while higher values shift it later (right-skewed), allowing for asymmetric sine shapes while preserving smoothness.

Example

Pulses an LED.

```
#include <Plaquette.h>

AnalogOut led(9);

SineWave osc;

void begin() {
  osc.frequency(5.0); // frequency of 5 Hz
```

(continues on next page)

(continued from previous page)

```
}

void step() {
  osc >> led;
}
```

class **SineWave** : public AbstractWave

Sine oscillator. Phase is expressed as % of period.

Public Functions

virtual float **get**()

Returns value in [0, 1].

virtual void **period**(float period)

Sets the period (in seconds).

Parameters
period – the period of oscillation (in seconds)

inline virtual float **period**() const

Returns the period (in seconds).

virtual void **frequency**(float frequency)

Sets the frequency (in Hz).

Parameters
frequency – the frequency of oscillation (in Hz)

inline virtual float **frequency**() const

Returns the frequency (in Hz).

virtual void **bpm**(float bpm)

Sets the frequency in beats-per-minute.

Parameters
bpm – the frequency of oscillation (in BPM)

inline virtual float **bpm**() const

Returns the frequency (in BPM).

virtual void **width**(float width)

Sets the width of the signal as a % of period.

Parameters
width – the width as a value in [0, 1]

inline virtual float **width**() const

Returns the width of the signal.

virtual void **amplitude**(float amplitude)

Sets the amplitude of the wave.

Parameters
amplitude – a value in [0, 1] that determines the amplitude of the wave (centered at 0.5).

inline virtual float **amplitude**() const

Returns the amplitude of the wave.

virtual void **phase**(float phase)

Sets the phase (ie.

the offset, in % of period).

Parameters
phase – the phase (in % of period)

inline virtual float **phase**() const

Returns the phase (in % of period).

virtual float **shiftBy**(float phaseShift)

Returns oscillator's value with given phase shift (in % of period).

Supports values outside [0,1], which will be wrapped accordingly. Eg. shiftBy(0.2) returns future value of oscillator after 20% of its period would have passed.

Parameters
phase – the phase shift (in % of period)

Returns
the value of oscillator with given phase shift

virtual float **atPhase**(float phase)

Returns the oscillator's value at a given absolute phase (in % of period).

Supports values outside [0,1], which will be wrapped accordingly. Eg: atPhase(0.25) returns the oscillator value at 25% of its period.

Parameters
phase – the absolute phase at which to evaluate the oscillator (in % of period)

Returns
the value of the oscillator at the given phase

virtual void **setTime**(float time)

Forces current time (in seconds).

virtual void **addTime**(float time)

Forces current time (in seconds).

inline virtual bool **isRunning**() const

Returns true iff the wave is currently running.

virtual void **forward**()

Sets the direction of oscillation to move forward in time.

virtual void **reverse**()

Sets the direction of oscillation to move backward in time.

virtual void **toggleReverse**()

Toggles the direction of oscillation.

inline virtual float **mapTo**(float toLow, float toHigh)

Maps value to new range.

virtual void **start**()

Starts/restarts the chronometer.

virtual void **stop**()

Interrupts the chronometer and resets to zero.

virtual void **pause**()

Interrupts the chronometer.

virtual void **resume**()

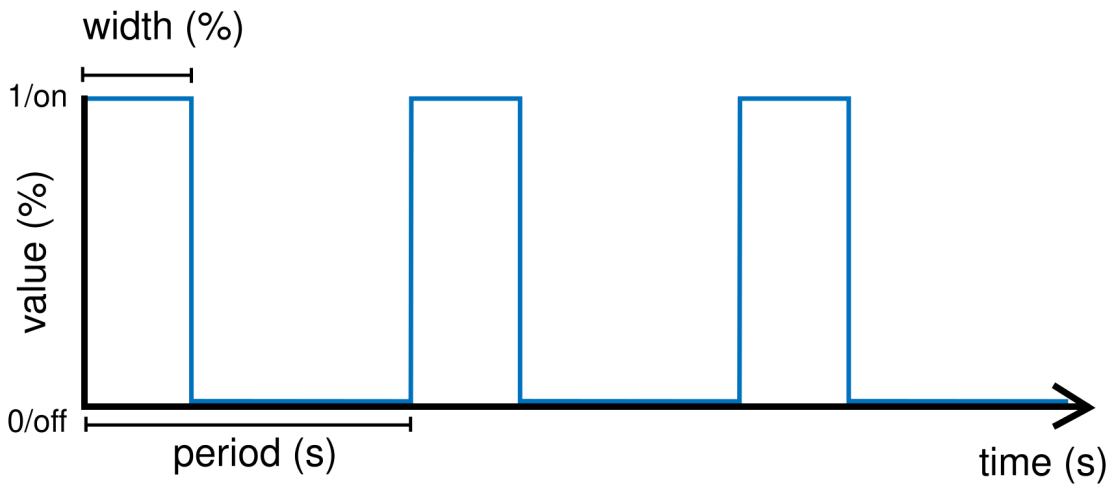
Resumes process.

See Also

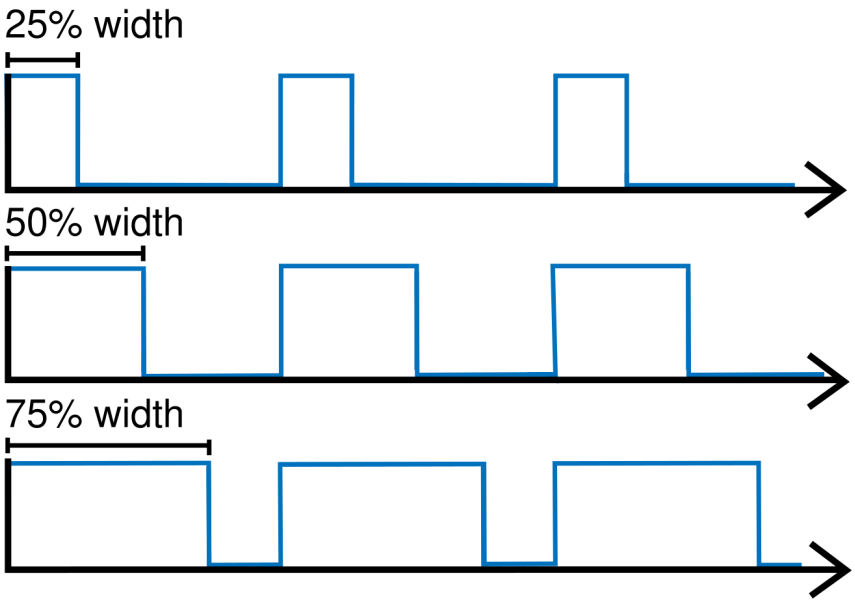
- *SquareWave*
- *TriangleWave*

3.2.2 SquareWave

A source unit that generates a square wave signal. The signal can be tuned by adjusting parameters such as `period`, `frequency`, `amplitude`, or `width`.



The `width` represents the proportion of time (expressed as a percentage) in each cycle (period) during which the wave is "on" – in other words, its *duty cycle*.



Example

Makes the built-in LED blink with a period of 4 seconds. Because the duty cycle is set to 25%, the LED will stay on for 1 second and then off for 3 seconds.

```
#include <Plaquette.h>

DigitalOut led(13);

SquareWave blinkOsc(4.0);

void begin() {
  blinkOsc.width(0.25); // Sets the duty cycle to 25%
}

void step() {
  blinkOsc >> led;
}
```

class **SquareWave** : public AbstractWave

Square oscillator. Duty cycle is expressed as % of period.

Public Functions

SquareWave(*Engine* &engine = *Engine*::*primary*())

Constructor.

Parameters
engine – the engine running this unit

SquareWave(float period, *Engine* &engine = *Engine*::*primary*())

Constructor.

Parameters

- **period** – the period of oscillation (in seconds)
- **engine** – the engine running this unit

SquareWave(float period, float width, *Engine* &engine = *Engine*::*primary*())

Constructor.

Parameters

- **period** – the period of oscillation (in seconds)
- **width** – the duty-cycle as a value in [0, 1]
- **engine** – the engine running this unit

virtual bool **shiftByIsOn**(float phaseShift)

Returns oscillator's on/off with given phase shift (in % of period).

Supports values outside [0,1], which will be wrapped accordingly.

Parameters
phase – the phase shift (in % of period)

Returns
the boolean value of oscillator with given phase shift

virtual bool **atPhaseIsOn**(float phase)

Returns the oscillator's on/off at a given absolute phase (in % of period).

Supports values outside [0,1], which will be wrapped accordingly.

Parameters
phase – the absolute phase at which to evaluate the oscillator (in % of period)

Returns
the value of the oscillator at the given phase

virtual float **get**()

Returns value in [0, 1].

virtual void **period**(float period)

Sets the period (in seconds).

Parameters
period – the period of oscillation (in seconds)

inline virtual float **period**() const

Returns the period (in seconds).

virtual void **frequency**(float frequency)

Sets the frequency (in Hz).

Parameters
frequency – the frequency of oscillation (in Hz)

inline virtual float **frequency**() const

Returns the frequency (in Hz).

virtual void **bpm**(float bpm)

Sets the frequency in beats-per-minute.

Parameters
bpm – the frequency of oscillation (in BPM)

inline virtual float **bpm**() const

Returns the frequency (in BPM).

virtual void **width**(float width)

Sets the width of the signal as a % of period.

Parameters
width – the width as a value in [0, 1]

inline virtual float **width**() const

Returns the width of the signal.

virtual void **amplitude**(float amplitude)

Sets the amplitude of the wave.

Parameters
amplitude – a value in [0, 1] that determines the amplitude of the wave (centered at 0.5).

inline virtual float **amplitude**() const

Returns the amplitude of the wave.

virtual void **phase**(float phase)

Sets the phase (ie.

the offset, in % of period).

Parameters
phase – the phase (in % of period)

inline virtual float **phase**() const

Returns the phase (in % of period).

virtual float **shiftBy**(float phaseShift)

Returns oscillator's value with given phase shift (in % of period).

Supports values outside [0,1], which will be wrapped accordingly. Eg. shiftBy(0.2) returns future value of oscillator after 20% of its period would have passed.

Parameters
phase – the phase shift (in % of period)

Returns
the value of oscillator with given phase shift

virtual float **atPhase**(float phase)

Returns the oscillator's value at a given absolute phase (in % of period).

Supports values outside [0,1], which will be wrapped accordingly. Eg: atPhase(0.25) returns the oscillator value at 25% of its period.

Parameters
phase – the absolute phase at which to evaluate the oscillator (in % of period)

Returns
the value of the oscillator at the given phase

virtual void **setTime**(float time)

Forces current time (in seconds).

virtual void **addTime**(float time)

Forces current time (in seconds).

inline virtual bool **isRunning**() const

Returns true iff the wave is currently running.

virtual void **forward**()

Sets the direction of oscillation to move forward in time.

virtual void **reverse**()

Sets the direction of oscillation to move backward in time.

virtual void **toggleReverse**()

Toggles the direction of oscillation.

inline virtual float **mapTo**(float toLow, float toHigh)

Maps value to new range.

virtual void **start**()

Starts/restarts the chronometer.

virtual void **stop**()

Interrupts the chronometer and resets to zero.

virtual void **pause**()

Interrupts the chronometer.

virtual void **resume**()

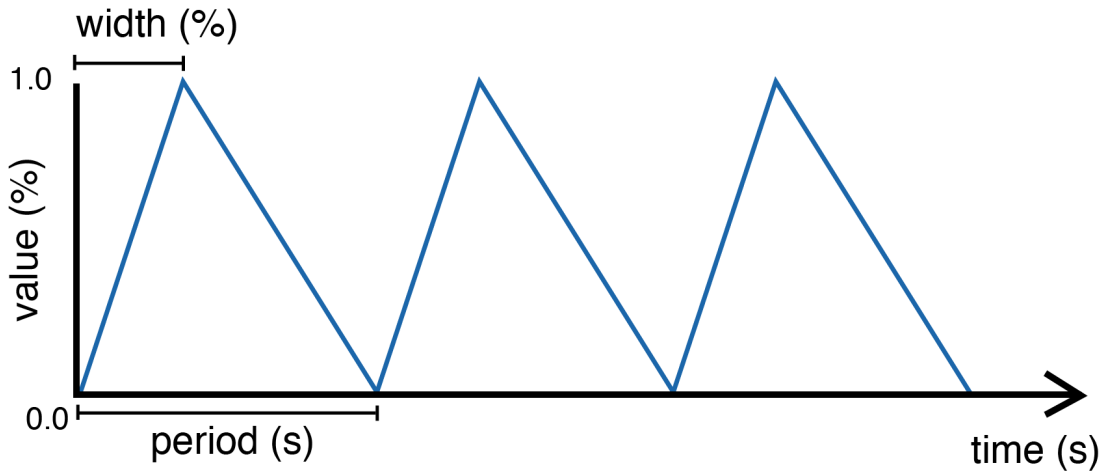
Resumes process.

See Also

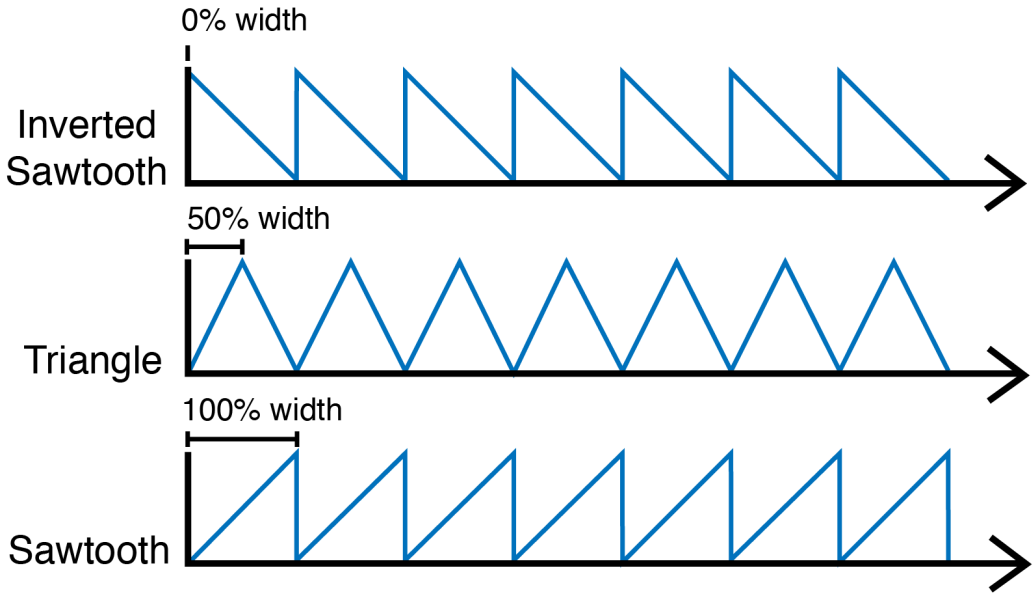
- *SineWave*
- *TriangleWave*

3.2.3 TriangleWave

A source unit that can generate a range of triangle-shaped signals such as the triangle wave and the sawtooth wave. The signal can be tuned by adjusting parameters such as `period`, `frequency`, `amplitude`, or `width`.



The `width` parameter represents the "turning point" during the period at which the signals reaches its maximum and starts going down again. Changing the width allows to generate different kinds of triangular-shaped waves. For example, by setting width to 1.0 (100%) one obtains a *sawtooth* wave; by setting it to 0.0 (0%) an *inverted sawtooth* is created; anything in between generates different flavors of *triangle* waves.



Example

Controls a set of traffic lights that go: red, yellow, green, red, yellow, green, and so on. It uses a sawtooth to iterate through these three states.

```
#include <Plaquette.h>

DigitalOut green(10);
DigitalOut yellow(11);
DigitalOut red(12);

TriangleWave osc(10.0);

void begin() {
  osc.width(1.0); // sawtooth wave
}

void step() {
  // Shut down all lights.
  0 >> led >> yellow >> green;
  // Switch appropriate LED.
  if (osc < 0.4)
    green.on();
  else if (osc < 0.6)
    yellow.on();
  else
    red.on();
}
```

class **TriangleWave** : public AbstractWave

Triangle/sawtooth oscillator.

Public Functions

virtual float **get**()

Returns value in [0, 1].

virtual void **period**(float period)

Sets the period (in seconds).

Parameters
period – the period of oscillation (in seconds)

inline virtual float **period**() const

Returns the period (in seconds).

virtual void **frequency**(float frequency)

Sets the frequency (in Hz).

Parameters
frequency – the frequency of oscillation (in Hz)

inline virtual float **frequency**() const

Returns the frequency (in Hz).

virtual void **bpm**(float bpm)

Sets the frequency in beats-per-minute.

Parameters
bpm – the frequency of oscillation (in BPM)

inline virtual float **bpm**() const

Returns the frequency (in BPM).

virtual void **width**(float width)

Sets the width of the signal as a % of period.

Parameters
width – the width as a value in [0, 1]

inline virtual float **width**() const

Returns the width of the signal.

virtual void **amplitude**(float amplitude)

Sets the amplitude of the wave.

Parameters
amplitude – a value in [0, 1] that determines the amplitude of the wave (centered at 0.5).

inline virtual float **amplitude**() const

Returns the amplitude of the wave.

virtual void **phase**(float phase)

Sets the phase (ie.

the offset, in % of period).

Parameters
phase – the phase (in % of period)

inline virtual float **phase**() const

Returns the phase (in % of period).

virtual float **shiftBy**(float phaseShift)

Returns oscillator's value with given phase shift (in % of period).

Supports values outside [0,1], which will be wrapped accordingly. Eg. shiftBy(0.2) returns future value of oscillator after 20% of its period would have passed.

Parameters
phase – the phase shift (in % of period)

Returns
the value of oscillator with given phase shift

virtual float **atPhase**(float phase)

Returns the oscillator's value at a given absolute phase (in % of period).

Supports values outside [0,1], which will be wrapped accordingly. Eg: atPhase(0.25) returns the oscillator value at 25% of its period.

Parameters
phase – the absolute phase at which to evaluate the oscillator (in % of period)

Returns
the value of the oscillator at the given phase

virtual void **setTime**(float time)

Forces current time (in seconds).

virtual void **addTime**(float time)

Forces current time (in seconds).

inline virtual bool **isRunning**() const

Returns true iff the wave is currently running.

virtual void **forward**()

Sets the direction of oscillation to move forward in time.

virtual void **reverse**()

Sets the direction of oscillation to move backward in time.

virtual void **toggleReverse**()

Toggles the direction of oscillation.

inline virtual float **mapTo**(float toLow, float toHigh)

Maps value to new range.

virtual void **start**()

Starts/restarts the chronometer.

virtual void **stop**()

Interrupts the chronometer and resets to zero.

virtual void **pause**()

Interrupts the chronometer.

virtual void **resume**()

Resumes process.

See Also

- *Ramp*
- *SineWave*
- *SquareWave*

3.3 Timing

Time-management source units.

3.3.1 Alarm

An alarm clock digital source unit. Counts time and becomes "on" when time is up. The alarm can be started, stopped, and resumed.

When started, the alarm stays "off" until it reaches its timeout duration, after which it becomes "on".

Example

Uses an alarm to activate built-in LED. Button is used to reset the alarm at random periods of time.

```
#include <Plaquette.h>

Alarm myAlarm(2.0); // an alarm with 2 seconds duration

DigitalOut led(13);

DigitalIn button(2, INTERNAL_PULLUP);

void begin() {
  myAlarm.start(); // start alarm
}

void step() {
  // Activate LED when alarm rings.
  myAlarm >> led; // the alarm will stay "on" until it is stopped or restarted

  // Reset alarm when button is pushed.
  if (myAlarm && button.rose())
  {
    // Restarts the alarm with a random duration between 1 and 5 seconds.
    myAlarm.duration(randomFloat(1.0, 5.0));
    myAlarm.start();
  }
}
```

Reference

class **Alarm** : public DigitalSource, public AbstractTimer

Chronometer class which becomes "on" after a given duration.

Public Functions

Alarm(*Engine* &engine = *Engine*::*primary*())

Constructor.

Parameters
engine – the engine running this unit

Alarm(float duration, *Engine* &engine = *Engine*::*primary*())

Constructor with duration.

Parameters

- **duration** – duration of the alarm
- **engine** – the engine running this unit

inline virtual bool **finished**()

Returns true iff the alarm just finished its process this step.

inline virtual void **onFinish**(EventCallback callback)

Registers event callback on finish event.

virtual void **setTime**(float time)

Set alarm at specific time.

inline virtual bool **isOn**()

Returns true iff the input is "on".

inline virtual void **onRise**(EventCallback callback)

Registers event callback on rise event.

inline virtual void **onFall**(EventCallback callback)

Registers event callback on fall event.

inline virtual void **onChange**(EventCallback callback)

Registers event callback on change event.

inline virtual bool **isOff**()

Returns true iff the input is "off".

inline virtual int **getInt**()

Returns value as integer (0 or 1).

inline virtual float **get**()

Returns value as float (either 0.0 or 1.0).

virtual void **start**()

Starts/restarts the chronometer.

virtual void **start**(float duration)

Starts/restarts the chronometer with specific duration.

virtual void **duration**(float duration)

Sets the duration of the chronometer.

inline virtual float **duration**() const

Returns duration.

virtual float **progress**() const

The progress of the timer process (in %).

inline virtual bool **isFinished**() const

Returns true iff the chronometer has finished its process.

virtual void **pause**()

Interrupts the chronometer.

virtual void **resume**()

Resumes process.

inline virtual float **elapsed**() const

The time currently elapsed by the chronometer (in seconds).

virtual void **addTime**(float time)

Adds/subtracts time to the chronometer.

inline virtual bool **isRunning**() const

Returns true iff the chronometer is currently running.

virtual void **stop**()

Interrupts the chronometer and resets to zero.

See Also

- *Chronometer*
- *Metronome*
- *Ramp*
- *SquareWave*

3.3.2 Chronometer

An analog unit that counts time in seconds. It can be started, stopped, paused, and resumed.

Example

Uses a chronometer to change the frequency a blinking LED. Restarts after 10 seconds.

```
#include <Plaquette.h>

Chronometer chrono;

DigitalOut led(13);

SquareOsc osc(1.0); // a square oscillator

void begin() {
  chrono.start(); // start chrono
}

void step() {
  // Adjust oscillator's duty cycle according to current timer progress.
  osc.frequency(chrono);

  // Apply oscillator to LED state.
  osc >> led;

  // If the chronometer reaches 10 seconds: restart it.
  if (chrono >= 10.0)
  {
    // Restarts the chronometer.
    chrono.start();
  }
}
```

Reference

class **Chronometer** : public Unit, public AbstractChronometer

Public Functions

Chronometer(*Engine* &engine = *Engine*::*primary*())
Constructor.

inline virtual float **get**()
Returns elapsed time since start (in seconds).

virtual void **pause**()
Interrupts the chronometer.

virtual void **resume**()
Resumes process.

inline virtual float **elapsed**() const
The time currently elapsed by the chronometer (in seconds).

virtual void **setTime**(float time)
Forces current time (in seconds).

virtual void **addTime**(float time)
Adds/subtracts time to the chronometer.

inline virtual bool **isRunning**() const
Returns true iff the chronometer is currently running.

virtual void **start**()
Starts/restarts the chronometer.

virtual void **stop**()
Interrupts the chronometer and resets to zero.

See Also

- *Alarm*
- *Metronome*
- *Ramp*

3.3.3 Metronome

A metronome digital source unit. Emits an "on" signal at a regular pace.

Example

```
#include <Plaquette.h>

Metronome myMetro(0.5); // a metronome with a half-second duration

DigitalOut led(13);

void begin() {
}

void step() {
  if (myMetro)
  {
    // Change LED state.
    led.toggle();
  }
}
```

Reference

class **Metronome** : public DigitalUnit, public Timeable

Chronometer digital unit which emits 1/true/"on" for one frame, at a regular pace.

Public Functions

Metronome(*Engine* &engine = *Engine*::*primary*())

Constructor.

Parameters

engine – the engine running this unit

Metronome(float period, *Engine* &engine = *Engine*::*primary*())

Constructor.

Parameters

- **period** – the period of oscillation (in seconds)
- **engine** – the engine running this unit

inline virtual bool **isOn**()

Returns true iff the metronome fires.

virtual void **period**(float period)

Sets the period (in seconds).

Parameters

period – the period of oscillation (in seconds)

inline virtual float **period**() const

Returns the period (in seconds).

virtual void **frequency**(float frequency)

Sets the frequency (in Hz).

Parameters
frequency – the frequency of oscillation (in Hz)

inline virtual float **frequency**() const

Returns the frequency (in Hz).

virtual void **bpm**(float bpm)

Sets the frequency in beats-per-minute.

Parameters
bpm – the frequency of oscillation (in BPM)

inline virtual float **bpm**() const

Returns the frequency (in BPM).

virtual void **phase**(float phase)

Sets the phase (ie.

the offset, in % of period).

Parameters
phase – the phase (in % of period)

inline virtual float **phase**() const

Returns the phase (in % of period).

virtual void **onBang**(EventCallback callback)

Registers event callback on metronome tick (“bang”) event.

inline virtual bool **isOff**()

Returns true iff the input is “off”.

inline virtual int **getInt**()

Returns value as integer (0 or 1).

inline virtual float **get**()

Returns value as float (either 0.0 or 1.0).

See Also

- *Alarm*
- *Chronometer*
- *Ramp*
- *SquareWave*

3.3.4 Ramp

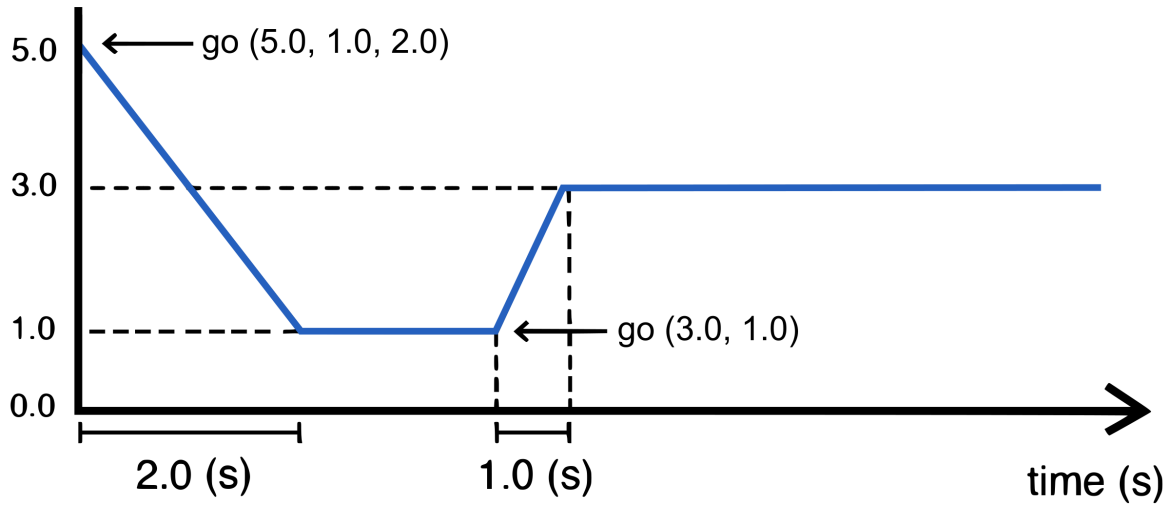
A source unit that generates a smooth transition between two values. The unit can be triggered to start transitioning to a target value for a certain duration.

There are two ways to start the ramp.

By calling `go(from, to, duration)` the ramp will transition from value `from` to value `to` in `duration` seconds.

Alternatively, calling `go(to, duration)` will start a transition from the ramp's current value to `to` in `duration` seconds.

The following diagram shows what happens to the ramp signal if `go(5.0, 1.0, 2.0)` is called, followed later by `go(3.0, 1.0)`:



Important: Ramps also support the use of easing functions in order to create different kinds of expressive effects with signals. An easing function can optionally be specified at the end of a `go()` command or by calling the `easing()` function.

Please refer to *this page* for a full list of available easing functions.

Example

Sequentially ramps through different values.

```
#include <Plaquette.h>

Ramp myRamp(3.0); // initial duration: 3 seconds

StreamOut serialOut(Serial);

```

(continues on next page)

(continued from previous page)

```
void begin() {
  // Apply an easing function (optional).
  myRamp.easing(easeOutSine);
  // Launch ramp: ramp from -10 to 10.
  myRamp.go(-10, 10);
}

void step() {
  if (myRamp.isFinished())
  {
    // Launch ramp from current value to half, increasing duration by one second.
    myRamp.go(myRamp / 2, myRamp.duration() + 1);
  }

  myRamp >> serialOut;
}
```

Reference

class **Ramp** : public Unit, public AbstractTimer

Provides a ramping / tweening mechanism that allows smooth transitions between two values.

Public Functions

Ramp(*Engine* &engine = *Engine*::*primary*())

Constructor.

Parameters
engine – the engine running this unit

Ramp(float duration, *Engine* &engine = *Engine*::*primary*())

Constructor with duration.

Parameters

- **duration** – duration of the ramp
- **engine** – the engine running this unit

virtual float **get**()

Returns value of ramp.

void **easing**(easing_function easing)

Sets easing function to apply to ramp.

Parameters
easing – the easing function

inline void **noEasing**()

Remove easing function (linear/no easing).

virtual void **to**(float to)

Assign final value of the ramp starting from current value.

Parameters
to – the final value

virtual void **from**(float from)

Assign initial value of the ramp.

Parameters
from – the initial value

virtual void **fromTo**(float from, float to)

Assign initial and final values of the ramp.

Parameters

- **from** – the initial value
- **to** – the final value

virtual void **duration**(float duration)

Sets the duration of the chronometer.

inline virtual float **duration**() const

Returns duration.

virtual void **speed**(float speed)

Sets the speed (rate of change) of the ramp in change-per-second.

virtual float **speed**() const

Returns speed (rate of change) of the ramp in change-per-second.

virtual void **start**()

Starts/restarts the ramp. Will repeat the last ramp.

virtual void **go**(float from, float to, float durationOrSpeed, easing_function easing = 0)

Starts a new ramp.

Parameters

- **from** – the initial value
- **to** – the final value
- **durationOrSpeed** – the duration of the ramp (in seconds) or speed (in change-per-second) depending on mode
- **easing** – the easing function (optional).

virtual void **go**(float to, float durationOrSpeed, easing_function easing = 0)

Starts a new ramp, starting from current value.

Parameters

- **to** – the final value
- **durationOrSpeed** – the duration of the ramp (in seconds) or speed (in change-per-second) depending on mode
- **easing** – the easing function (optional)

virtual void **go**(float to, easing_function easing = 0)

Starts a new ramp, starting from current value (keeping the same duration/speed).

Parameters

- **to** – the final value
- **easing** – the easing function (optional)

virtual void **mode**(uint8_t mode)

Changes the mode of the component (RAMP_DURATION or RAMP_SPEED).

inline uint8_t **mode**() const

Returns the mode of the component (RAMP_DURATION or RAMP_SPEED).

inline virtual bool **finished**()

Returns true iff the ramp just finished its process this step.

inline virtual void **onFinish**(EventCallback callback)

Registers event callback on finish event.

virtual void **setTime**(float time)

Forces current time (in seconds).

virtual void **start**(float to, float durationOrSpeed, easing_function easing = 0)

Deprecated:

virtual void **start**(float from, float to, float durationOrSpeed, easing_function easing = 0)

Deprecated:

virtual void **start**(float duration)

Starts/restarts the chronometer with specific duration.

virtual float **progress**() const

The progress of the timer process (in %).

inline virtual bool **isFinished**() const

Returns true iff the chronometer has finished its process.

virtual void **pause**()

Interrupts the chronometer.

virtual void **resume**()

Resumes process.

inline virtual float **elapsed**() const

The time currently elapsed by the chronometer (in seconds).

virtual void **addTime**(float time)

Adds/subtracts time to the chronometer.

inline virtual bool **isRunning**() const

Returns true iff the chronometer is currently running.

virtual void **stop**()

Interrupts the chronometer and resets to zero.

See Also

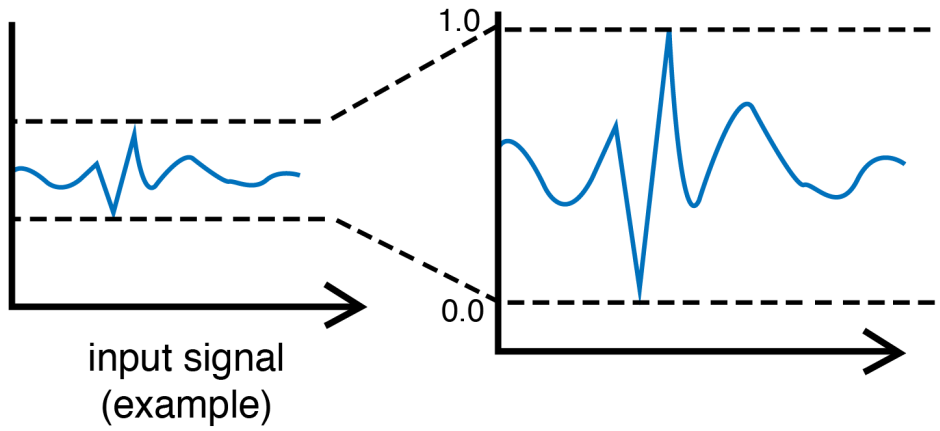
- *Alarm*
- *Chronometer*
- *Easings*
- *Metronome*
- *TriangleWave*

3.4 Filters

Filtering units for real-time signal processing.

3.4.1 MinMaxScaler

This filtering unit regularizes incoming signals by remapping them into a new interval of [0, 1]. It does so by keeping track of the minimum and the maximum values ever taken by the signal and rescales it such that the minimum value of the signal is mapped to 0 and the maximum value is mapped to 1.



In order to accommodate signals that might be changing through time, the user can specify a "decay time window" to control the rate of decay of the minimum and maximum boundaries. The principle is similar to the how the *Smoother* and the *Normalizer* make use of exponential moving average.

Caution: This filtering unit works well as long as there are no "outliers" in the signal (ie. extreme values) that appear in rare conditions. Such values will replace the minimum or maximum value and greatly restrict the spread of the filtered values.

There are three ways to prevent this:

1. Specifying a decay window using the `time(decayTime)` function.
2. Smoothing incoming values using the `smooth()` method or a *Smoother* unit before sending to the `MinMaxScaler`.
3. Using a regularization unit that is less prone to outliers such as the *Normalizer*.

Example

Reacts to high input values by activating an output LED. Scaler is used to automatically adapt to incoming sensor values.

```
#include <Plaquette.h>

AnalogIn sensor(A0);

MinMaxScaler scaler;

DigitalOut led(13);

void begin() {}

void step() {
  // Rescale value.
  sensor >> scaler;

  // Light led on threshold of 80%.
  (scaler > 0.8) >> led;
}
```

Reference

class **MinMaxScaler** : public MovingFilter

Regularizes signal into [0,1] by rescaling it using the min and max values.

Public Functions

MinMaxScaler(*Engine* &engine = *Engine*::*primary*())

Default constructor.

Assigns infinite time window.

Parameters
engine – the engine running this unit

MinMaxScaler(float timeWindow, *Engine* &engine = *Engine*::*primary*())

Constructor with time window.

Parameters

- **timeWindow** – the time window (in seconds)

- **engine** – the engine running this unit

virtual void **infiniteTimeWindow**()

Sets time window to infinite.

virtual void **timeWindow**(float seconds)

Changes the time window (expressed in seconds).

virtual float **timeWindow**() const

Returns the time window (expressed in seconds).

virtual bool **timeWindowIsInfinite**() const

Returns true if time window is infinite.

virtual void **reset**()

Resets the moving filter.

virtual float **put**(float value)

Pushes value into the unit.

If isRunning() is false the filter will not be updated but will just return the filtered value.

Parameters
value – the value sent to the unit

Returns
the new value of the unit

virtual void **resumeCalibrating**()

Switches to calibration mode (default).

Calls to put(value) will return filtered value AND update the normalization statistics.

virtual void **pauseCalibrating**()

Switches to non-calibration mode: calls to put(value) will return filtered value without updating the normalization statistics.

virtual bool **isCalibrating**() const

Returns true iff the moving filter is in calibration mode.

inline virtual float **get**()

Returns value in [0, 1].

inline virtual float **mapTo**(float toLow, float toHigh)

Maps value to new range.

See Also

- *Normalizer*
- *Smoother*

3.4.2 Normalizer

This filtering unit regularizes incoming signals by normalizing them around a target mean and standard deviation. It works by computing the normal distribution of the incoming data (mean and standard variation) and uses this information to re-normalize the data according to a different normal distribution (target mean and variance).

By default, the unit computes the mean and variance over all the data ever received. However, it can instead compute over a time window using an exponential moving average.

Example

Uses a normalizer to analyze input sensor values and detect extreme values.

```
#include <Plaquette.h>

// Analog sensor (eg. photocell or microphone).
AnalogIn sensor(A0);

// Creates a normalizer with mean 0 and standard deviation 1.
Normalizer normalizer(0, 1);

// Output indicator LED.
DigitalOut led(13);

void begin() {}

void step() {
  // Normalize value.
  sensor >> normalizer;

  // Light led if value differs from mean by more
  // than twice the standard deviation.
  (abs(normalizer) > 2.0) >> led;
}
```

Reference

class **Normalizer** : public MovingFilter, public MovingStats

Adaptive normalizer: normalizes values on-the-run using exponential moving averages over mean and standard deviation.

Public Functions

Normalizer(*Engine* &engine = *Engine*::*primary*())

Default constructor.

Assigns infinite time window. Will renormalize data around a mean of 0.5 and a standard deviation of 0.15.

Parameters

engine – the engine running this unit

Normalizer(float timeWindow, *Engine* &engine = *Engine*::*primary*())

Constructor with time window.

Will renormalize data around a mean of 0.5 and a standard deviation of 0.15.

Parameters

- **timeWindow** – the time window over which the normalization applies (in seconds)
- **engine** – the engine running this unit

Normalizer(float mean, float stdDev, *Engine* &engine = *Engine*::*primary*())

Constructor with infinite time window.

Parameters

- **mean** – the target mean
- **stdDev** – the target standard deviation

Normalizer(float mean, float stdDev, float timeWindow, *Engine* &engine = *Engine*::*primary*())

Constructor with time window.

Parameters

- **mean** – the target mean
- **stdDev** – the target standard deviation
- **timeWindow** – the time window over which the normalization applies (in seconds)

inline void **targetMean**(float mean)

Sets target mean of normalized values.

Parameters

mean – the target mean

inline float **targetMean**() const

Returns target mean.

inline void **targetStdDev**(float stdDev)

Sets target standard deviation of normalized values.

Parameters

stdDev – the target standard deviation

inline float **targetStdDev**() const

Returns target standard deviation.

virtual void **infiniteTimeWindow**()

Sets time window to infinite.

virtual void **timeWindow**(float seconds)

Changes the time window (expressed in seconds).

virtual float **timeWindow**() const

Returns the time window (expressed in seconds).

virtual bool **timeWindowIsInfinite**() const

Returns true if time window is infinite.

virtual void **reset**()

Resets the statistics.

virtual float **put**(float value)

Pushes value into the unit.

If isRunning() is false the filter will not be updated but will just return the filtered value.

Parameters
value – the value sent to the unit

Returns
the new value of the unit

virtual float **lowOutlierThreshold**(float nStdDev = 1.5f) const

Returns value above which value is considered to be a low outler (below average).

Parameters
nStdDev – the number of standard deviations (typically between 1 and 3); low values = more sensitive

virtual float **highOutlierThreshold**(float nStdDev = 1.5f) const

Returns value above which value is considered to be a high outler (above average).

Parameters
nStdDev – the number of standard deviations (typically between 1 and 3); low values = more sensitive

bool **isClamped**() const

Return true iff the normalized value is clamped within reasonable range.

void **clamp**(float nStdDev = NORMALIZER_DEFAULT_CLAMP_STDDEV)

Assign clamping value.

Values will then be clamped between reasonable range (*targetMean()* +/- nStdDev * *targetStdDev()*).

Parameters
nStdDev – the number of standard deviations (default: 3.333333333)

void **noClamp**()

Remove clamping.

virtual float **mapTo**(float toLow, float toHigh)

Maps value to new range.

virtual void **resumeCalibrating**()

Switches to calibration mode (default).

Calls to put(value) will return filtered value AND update the normalization statistics.

virtual void **pauseCalibrating**()

Switches to non-calibration mode: calls to put(value) will return filtered value without updating the normalization statistics.

virtual bool **isCalibrating**() const

Returns true iff the moving filter is in calibration mode.

inline virtual float **get**()

Returns value in [0, 1].

virtual bool **isOutlier**(float value, float nStdDev = 1.5f) const

Returns true if the value is considered an outlier.

Parameters

- **value** – the raw value to be tested (non-normalized)
- **nStdDev** – the number of standard deviations (typically between 1 and 3); low values = more sensitive

Returns
true if value is nStdDev number of standard deviations above or below mean

virtual bool **isLowOutlier**(float value, float nStdDev = 1.5f) const

Returns true if the value is considered a low outlier (below average).

Parameters

- **value** – the raw value to be tested (non-normalized)
- **nStdDev** – the number of standard deviations (typically between 1 and 3); low values = more sensitive

Returns
true if value is nStdDev number of standard deviations below mean

virtual bool **isHighOutlier**(float value, float nStdDev = 1.5f) const

Returns true if the value is considered a high outlier (above average).

Parameters

- **value** – the raw value to be tested (non-normalized)
- **nStdDev** – the number of standard deviations (typically between 1 and 3); low values = more sensitive

Returns
true if value is nStdDev number of standard deviations above mean

See Also

- *MinMaxScaler*
- *Smoother*

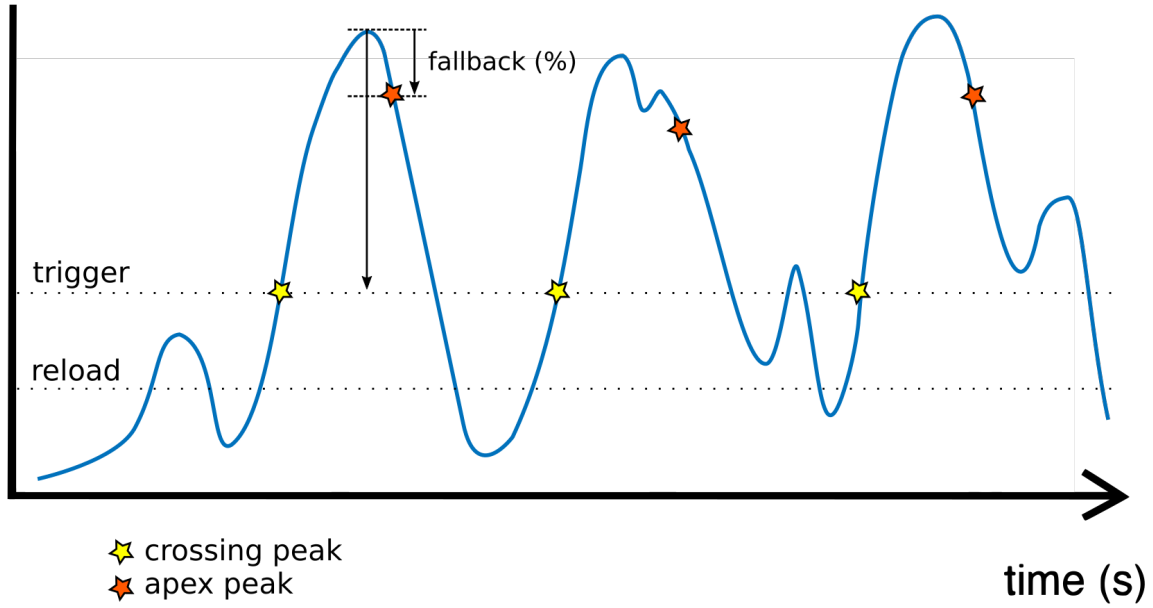
3.4.3 PeakDetector

This unit detects peaks (minima or maxima) in an incoming signal. Peaks are detected based on crossing a trigger threshold above (or below) which a peak is detected.

Two different ways are supported to do this:

- In **crossing** modes (`PEAK_RISING` and `PEAK_FALLING`) the peak is detected *as soon as the signal crosses* the `triggerThreshold`.
- In **apex** modes (`PEAK_MAX` and `PEAK_MIN`) the peak is detected after the signal crosses the `triggerThreshold`, reaches its apex, and then *falls back* by a certain proportion (%) between the threshold and the apex (controlled by the `fallbackTolerance` parameter).

In all cases, after a peak is detected, the detector will wait until the signal crosses back the `reloadThreshold` (which can be adjusted to control detection sensitivity) before it can be triggered again.



In summary, the four different modes available are:

- `PEAK_RISING` : peak detected as soon as `value >= triggerThreshold`, then wait until `value < reloadThreshold`
- `PEAK_FALLING` : peak detected as soon as `value <= triggerThreshold`, then wait until `value > reloadThreshold`
- `PEAK_MAX` : peak detected after `value >= triggerThreshold` and then *falls back* after peaking; then waits until `value < reloadThreshold`
- `PEAK_MIN` : peak detected after `value <= triggerThreshold` and then *falls back* after peaking; then waits until `value > reloadThreshold`

Important: Before sending a signal to a PeakDetector unit, it is recommended to normalize signals, preferably using the *Normalizer* unit. Furthermore, to avoid a noisy signal to generate false peaks, it is recommended to smooth the signal by calling the source unit's `smooth()` method or by using a *Smoother* unit.

Example

Uses a Normalizer and a PeakDetector to analyze input sensor values and detect peaks. Toggle and LED each time a peak is detected.

```
#include <Plaquette.h>

// Analog sensor (eg. photocell or microphone).
AnalogIn sensor(A0);

// Normalization unit to normalize values.
Normalizer normalizer;
```

(continues on next page)

(continued from previous page)

```
// Peak detector. Threshold is set at 1.5 standard deviations above normal.
PeakDetector detector(normalizer.highOutlierThreshold(1.5)); // default mode = PEAK_MAX
// NOTE: You can change mode using optional 2nd parameter, example:
// PeakDetector detector(1.5, PEAK_FALLING));

// Digital LED output.
DigitalOut led;

void begin() {
  // Adjust reload threshold to smaller value than reloadThreshold.
  detector.reloadThreshold(normalizer.highOutlierThreshold(1.0));

  // Adjust fallback tolerance as % between apex and trigger threshold.
  detector.fallbackTolerance(0.2); // 0.2 = 20% (default: 10%)

  // Smooth signal to avoid false peaks due to noise.
  sensor.smooth();

  // Set a time window of 1 minute (60 seconds) on normalizer.
  // This will allow the normalier to slowly readjust itself
  // if the lighting conditions change.
  normalizer.timeWindow(60.0f);
};

void step() {
  // Signal is normalized and sent to peak detector.
  sensor >> normalizer >> detector;

  // Toggle LED when peak detector triggers.
  if (detector)
    led.toggle();
}
```

Reference

class **PeakDetector** : public DigitalUnit

Emits a “bang” signal when another signal peaks.

Public Functions

PeakDetector(float triggerThreshold, *Engine* &engine = *Engine*::*primary*())

Constructor with default mode (PEAK_MAX).

Parameters

- **triggerThreshold** – value that triggers peak detection
- **engine** – the engine running this unit

PeakDetector(float triggerThreshold, uint8_t mode, *Engine* &engine = *Engine*::*primary*())

Constructor.

Possible modes are:

- PEAK_RISING : peak detected when value becomes >= triggerThreshold, then wait until it becomes < reloadThreshold (*)
- PEAK_FALLING : peak detected when value becomes <= triggerThreshold, then wait until it becomes > reloadThreshold (*)
- PEAK_MAX : peak detected after value becomes >= triggerThreshold and then falls back after peaking; then waits until it becomes < reloadThreshold (*)
- PEAK_MIN : peak detected after value becomes <= triggerThreshold and then rises back after peaking; then waits until it becomes > reloadThreshold (*)

Parameters

- **triggerThreshold** – value that triggers peak detection
- **mode** – peak detection mode
- **engine** – the engine running this unit

void **triggerThreshold**(float triggerThreshold)

Sets triggerThreshold.

inline float **triggerThreshold**() const

Returns triggerThreshold.

void **reloadThreshold**(float reloadThreshold)

Sets minimal threshold that "resets" peak detection in crossing (rising/falling) and peak (min/max) modes.

inline float **reloadThreshold**() const

Returns minimal value "drop" for reset.

void **fallbackTolerance**(float fallbackTolerance)

Sets minimal relative "drop" after peak to trigger detection in peak (min/max) modes, expressed as proportion (%) of peak minus triggerThreshold.

inline float **fallbackTolerance**() const

Returns minimal relative "drop" after peak to trigger detection in peak modes.

bool **modeInverted**() const

Returns true if mode is PEAK_FALLING or PEAK_MIN.

bool **modeCrossing**() const

Returns true if mode is PEAK_RISING or PEAK_FALLING.

void **mode**(uint8_t mode)

Sets mode.

inline uint8_t **mode**() const

Returns mode.

virtual float **put**(float value)

Pushes value into the unit.

Parameters

value – the value sent to the unit

Returns
the new value of the unit

inline virtual bool **isOn**()
Returns true iff the triggerThreshold is crossed.

virtual void **onBang**(EventCallback callback)
Registers event callback on peak detection.

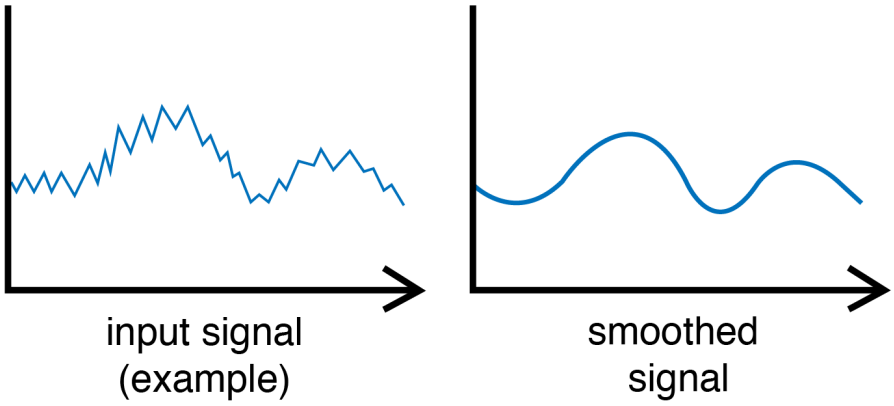
inline virtual float **get**()
Returns value as float (either 0.0 or 1.0).

See Also

- *Normalizer*
- *MinMaxScaler*
- *Smoother*

3.4.4 Smoother

Smooths the incoming signal by removing fast variations and noise (high frequencies).



Example

Smooth a sensor over time.

```
#include <Plaquette.h>

AnalogIn sensor(A0);

// Smooths over time window of 10 seconds.
Smoother smoother(10.0);

StreamOut serialOut(Serial);

void begin() {}

void step() {
  // Smooth value and send it to serial output.
  sensor >> smoother >> serialOut;
}
```

Note: The filter uses an exponential moving average which corresponds to a form of low-pass filter.

Reference

class **Smoother** : public Unit, public MovingAverage

Simple moving average transform filter.

Public Functions

Smoother(*Engine* &engine = *Engine*::*primary*())

Constructor with default smoothing.

Parameters
engine – the engine running this unit

Smoother(float smoothingWindow, *Engine* &engine = *Engine*::*primary*())

Constructor with smoothing window.

Parameters

- **smoothingWindow** – the time window over which the smoothing applies (in seconds)
- **engine** – the engine running this unit

virtual float **put**(float value)

Pushes value into the unit.

Parameters
value – the value sent to the unit

Returns
the new value of the unit

inline virtual float **get**()

Returns smoothed value.

void **timeWindow**(float seconds)

Changes the smoothing window (expressed in seconds).

inline float **timeWindow**() const

Returns the smoothing window (expressed in seconds).

void **cutoff**(float hz)

Changes the smoothing window cutoff frequency (expressed in Hz).

float **cutoff**() const

Returns the smoothing window cutoff frequency (expressed in Hz).

See Also

- *AnalogIn*
- *DigitalIn*

3.5 Functions

Standalone utility functions.

3.5.1 mapFloat()

Re-maps a number from one range to another. That is, a value of `fromLow` would get mapped to `toLow`, a value of `fromHigh` to `toHigh`, and values in-between to values in-between, proportionally.

```
float y = mapFloat(x, 10.0, 50.0, 100.0, 0.0);
```

The function also handles negative numbers well, so that this example

```
float y = mapFloat(x, 10.0, 50.0, 100.0, -100.0);
```

is also valid and works well.

By default, does *not* constrain output to stay within the [`fromHigh`, `toHigh`] range, because out-of-range values are sometimes intended and useful. In order to constrain the return value within range, you can use one of the alternative modes: * the `CONSTRAIN` mode to simply keep the value within range by restricting extreme values as in *constrain() <https://www.arduino.cc/reference/en/language/functions/math/constrain/>* * the `WRAP` mode to wrap the values around as in *wrap()*

```
mapFloat(x, 10.0, 50.0, 100.0, -100.0, CONSTRAIN);
mapFloat(x, 10.0, 50.0, 100.0, -100.0, WRAP);
```

Note: The "lower bounds" (`fromLow` and `toLow`) of either range may be larger or smaller than the "upper bounds" (`fromHigh` and `toHigh`) so the `mapFloat()` function may be used to reverse a range of numbers.

Important: Unlike the Arduino map() function, `mapFloat()` uses floating-point math and *will* generate fractions.

Example

```
#include <Plaquette.h>

SquareWave oscillator(1.0);

DigitalOut led(13);

void begin() {
}

void step() {
  // Change frequency between 2Hz and 15Hz over a 30 seconds period, then the frequency␣
↪will stay at 15Hz.
  float freq = mapFloat(seconds(), 0.0, 30.0, 2.0, 15.0, CONSTRAIN); // try removing␣
↪CONSTRAIN and see what happens
  oscillator.frequency(freq);

  // Send to LED.
  oscillator >> led;
}
```

Reference

float pq::**mapFloat**(double value, double fromLow, double fromHigh, double toLow, double toHigh, uint8_t mode = UNCONSTRAIN)

Re-maps a number from one range to another.

Parameters

- **value** – the number to map
- **fromLow** – the lower bound of the value's current range
- **fromHigh** – the upper bound of the value's current range
- **toLow** – the lower bound of the value's target range
- **toHigh** – the upper bound of the value's target range
- **mode** – set to CONSTRAIN to constrain the return value between toLow and toHigh or WRAP for the value to wrap around

Returns

the mapped value

See Also

- *mapFrom01()*
- *mapTo01()*

3.5.2 mapFrom01()

Re-maps a number in the range [0, 1] to another range. That is, a value of 0 would get mapped to `toLow`, a value of 1 to `toHigh`, values in-between to values in-between, etc.

```
mapFrom01(x, toLow, toHigh)
```

is equivalent to:

```
mapFloat(x, 0, 1, toLow, toHigh)
```

By default, does *not* constrain output to stay within the [`fromHigh`, `toHigh`] range, because out-of-range values are sometimes intended and useful. In order to constrain the return value within range, use the `CONSTRAIN` argument as the last parameter:

```
mapFrom01(x, toLow, toHigh, CONSTRAIN)
```

See *mapFloat()* for more details.

Example

```
#include <Plaquette.h>

SineWave modulator(10.0);

SquareWave oscillator(1.0);

DigitalOut led(13);

void begin() {
}

void step() {
  // Change duty-cycle of oscillator in range [0.2, 0.8].
  float width = mapFrom01(modulator, 0.2, 0.8); // alternative: modulator.mapTo(0.2, 0.8)
  oscillator.width(width);

  // Send to LED.
  oscillator >> led;
}
```

Reference

float pq::**mapFrom01**(double value, double toLow, double toHigh, uint8_t mode = UNCONSTRAIN)

Re-maps a number in range [0, 1] to a new range.

Parameters

- **value** – the number to map (in [0,1])
- **toLow** – the lower bound of the value's target range
- **toHigh** – the upper bound of the value's target range
- **mode** – set to CONSTRAIN to constrain the return value between toLow and toHigh or WRAP for the value to wrap around

Returns

the mapped value in [toLow, toHigh]

See Also

- *mapFloat()*
- *mapTo01()*

3.5.3 mapTo01()

Re-maps a number between 0.0 and 1.0. That is, a value of `fromLow` would get mapped to 0.0, a value of `fromHigh` to 1.0, values in-between to values in-between, etc.

```
mapTo01(x, fromLow, fromHigh)
```

is equivalent to:

```
mapFloat(x, fromLow, fromHigh, 0, 1)
```

By default, does *not* constrain output to stay within the [`fromHigh`, `toHigh`] range, because out-of-range values are sometimes intended and useful. In order to constrain the return value within range, use the `CONSTRAIN` argument as the last parameter:

```
mapTo01(x, fromLow, fromHigh, CONSTRAIN)
```

See *mapFloat()* for more details.

Example

```
#include <Plaquette.h>

AnalogOut led(9);

void begin() {
}

void step() {
```

(continues on next page)

(continued from previous page)

```
  // Generate a sinusoidal values between -1 and 1.
  float x = sin(seconds());

  // Remap to the range [0, 1] and send to LED.
  mapTo01(x, -1, 1) >> led;
}
```

Reference

float pq::**mapTo01**(double value, double fromLow, double fromHigh, uint8_t mode = UNCONSTRAIN)

Re-maps a number to the [0, 1] range.

Parameters

- **value** – the number to map
- **fromLow** – the lower bound of the value's current range
- **fromHigh** – the upper bound of the value's current range
- **mode** – set to CONSTRAIN to constrain the return value between toLow and toHigh or WRAP for the value to wrap around

Returns

the mapped value in [0, 1]

See Also

- *mapFloat()*
- *mapFrom01()*

3.5.4 randomFloat()

This function returns a random real-valued number.

Example

```
#include <Plaquette.h>

DigitalOut led(13);

void begin() {
}

void step() {
  // 2% probability to toggle the LED
  if (randomFloat() < 0.02)
    led.toggle();
}
```

Reference

float pq::**randomFloat**()
: Generates a uniform random number in the interval [0,1).

float pq::**randomFloat**(float max)
: Generates a uniform random number in the interval [0,max).

float pq::**randomFloat**(float min, float max)
: Generates a uniform random number in the interval [min,max) (b>a).

See Also

- random()

3.5.5 seconds()

This function returns the number of seconds since the program started.

Example

```
#include <Plaquette.h>

DigitalOut led(13, DIRECT);

void begin() {
  led.off();
}

void step() {
  // Switch the LED on after 10 seconds.
  if (seconds() > 10)
    led.on();
}
```

Reference

float pq::**seconds**(bool referenceTime = true)
: Returns time in seconds.

  Optional parameter allows to ask for reference time (default) which will yield the same value through one iteration of step(), or "real" time which will return the current total running time.

  Parameters
  : **referenceTime** – determines whether the function returns the reference time or the real time

  Returns
  : the time in seconds

See Also

- micros()
- millis()

3.5.6 wrap()

Restricts a value to an interval [low, high) by wrapping it around.

Code	Result
`wrap(1.0, 1.0, 5.0)`	`1.0`
`wrap(3.0, 1.0, 5.0)`	`3.0`
`wrap(4.9999, 1.0, 5.0)`	`4.9999`
`wrap(5.0, 1.0, 5.0)`	`1.0`
`wrap(10.0, 1.0, 5.0)`	`1.0`
`wrap(13.0, 1.0, 5.0)`	`3.0`

Two alternative versions are provided:

Version	Equivalent to
`wrap(x, high)`	`wrap(x, 0.0, high)`
`wrap01(x)`	`wrap(x, 0.0, 1.0)`

Example

Ramp LED up and then back to zero once every 10 second:

```
#include <Plaquette.h>

AnalogOut led(9);

void begin() {
}

void step() {
   wrap(seconds(), 10.0) >> led;
}
```

Reference

float pq::**wrap**(double x, double low, double high)

Restricts value to the interval [low, high) by wrapping it around.

Parameters

- **x** – the value to wrap
- **low** – the lower boundary
- **high** – the higher boundary

Returns
the value wrapped around [low, high) or [high, low) if high < low

float `pq::wrap`(double x, double high)

Restricts value to the interval [0, high) by wrapping it around.

Parameters

- **x** – the value to wrap
- **high** – the higher bound

Returns
the value wrapped around [0, high) or [high, 0) if high is negative

float `pq::wrap01`(double x)

Restricts value to the interval [0, 1) by wrapping it around.

Parameters
x – the value to wrap

Returns
the value wrapped around [0, 1).

See Also

- *mapFloat()*
- *mapTo01()*

3.6 Structure

Core structural functions and operators.

3.6.1 Engine

A control structure that acts like the **conductor of an orchestra**, managing an ensemble of **units**. It handles their initialization, updates, and timing, ensuring that all components remain synchronized.

By default, all units are automatically added to the **primary engine** which can be accessed using the global object `Plaquette`. By using **secondary engines** you can organize and optimize your code, allowing for multi-tasking, grouping units, switching between ensembles of units, and save power by running engines at lower frequency.

Usage

To create and use an engine, simply declare it:

```
Engine myEngine;
```

You will need to call the engine's `begin()` function at initialization, and then its `step()` function at a regular pace.

To assign units to a specific engine, add it as the last parameter at construction:

```
SquareWave wave(1.0, 0.2, myEngine); // Use myEngine instead of the default
```

Each engine provides its own time measurements:

```
float sec = myEngine.seconds();        // Time since engine started, in seconds
uint32_t ms = myEngine.milliSeconds(); // Time in milliseconds
uint64_t us = myEngine.microSeconds(); // Time in microseconds
```

For more in-depth explanations and examples please read *Synchronizing Groups of Units with Secondary Engines*.

Example

This example demonstrates the use of a secondary engine on an Arduino Uno or Nano using timer2 interrupt.

```
#include <Plaquette.h>

Engine timerEngine; // The secondary timer engine.

Metronome serialMetro(1.0); // Metronome (primary engine).

Metronome toggleMetro(0.25, timerEngine); // Metronome (timer engine).
DigitalOut led(LED_BUILTIN, timerEngine); // Built-in LED (timer engine).

// Primary engine begin().
void begin() {
  timerEngine.begin(); // Begin timer engine.
  timerSetup();        // Initialize timer interrupt timer2.
}

// Primary engine step().
void step() {
  if (serialMetro)
    println("step");
}

// Timer2 interrupt: will be called at 1kHz frequency.
ISR(TIMER2_COMPA_vect) {
  timerEngine.step(); // Step engine.

  if (toggleMetro) // Toggle LED on metro bang.
    led.toggle();
}

// Timer2 setup for 1kHz on AVR.
void timerSetup() {
  // Stop Timer2
  TCCR2A = 0;
  TCCR2B = 0;
  TCNT2  = 0;

  // Set compare match register for 1 kHz increments.
  // 16 MHz / (prescaler * 1000) - 1 = OCR2A
  // Try prescaler = 128 => OCR2A = (16e6 / (128 * 1000)) - 1 = ~124
  OCR2A = 124;
```

(continues on next page)

(continued from previous page)

```
  TCCR2A |= (1 << WGM21); // CTC mode (Clear Timer on Compare Match).
  TCCR2B |= (1 << CS22) | (1 << CS20); // Set prescaler to 128.
  TIMSK2 |= (1 << OCIE2A); // Enable Timer2 compare interrupt.

  sei(); // Enable global interrupts.
}
```

Reference

class **Engine**

The main Plaquette static class containing all the units.

Public Functions

inline void **begin**(unsigned long baudrate = PLAQUETTE_SERIAL_BAUD_RATE)

Function to be used within the PlaquetteLib context (needs to be called at top of setup() method).

inline void **step**()

Function to be used within the PlaquetteLib context (needs to be called at top of loop() method).

inline size_t **nUnits**()

Returns the current number of units.

float **seconds**(bool referenceTime = true) const

Returns time in seconds.

Optional parameter allows to ask for reference time (default) which will yield the same value through one iteration of *step()*, or "real" time which will return the current total running time.

Parameters
referenceTime – determines whether the function returns the reference time or the real time

Returns
the time in seconds

uint32_t **milliSeconds**(bool referenceTime = true) const

Returns time in milliseconds.

Optional parameter allows to ask for reference time (default) which will yield the same value through one iteration of *step()*, or "real" time which will return the current total running time.

Parameters
referenceTime – determines whether the function returns the reference time or the real time

Returns
the time in milliseconds

uint64_t **microSeconds**(bool referenceTime = true) const

Returns time in microseconds.

Optional parameter allows to ask for reference time (default) which will yield the same value through one iteration of *step()*, or "real" time which will return the current total running time.

Parameters
referenceTime – determines whether the function returns the reference time or the real time

Returns
the time in microseconds

inline unsigned long **nSteps**() const
Returns number of steps.

void **sampleRate**(float sampleRate)
Sets sample rate to a fixed value, thus disabling auto sampling rate.

Deprecated:

void **samplePeriod**(float samplePeriod)
Sets sample period to a fixed value, thus disabling auto sampling rate.

Deprecated:

inline float **sampleRate**() const
Returns sample rate.

inline float **samplePeriod**() const
Returns sample period.

inline bool **isPrimary**() const
Returns true if this *Engine* is the main.

Public Static Functions

static *Engine* &**primary**()
Returns the main instance of Plaquette.

See Also

- *begin()*
- *step()*
- *seconds()*
- *Synchronizing Groups of Units with Secondary Engines*

3.6.2 begin()

The `begin()` function is called when a sketch starts. Use it to initialize units, start using libraries, etc. The `begin()` function will only run once, after each powerup or reset of the board.

Hint: Function `begin()` is the Plaquette equivalent of Arduino's setup(). However, Plaquette takes care of many of the initialization calls that need to be done in Arduino such as `pinMode()`. Therefore in many cases it will contain only a few calls or even be left empty.

Example

```
#include <Plaquette.h>

SquareWave oscillator;
AnalogIn input(A0);

void begin() {
  oscillator.period(1.0);
  oscillator.width(0.75);
  input.smooth();
}

void step() {
  // ...
}
```

See Also

- *step()*

3.6.3 step()

After creating a `begin()` function, which initializes and sets the initial values, the `step()` function does precisely what its name suggests, and performs one processing step that loops indefinitely as fast as possible, allowing your program to change and respond. Use it to actively control the board.

Important: Function `step()` is the Plaquette equivalent of Arduino's loop(). However, it is highly recommended that this function executes as fast as possible. Hence, one should performing computationally-intensive processing or calling blocking functions such as `delay()`

Example

```
#include <Plaquette.h>

DigitalIn button(2);

DigitalOut led(13);

void begin() {
}

void step() {
  button >> led;
}
```

See Also

- *begin()*

3.6.4 [] (arrays)

An array is a collection of variables or objects that are accessed with an index number. Arrays can be complicated, but using simple arrays is relatively straightforward.

For a general description of arrays, please refer to this page.

Arrays of Plaquette units such as *DigitalIn*. *SineWave*, and *MinMaxScaler* can be easily created using the following syntax:

```
UnitType array[] = { UnitType(...), UnitType(...), ... };
```

For example, the following code will create an array of three (3) digital outputs on pins 10, 11, and 12:

```
DigitalOut leds[] = { DigitalOut(10), DigitalOut(11), DigitalOut(12) };
```

When initializing a unit with a single parameter, one can simply use the value of the parameter at creation time. Hence the previous code could be rewritten as:

```
DigitalOut leds[] = { 10, 11, 12 };
```

When more than a single parameter is used, however, it needs to be called explicitely with the unit name:

```
SquareWave oscillators[] = { 1.0, 2.0, SquareWave(3.0, 0.8) };
```

Warning: Units in array need to be all of the same type. In other words, it is not currently possible to mix different types of objects such as DigitalIn and SquareWave in the same array.

Example

```
#include <Plaquette.h>

AnalogOut leds[] = { 9, 10, 11 };

// Creates three triangle oscillators with a 2 seconds period, with different width.
TriangleWave oscillators[] = { TriangleWave(2.0, 0.0), 2.0, TriangleWave(2.0, 1.0) };

void begin() {}

void step() {
  // Send each oscillator to its corresponding LED.
  for (int i=0; i<3; i++) {
    oscillators[i] >> leds[i];
  }
}
```

3.6.5 . (dot)

Provides access to an object's methods and data. An object is one instance of a class and may contain both methods (object functions) and data (object variables and constants), as specified in the class definition. The dot operator directs the program to the information encapsulated within an object.

Example

Switches LED on every 4 seconds.

```
#include <Plaquette.h>

DigitalOut led(13);

void begin() {
  led.off();
}

void step() {
  if (round(seconds()) % 4 == 0)
    led.on();
 else
    led.off();
}
```

Syntax

```
object.method()
object.variable
```

3.6.6 >> (pipe)

Sends data across units from left to right. This operator is specific to Plaquette and can be used in a chained manner.

The operation uses the `get()` and `put()` methods of units in such a way that:

```
input >> output;
```

is equivalent to:

```
output.put(input.get());
```

Numerical and boolean values can also be used:

```
12 >> output;
0.8 >> output;
true >> output;
```

Example

```
#include <Plaquette.h>

AnalogIn sensor(A0);

MinMaxScaler scaler;

AnalogOut led(9);

void begin() {}

void step() {
  // Rescale value and send the result to LED.
  sensor >> scaler >> led;
}
```

Syntax

```
input >> output
input >> filter >> output
```

3.7 Extra

Extra units and functions.

3.7.1 Easings

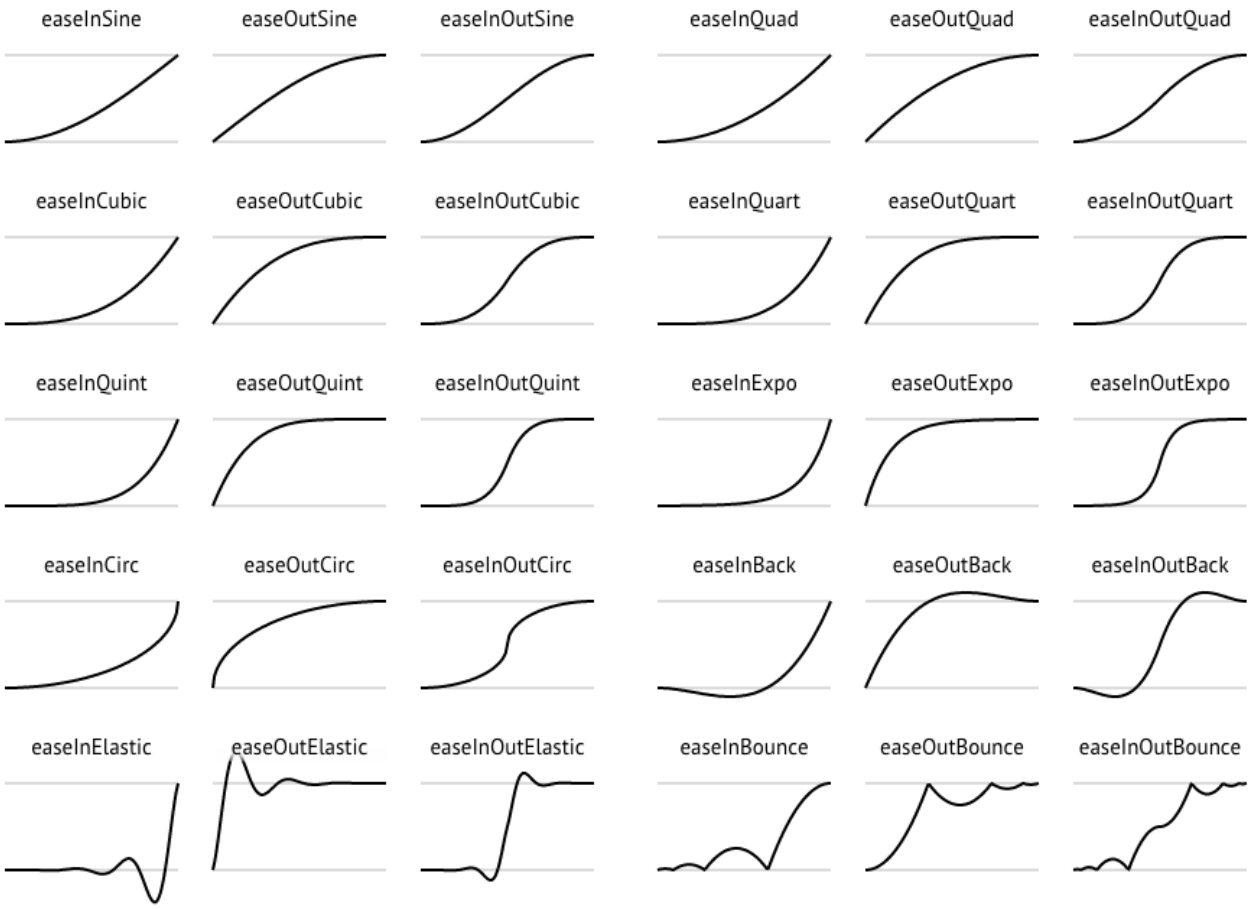
Easing functions apply non-linear effects to changing values, in order to create expressive real-time outputs. Plaquette provides users with a wide range of such functions, typically used with a Ramp unit.

All easing functions have the same signature:

```
float easeFunction(float t)
```

Value `x` should be between 0.0 and 1.0, the returned value is also between 0.0 and 1.0.

This is the list of all easing functions (source: http://easings.net):



See Also

- *Ramp*

3.7.2 ContinuousServoOut

A source unit that controls a continuous rotation servo-motor. A continuous servo-motor can move indefinitely forward or backwards.

Servo motors have three wires: power, ground, and signal. The power wire is typically red, and should be connected to the 5V pin on the Arduino board. The ground wire is typically black or brown and should be connected to a ground pin on the Arduino board. The signal pin is typically yellow, orange or white and should be connected to a digital pin on the Arduino board. Note that servos draw considerable power, so if you need to drive more than one or two, you'll probably need to power them from a separate supply (i.e. not the +5V pin on your Arduino). Be sure to connect the grounds of the Arduino and external power supply together.

Example

Every time a button is pushed, the motor is stopped. Then upon button release it starts moving in the opposite direction.

```
#include <Plaquette.h>
#include <PqServo.h>

// The servo-motor output on pin 9.
ContinuousServoOut servo(9);

// The push-button.
DigitalIn button(2);

// Preserves the servo last speed value.
float lastValue = 0;

void begin() {
  // Debounce button.
  button.debounce();
  // Starts the servo.
  servo.put(1.0);
}

void step() {
  if (button) {
    // Save speed.
    lastValue = servo.get();
    // Stop servo.
    servo.stop();
  }
  else if (button.fell()) {
    // Reset speed.
    servo.put(lastValue);
    // ... then invert it.
    servo.reverse();
  }
}
```

Warning: doxygenclass: Cannot find class “ContinuousServoOut” in doxygen xml output for project “Plaquette” from directory: xml

See Also

- *AnalogOut*
- *ServoOut*

3.7.3 ServoOut

A source unit that controls a standard servo-motor.

Servo motors have three wires: power, ground, and signal. The power wire is typically red, and should be connected to the 5V pin on the Arduino board. The ground wire is typically black or brown and should be connected to a ground pin on the Arduino board. The signal pin is typically yellow, orange or white and should be connected to a digital pin on the Arduino board. Note that servos draw considerable power, so if you need to drive more than one or two, you'll probably need to power them from a separate supply (i.e. not the +5V pin on your Arduino). Be sure to connect the grounds of the Arduino and external power supply together.

Example

Sweeps the shaft of a servo motor back and forth across 180 degrees.

```
#include <Plaquette.h>
#include <PqServo.h>

// The servo-motor output on pin 9.
ServoOut servo(9);

// Oscillator to make the servo sweep.
SineWave oscillator(2.0);

void begin() {
  // Position the servo in center.
  servo.center();
}

void step() {
  // Updates the value and send it back as output.
  oscillator >> servo;
}
```

Warning: doxygenclass: Cannot find class "ServoOut" in doxygen xml output for project "Plaquette" from directory: xml

See Also

- *AnalogOut*
- *ContinuousServoOut*

3.7.4 StreamIn

An input unit that can receive values transmitted through a stream – for example, the Arduino serial line. Values are sent in clear text and separated by newlines and/or carriage returns.

Example

Controls the value of a LED using serial. Try opening the serial monitor and sending values between 0 and 1.

```
#include <Plaquette.h>

StreamIn serialIn(Serial);

AnalogOut led(9);

void begin() {}

void step() {
  serialIn >> led;
}
```

To run this example:

1. Upload the code.
2. In the Arduino software open the serial monitor: **Tools > Serial Monitor**.
3. Make sure the default baudrate of **9600** bps is selected.
4. Make sure one of the options "Newline", "Carriage return", or "Both NL + CR" is selected.
5. Write a number between 0.0 and 1.0 and press "Enter". This should allow you to set the LED intensity.
6. Try different values.

Reference

class **StreamIn** : public AnalogSource

Stream/serial input. Reads float values using Arduino built-in parseFloat().

Public Functions

StreamIn(*Engine* &engine = *Engine*::*primary*())

Default constructor.

Parameters

engine – the engine running this unit

StreamIn(Stream &stream, *Engine* &engine = *Engine*::*primary*())

Constructor.

Parameters

- **stream** – a reference to a Stream object
- **engine** – the engine running this unit

inline virtual float **mapTo**(float toLow, float toHigh)

Maps value to new range.

inline virtual bool **updated**()

Returns true iff value was changed.

inline virtual void **onUpdate**(EventCallback callback)

Registers event callback on finish event.

inline virtual float **get**()

Returns value in [0, 1].

See Also

- *AnalogIn*
- *DigitalIn*
- *StreamOut*
- Arduino serial
- Arduino streams

3.7.5 StreamOut

An output unit that transmits values through a stream – for example, the Arduino serial line. Values are sent in clear text and separated by newlines and/or carriage returns.

Example

Outputs the number of seconds to serial.

```
#include <Plaquette.h>

StreamOut serialOut(Serial);

void begin() {}
```

(continues on next page)

(continued from previous page)

```
void step() {
  // Output the number of seconds
  seconds() >> serialOut;
}
```

To run this example:

1. Upload the code.
2. In the Arduino software open the serial monitor: **Tools > Serial Monitor**.
3. Make sure the default baudrate of **9600** bps is selected.
4. You should see the seconds increase.
5. Close the monitor and open serial plotter: **Tools > Serial Plotter**.
6. You should see a graphical representation of the seconds.
7. Replace the line in `step()` by: `sin(seconds()) >> serialOut` and upload. You should now see a sine wave signal in the serial plotter.

Reference

class **StreamOut** : public AnalogSource

Stream/serial output. Number of digits of precision is configurable.

Public Functions

StreamOut(*Engine* &engine = *Engine*::*primary*())

Default constructor.

Parameters
engine – the engine running this unit

StreamOut(Stream &stream, *Engine* &engine = *Engine*::*primary*())

Constructor.

Parameters

- **stream** – a reference to a Stream object
- **engine** – the engine running this unit

virtual float **put**(float value)

Pushes value into the unit.

Parameters
value – the value sent to the unit

Returns
the new value of the unit

virtual void **precision**(uint8_t digits)

Sets precision of the output.

Parameters
digits – the number of digits to show after decimal point

inline virtual float **get**()
Returns value in [0, 1].

See Also

- *AnalogOut*
- *DigitalOut*
- *StreamIn*
- Arduino serial
- Arduino streams

CHAPTER

FOUR

RELATED INFO

4.1 Credits

4.1.1 Core Developers

- Sofian Audry (main code, API design, documentation) • Website • GitHub
- Thomas Ouellet Fredericks (original concept, API design) • Website • GitHub

4.1.2 Contributors

- API Design & Documentation: Erin Gee • Website • GitHub
- Testing: Luana Belinsky • Website
- Logo: Ian Donnelly • Website
- Code: Matthew Loewen • Website • GitHub
- Code: Samuel Favreau • Website

4.1.3 Partners

- mXlab
- LFO
- EnsadLab
- SAT

4.1.4 Funding

- Canada Council for the Arts
- NSERC
- FRQSC

Plaquette's base source code was produced as part of a research project at labXmodal. A special thanks to Chris Salter for his support.

4.1.5 Inspiration

Plaquette borrows ideas from the Arduino, ChucK, mbed, Processing, and Pure Data.

4.2 License

Plaquette is distributed under the Gnu General Public License v 3.0.

The text of the Plaquette documentation is licensed under a Creative Commons Attribution-ShareAlike 3.0 License. Parts of the text was copied and/or adapted from the Arduino documentation. Code samples in the guide are released into the public domain.

The Plaquette documentation is licensed under a Creative Commons Attribution-Share Alike 3.0 License. Parts of the documentation has been borrowed and/or adapted from the Arduino Reference and from the Processing Reference texts.

INDEX

A

Alarm (*C++ class*), 84
Alarm::addTime (*C++ function*), 85
Alarm::Alarm (*C++ function*), 84
Alarm::duration (*C++ function*), 85
Alarm::elapsed (*C++ function*), 85
Alarm::finished (*C++ function*), 84
Alarm::get (*C++ function*), 85
Alarm::getInt (*C++ function*), 85
Alarm::isFinished (*C++ function*), 85
Alarm::isOff (*C++ function*), 85
Alarm::isOn (*C++ function*), 85
Alarm::isRunning (*C++ function*), 85
Alarm::onChange (*C++ function*), 85
Alarm::onFall (*C++ function*), 85
Alarm::onFinish (*C++ function*), 84
Alarm::onRise (*C++ function*), 85
Alarm::pause (*C++ function*), 85
Alarm::progress (*C++ function*), 85
Alarm::resume (*C++ function*), 85
Alarm::setTime (*C++ function*), 85
Alarm::start (*C++ function*), 85
Alarm::stop (*C++ function*), 86
AnalogIn (*C++ class*), 64
AnalogIn::AnalogIn (*C++ function*), 64
AnalogIn::cutoff (*C++ function*), 64
AnalogIn::get (*C++ function*), 64
AnalogIn::mapTo (*C++ function*), 64
AnalogIn::mode (*C++ function*), 64
AnalogIn::noSmooth (*C++ function*), 64
AnalogIn::pin (*C++ function*), 64
AnalogIn::rawRead (*C++ function*), 64
AnalogIn::read (*C++ function*), 64
AnalogIn::smooth (*C++ function*), 64
AnalogOut (*C++ class*), 66
AnalogOut::AnalogOut (*C++ function*), 66
AnalogOut::get (*C++ function*), 66
AnalogOut::invert (*C++ function*), 66
AnalogOut::mode (*C++ function*), 66
AnalogOut::pin (*C++ function*), 66
AnalogOut::put (*C++ function*), 66
AnalogOut::rawWrite (*C++ function*), 66
AnalogOut::write (*C++ function*), 66

C

Chronometer (*C++ class*), 87
Chronometer::addTime (*C++ function*), 87
Chronometer::Chronometer (*C++ function*), 87
Chronometer::elapsed (*C++ function*), 87
Chronometer::get (*C++ function*), 87
Chronometer::isRunning (*C++ function*), 87
Chronometer::pause (*C++ function*), 87
Chronometer::resume (*C++ function*), 87
Chronometer::setTime (*C++ function*), 87
Chronometer::start (*C++ function*), 87
Chronometer::stop (*C++ function*), 87

D

DigitalIn (*C++ class*), 68
DigitalIn::changed (*C++ function*), 68
DigitalIn::changeState (*C++ function*), 68
DigitalIn::debounce (*C++ function*), 69
DigitalIn::debounceMode (*C++ function*), 69
DigitalIn::DigitalIn (*C++ function*), 68
DigitalIn::fell (*C++ function*), 68
DigitalIn::get (*C++ function*), 68
DigitalIn::getInt (*C++ function*), 68
DigitalIn::isOff (*C++ function*), 68
DigitalIn::isOn (*C++ function*), 68
DigitalIn::mode (*C++ function*), 68, 69
DigitalIn::noDebounce (*C++ function*), 69
DigitalIn::onChange (*C++ function*), 68
DigitalIn::onFall (*C++ function*), 68
DigitalIn::onRise (*C++ function*), 68
DigitalIn::pin (*C++ function*), 68
DigitalIn::rawRead (*C++ function*), 68
DigitalIn::read (*C++ function*), 68
DigitalIn::rose (*C++ function*), 68
DigitalOut (*C++ class*), 70
DigitalOut::changed (*C++ function*), 70
DigitalOut::changeState (*C++ function*), 70
DigitalOut::DigitalOut (*C++ function*), 70
DigitalOut::fell (*C++ function*), 70
DigitalOut::get (*C++ function*), 71

DigitalOut::getInt (C++ function), 71
DigitalOut::isOff (C++ function), 71
DigitalOut::isOn (C++ function), 70
DigitalOut::mode (C++ function), 70, 71
DigitalOut::off (C++ function), 71
DigitalOut::on (C++ function), 71
DigitalOut::onChange (C++ function), 71
DigitalOut::onFall (C++ function), 71
DigitalOut::onRise (C++ function), 71
DigitalOut::pin (C++ function), 71
DigitalOut::put (C++ function), 71
DigitalOut::rawWrite (C++ function), 70
DigitalOut::rose (C++ function), 70
DigitalOut::toggle (C++ function), 70
DigitalOut::write (C++ function), 70

E

Engine (C++ class), 115
Engine::begin (C++ function), 115
Engine::isPrimary (C++ function), 116
Engine::microSeconds (C++ function), 115
Engine::milliSeconds (C++ function), 115
Engine::nSteps (C++ function), 116
Engine::nUnits (C++ function), 115
Engine::primary (C++ function), 116
Engine::samplePeriod (C++ function), 116
Engine::sampleRate (C++ function), 116
Engine::seconds (C++ function), 115
Engine::step (C++ function), 115

M

Metronome (C++ class), 88
Metronome::bpm (C++ function), 89
Metronome::frequency (C++ function), 88, 89
Metronome::get (C++ function), 89
Metronome::getInt (C++ function), 89
Metronome::isOff (C++ function), 89
Metronome::isOn (C++ function), 88
Metronome::Metronome (C++ function), 88
Metronome::onBang (C++ function), 89
Metronome::period (C++ function), 88
Metronome::phase (C++ function), 89
MinMaxScaler (C++ class), 95
MinMaxScaler::get (C++ function), 96
MinMaxScaler::infiniteTimeWindow (C++ function), 96
MinMaxScaler::isCalibrating (C++ function), 96
MinMaxScaler::mapTo (C++ function), 96
MinMaxScaler::MinMaxScaler (C++ function), 95
MinMaxScaler::pauseCalibrating (C++ function), 96
MinMaxScaler::put (C++ function), 96
MinMaxScaler::reset (C++ function), 96
MinMaxScaler::resumeCalibrating (C++ function), 96
MinMaxScaler::timeWindow (C++ function), 96
MinMaxScaler::timeWindowIsInfinite (C++ function), 96

N

Normalizer (C++ class), 97
Normalizer::clamp (C++ function), 99
Normalizer::get (C++ function), 99
Normalizer::highOutlierThreshold (C++ function), 99
Normalizer::infiniteTimeWindow (C++ function), 98
Normalizer::isCalibrating (C++ function), 99
Normalizer::isClamped (C++ function), 99
Normalizer::isHighOutlier (C++ function), 100
Normalizer::isLowOutlier (C++ function), 100
Normalizer::isOutlier (C++ function), 99
Normalizer::lowOutlierThreshold (C++ function), 99
Normalizer::mapTo (C++ function), 99
Normalizer::noClamp (C++ function), 99
Normalizer::Normalizer (C++ function), 97, 98
Normalizer::pauseCalibrating (C++ function), 99
Normalizer::put (C++ function), 98
Normalizer::reset (C++ function), 98
Normalizer::resumeCalibrating (C++ function), 99
Normalizer::targetMean (C++ function), 98
Normalizer::targetStdDev (C++ function), 98
Normalizer::timeWindow (C++ function), 98
Normalizer::timeWindowIsInfinite (C++ function), 98

P

PeakDetector (C++ class), 102
PeakDetector::fallbackTolerance (C++ function), 103
PeakDetector::get (C++ function), 104
PeakDetector::isOn (C++ function), 104
PeakDetector::mode (C++ function), 103
PeakDetector::modeCrossing (C++ function), 103
PeakDetector::modeInverted (C++ function), 103
PeakDetector::onBang (C++ function), 104
PeakDetector::PeakDetector (C++ function), 102
PeakDetector::put (C++ function), 103
PeakDetector::reloadThreshold (C++ function), 103
PeakDetector::triggerThreshold (C++ function), 103
pq::mapFloat (C++ function), 107
pq::mapFrom01 (C++ function), 109
pq::mapTo01 (C++ function), 110
pq::randomFloat (C++ function), 111

pq::seconds (C++ function), 111
pq::wrap (C++ function), 112, 113
pq::wrap01 (C++ function), 113

R

Ramp (C++ class), 91
Ramp::addTime (C++ function), 93
Ramp::duration (C++ function), 92
Ramp::easing (C++ function), 91
Ramp::elapsed (C++ function), 93
Ramp::finished (C++ function), 93
Ramp::from (C++ function), 92
Ramp::fromTo (C++ function), 92
Ramp::get (C++ function), 91
Ramp::go (C++ function), 92
Ramp::isFinished (C++ function), 93
Ramp::isRunning (C++ function), 93
Ramp::mode (C++ function), 93
Ramp::noEasing (C++ function), 91
Ramp::onFinish (C++ function), 93
Ramp::pause (C++ function), 93
Ramp::progress (C++ function), 93
Ramp::Ramp (C++ function), 91
Ramp::resume (C++ function), 93
Ramp::setTime (C++ function), 93
Ramp::speed (C++ function), 92
Ramp::start (C++ function), 92, 93
Ramp::stop (C++ function), 93
Ramp::to (C++ function), 91

S

SineWave (C++ class), 73
SineWave::addTime (C++ function), 74
SineWave::amplitude (C++ function), 73
SineWave::atPhase (C++ function), 74
SineWave::bpm (C++ function), 73
SineWave::forward (C++ function), 74
SineWave::frequency (C++ function), 73
SineWave::get (C++ function), 73
SineWave::isRunning (C++ function), 74
SineWave::mapTo (C++ function), 74
SineWave::pause (C++ function), 75
SineWave::period (C++ function), 73
SineWave::phase (C++ function), 74
SineWave::resume (C++ function), 75
SineWave::reverse (C++ function), 74
SineWave::setTime (C++ function), 74
SineWave::shiftBy (C++ function), 74
SineWave::start (C++ function), 74
SineWave::stop (C++ function), 74
SineWave::toggleReverse (C++ function), 74
SineWave::width (C++ function), 73
Smoother (C++ class), 105
Smoother::cutoff (C++ function), 106
Smoother::get (C++ function), 105
Smoother::put (C++ function), 105
Smoother::Smoother (C++ function), 105
Smoother::timeWindow (C++ function), 106
SquareWave (C++ class), 76
SquareWave::addTime (C++ function), 79
SquareWave::amplitude (C++ function), 78
SquareWave::atPhase (C++ function), 78
SquareWave::atPhaseIsOn (C++ function), 77
SquareWave::bpm (C++ function), 78
SquareWave::forward (C++ function), 79
SquareWave::frequency (C++ function), 77, 78
SquareWave::get (C++ function), 77
SquareWave::isRunning (C++ function), 79
SquareWave::mapTo (C++ function), 79
SquareWave::pause (C++ function), 79
SquareWave::period (C++ function), 77
SquareWave::phase (C++ function), 78
SquareWave::resume (C++ function), 79
SquareWave::reverse (C++ function), 79
SquareWave::setTime (C++ function), 79
SquareWave::shiftBy (C++ function), 78
SquareWave::shiftByIsOn (C++ function), 77
SquareWave::SquareWave (C++ function), 77
SquareWave::start (C++ function), 79
SquareWave::stop (C++ function), 79
SquareWave::toggleReverse (C++ function), 79
SquareWave::width (C++ function), 78
StreamIn (C++ class), 124
StreamIn::get (C++ function), 125
StreamIn::mapTo (C++ function), 125
StreamIn::onUpdate (C++ function), 125
StreamIn::StreamIn (C++ function), 125
StreamIn::updated (C++ function), 125
StreamOut (C++ class), 126
StreamOut::get (C++ function), 126
StreamOut::precision (C++ function), 126
StreamOut::put (C++ function), 126
StreamOut::StreamOut (C++ function), 126

T

TriangleWave (C++ class), 81
TriangleWave::addTime (C++ function), 83
TriangleWave::amplitude (C++ function), 82
TriangleWave::atPhase (C++ function), 82
TriangleWave::bpm (C++ function), 81, 82
TriangleWave::forward (C++ function), 83
TriangleWave::frequency (C++ function), 81
TriangleWave::get (C++ function), 81
TriangleWave::isRunning (C++ function), 83
TriangleWave::mapTo (C++ function), 83
TriangleWave::pause (C++ function), 83
TriangleWave::period (C++ function), 81
TriangleWave::phase (C++ function), 82

TriangleWave::resume (*C++ function*), 83
TriangleWave::reverse (*C++ function*), 83
TriangleWave::setTime (*C++ function*), 82
TriangleWave::shiftBy (*C++ function*), 82
TriangleWave::start (*C++ function*), 83
TriangleWave::stop (*C++ function*), 83
TriangleWave::toggleReverse (*C++ function*), 83
TriangleWave::width (*C++ function*), 82