
Pixelblaze Language Reference

The language reference below is current for firmware v3.30 (for v3 devices) and v2.24 (for Pixelblaze v2). Consult the Edit tab of your Pixelblaze's browser app for the language reference applicable to the version you're running.

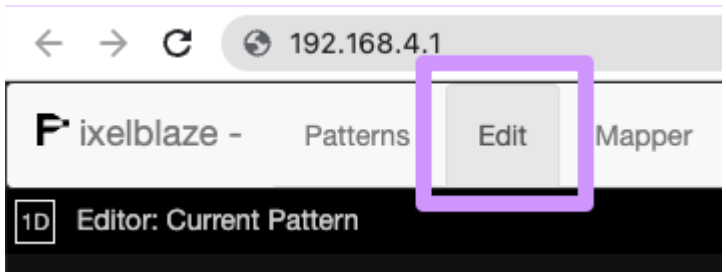

Table of Contents

Writing Patterns

Enter code above, and everything you type is compiled on the fly. If your program is valid, it is sent down to Pixelblaze, and you'll see your changes instantly! Look for syntax or run-time errors in the left sidebar and the status area just below the editor.

Pixelblaze looks for a few exported functions that it can call to generate pixels.

The exported `render(index)` function is called for each pixel in the strip. The `index` argument tells you which pixel is being rendered. Use the `hsv` or `rgb` function to set the current pixel's color.

Alternative render functions `render2D(index, x, y)` and `render3D(index, x, y, z)` can also be specified to use when a 2D or 3D pixel map is available. Multiple alternatives can be in the same program and the appropriate one will be selected automatically. See the Mapper tab for more info.

The exported `beforeRender(delta)` function is called before it is going to render a new frame of pixels to the strip. The `delta` argument is the number of elapsed milliseconds (with a resolution of 6.25ns!) since the last time `beforeRender` was called. You can use `delta` to create animations that run at the same speed regardless of the frame rate.

Pixelblaze's language is based on JavaScript (ES6) syntax, but with a subset of the language features available. All numbers in Pixelblaze are 16.16 fixed-point numbers. This can handle values between -32,768 to +32,768 with fractional accuracy down to 1/65,536ths.

A global called `pixelCount` is defined based on how many pixels you've configured in settings. You can use this in initialization code or in any function.

Watching Variables

Next to the editor is a **Var Watcher** which will display the value of any variable that has been exported. It works for arrays as well, and will show each element with the corresponding index.

Data from the sensor expansion board can be watched as well. Use this to explore data from ADC inputs, accelerometer, light sensor, or all of the sound analysis data.

The data is only sampled at the end of rendering. If you export a global variable that is modified inside `render()` normally only the last value will be shown. If you want to inspect a bit of data only for a particular pixel index, you can conditionally set that variable. e.g.: `if (index == 42) myExportedVar = someInterestingData`

User Interface Controls

Custom interface controls can be created that will show up when a pattern is active. This can be used to change how the pattern behaves without editing the code. These are persistent and their settings will be preserved across restarts or pattern switches.

Some controls allow input, while others can be used to display some information.

Create a control by exporting a function with a special name prefix (see below), folowed by the name you would like to use. CamelCase or snake_case can be used to separate words.

Whenever input controls are changed, the function will be called with the new value. It will also be called with a saved value when the pattern is switched to before rendering occurs. To make use of this you may store the value or use it to calculate something used in your pattern.

Output controls work by calling the function and using the return value. These may be called very frequently to refresh the interface.

Range Sliders

To create a slider, export a function that starts with the key word `slider` followed by the name of the slider. For example, to create a slider called "My Slider":

```
export function sliderMySlider(v) {...}
```

Whenever the slider is moved, this function will be called with a new value between 0.0 and 1.0. It will also be called with a saved value when the pattern is switched to before rendering occurs.

To make use of this you may store the value or use it to calculate something used in your pattern. A common way to do this is to create a variable to hold the value and initialize it with a default value:

```
var mySetting = 0.5
  export function sliderMySetting(v) {
      mySetting = v
  }
```

Color Pickers

To create a color picker, export a function that starts with the key words `hsvPicker` or `rgbPicker` followed by the name of the color picker. For example, to create a color picker called "Primary Color":

For HSV:

```
export function hsvPickerPrimaryColor(h, s, v) {...}
```

For RGB:

```
export function rgbPickerPrimaryColor(r, g, b) {...}
```

This creates a color well that when clicked will open a color picker. When it is changed, this function will be called with either the hue, saturation, and value for `hsvPicker` or the red, green, and blue for `rgbPicker`. All values are between 0.0 and 1.0, and suitable for passing to the `hsv()` or `rgb()` functions later on to set a pixel color.

Toggle Switches

To create a toggle switch, export a function that starts with the key word `toggle` followed by the name of the toggle switch. For example, to create a toggle switch called "Enable Awesomeness":

```
export function toggleEnableAwesomeness(isEnabled) {...}
```

Whenever the toggle is toggled on or off, this function will be called with a value of `true` (1) when turned on and `false` (0) when turned off.

Trigger Buttons

To create a trigger button, export a function that starts with the key word `trigger` followed by the name of the button. For example, to create a button called "Fire Lasers":

```
export function triggerFireLasers() {...}
```

Whenever the button is pressed, this function will be called. Unlike other input controls, this has no parameters and is not called when the patter first loads.

Inputs for Numbers

To create an input for accepting numbers, export a function that starts with the key word `inputNumber` followed by the name of the input. For example, to create a input called "Scale":

```
export function inputNumberScale(v) {...}
```

Whenever a number is entered, this function will be called with the new value. Any positive or negative number, either whole or a decimal is allowed. It will also be called with a saved value when the pattern is switched to before rendering occurs.

Showing Numbers

To show a number, export a function that starts with the key word `showNumber` followed by the name. For example, to display a number called "Energy Average":

```
export function showNumberEnergyAverage() {return ...}
```

The returned number can be any number, and will be displayed with 4 digits after the decimal point. This function will be called frequently to update the interface.

Gauges

To show a gauge, export a function that starts with the key word `gauge` followed by the name. For example, to display a gague called "Light Level":

```
export function gaugeLightLevel() {return ...}
```

The returned number should be between 0.0 and 1.0 and are scaled as percentages. Values outside of this range are OK, but will be clamped to the minimum or maximum. This function will be called frequently to update the interface.

Supported Language Features

- All of the usual math operators work. Most work on 16.16 fixed-point math. `=`, `+`, `-`, `!`, `*`, `/`, `%`, `>>`, `<<`, `|`, `&`, `~`, `^`, `>`, `<`, `>=`, `<=`, `==`, `!=`, `||`, `&&`, `?`. Most bitwise operators work on all 32 bits (16 in the integer part, and 16 in the fractional part), which is different from JavaScript. The exception is `~` which zeros out the lower 16 bits.
- Logical operators work like JavaScript and carry over the value, not just a boolean. e.g. `v = 0 || 42` will result in 42.
- Trig and other math functions. `abs`, `floor`, `ceil`, `min`, `max`, `clamp`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`, `sqrt`, `exp`, `log`, `log2`, `pow`, `random`
- Declare global or function-local variables using `var` or globals implicitly.
- Use `if` and `else` to have some code run conditionally.
- Use `while` and `for` for making loops and use `break` and `continue` statements to escape a loop or loop early.
- Define functions using the `function` keyword or short lambda-style form.
  - `function myFunction(arg1) {return arg1 * 2}`
  - `myFunction = (arg1) => arg1 * 2`
- Functions can be stored in variables, passed as arguments, and returned from other functions.
- Create arrays using the `array(size)` function or array literals and access them with the bracket syntax. Arrays can be passed around just like any other type.

Language limitations

These are language features you'd expect to work writing JavaScript that won't run on Pixelblaze.

- Objects, named properties, classes, etc.
- Garbage collection or freeing memory. Arrays are currently the only dynamically allocated memory, and you can't (yet) free them once created.
- Closures aren't supported, so any function defined in another function won't have access to parameters or local variables. It can still access globals or its own parameters.
- `switch` + `case` statements. You can use chained `else if` statements, or put functions in an array and use it as a lookup table.

```
modes[0] = () => {/* do mode 0 */};
  modes[1] = () => {/* do mode 1 */};
  // ...
  modes[currentMode]();
```

- Narrow scoped variables using `let` or read-only variables with `const`

Variables

The `pixelCount` variable is available as a global even during initialization. This is the number of LED pixels that have been configured in settings.

Variables can be created/assigned implicitly with the `=` operator. e.g.: `foo = wave(time(0.1))` or explicitly using the `var` keyword. e.g.: `var foo = 1`.

Explicitly declared variables will either be global or local depending on where they are declared. Local variables declared using `var` inside a function are visible inside that function. Local variables can shadow global variables with the same name.

Implicitly defined variables are always globally scoped, even if first assigned inside a function. e.g.: `function() {bar = 123}` will define `bar` as a global while `function() {var baz = 123}` defines `baz` as a local variable since it uses the `var` keyword inside a function.

Global variables can also be exported with the `export` keyword. These will be visible in the Var Watcher and can be used with the `getVars` and `setVars` websocket API.

Constants

`E` , `PI`, `PI2` (PI * 2), `PI3_4` (PI * 3 / 4), `PISQ` (PI * PI), `LN2`, `LN10`, `LOG2E`, `LOG10E`, `SQRT1_2`, `SQRT2`

Functions

Math Functions

abs(v)

Returns the absolute value. `abs(-2) == 2`

acos(x)

Returns the arccosine (in radians) of a number.

asin(x)

Returns the arcsine (in radians) of a number.

atan(x)

Returns the arctangent (in radians) of a number.

atan2(y, x)

This returns the angle (in radians) between the positive x axis and the line to the point `(x, y)`.

ceil(v)

Rounds up to the next largest integer. `ceil(5.1) == 6`, `ceil(-5.9) == -5`

clamp(value, low, hi)

Clamps `value` such that it isn't less than `low` or greater than `high`

cos(angleRads)

Returns the cosine of the specified angle (in radians).

exp(x)

Returns e^x, where x is the argument, and e is Euler's number.

floor(v)

Rounds down to the next smallest integer. `floor(5.9) == 5`, `floor(-5.1) == -6`

frac(v)

Returns the fractional component of a number. `frac(5.5) == 0.5`, `frac(-5.5) == -0.5`

hypot(x,y)

Calculate the square root of the sum of the squares of x and y, which is the hypotenuse of a right triangle with sides `x` and `y`, and the distance of the point `(x,y)` from the origin `(0,0)`. `hypot(3, 4) == 5`

hypot3(x,y,z)

Calculate the square root of the sum of the squares of x, y, and z. This can be used to find the distance of the point `(x,y,z)` from the origin `(0,0,0)`.

log(v)

Returns the natural logarithm (base e) of a number.

log2(v)

Returns the base 2 logarithm of a number. `log2(256) == 8`

max(v1, v2)

Returns the larger of two numbers.

min(v1, v2)

Returns the smaller of two numbers.

mod(x,y)

The floored remainder of the division `x/y`. The result uses the same sign as `y`. For example `mod(-3.5, 3) == 2.5` whereas `-3.5 % 3 == -.5`. This provides the same "wrapping" behavior used in the animation functions like `triangle` when `y` == 1.

pow(base, exponent)

Returns the `base` to the `exponent` power, that is, $base^{exponent}$. `pow(10,2) == 100`

random(max)

A random number between 0.0 and `max` (exclusive).

prng(`max`)

A pseudorandom number generator for values between 0.0 and `max` (exclusive). Given the same initial seed with `prngSeed()`, calls to `prng()` produce the same sequence of numbers.

prngSeed(`seed`)

Set a seed for use with `prng()`. The old seed is returned.

round(`v`)

Returns the value of a number rounded to the nearest integer.

sin(`angleRads`)

Returns the sine of an angle (in radians).

sqrt(`v`)

Returns the square root of a number. `sqrt(9) == 3`

tan(`angleRads`)

Returns the tangent of an angle (in radians).

trunc(`v`)

Returns the integer component of a number. `trunc(5.5) == 5`, `trunc(-5.5) == -5`

Array Functions

Array functions can also be accessed as methods and arrays support the read-only length property.

array(`n`)

Create a new array with `n` elements.

arrayForEach(`a`, `fn`) ↔ `a`.forEach(`fn`)

Iterate over an array and call `fn(value, index, array)` for each element.

arrayLength(`a`) ↔ `a`.length

Return the length/size of an array. Note that the `a`.length form is not a function call.

arrayMapTo(`src`, `dest`, `fn`) ↔ `src`.mapTo(`dest`, `fn`)

Iterate over the `src` array and call `fn(value, index, array)` for each element. Return values are then stored in `dest`. The `dest` array may be smaller than `src`, in which case extra results are calculated and then discarded.

arrayMutate(a, fn) ↔ a.mutate(fn)

Modify an array in place by calling `fn(value, index, array)` for each element and storing the return values.

arrayReduce(a, fn, initialValue) ↔ a.reduce(fn, initialValue)

Returns a value by calling `fn(accumulator, value, index, array)` (the reducer) for each element of the array, resulting in a single value (the accumulator). The `accumulator` parameter is seeded with `initialValue` and updated with the return value from each call to `fn`. Examples:

- Sum: `arrayReduce(a, (acc, v) => acc + v, 0)`
- Max: `arrayReduce(a, (acc, v) => max(acc,v), a[0])`
- Count instances: `key = 5; arrayReduce(a, (acc, v) => acc + (v == key), 0)`
- Index of closest: `target = 5; arrayReduce(a, (acc, v, i, a) => abs(target - v) < abs(target - a[acc]) ? i : acc , 0)`

arrayReplace(a, ...) ↔ a.replace(...)

Replace the elements of an array with any number of arguments, starting at index 0. The argument list must fit into the array. If the array is larger than the arguments, the remaining array elements are unchanged.

arrayReplaceAt(a, offset, ...) ↔ a.replace(offset, ...)

Replace some elements of an array with any number of arguments, starting at `offset`. The argument list must fit into the array at the given `offset`. The other array elements are unchanged.

arraySort(a) ↔ a.sort()

Sort an array of numbers in ascending order.

arraySortBy(a, fn) ↔ a.sortBy(fn)

Sort an array using `fn(v1, v2)` to compare element values. The compare function should return a negative number if `v1` is less than `v2`. The order of equal elements is not guaranteed to be preserved.

arraySum(a) ↔ a.sum()

Returns the sum of all elements in an array. Tip: this can be used to calculate an average: `average = arraySum(a) / arrayLength(a)`

Waveform Functions

time(interval)

A sawtooth waveform between 0.0 and 1.0 that loops about every 65.536*`interval` seconds. e.g. use .015 for approximately 1 second.

Patterns using this can be synchronized across the network using either Firestorm, or when connecting to a Pixelblaze in AP mode.

wave(v)

Converts a sawtooth waveform `v` between 0.0 and 1.0 to a sinusoidal waveform between 0.0 to 1.0. Same as `(1+sin(v*PI2))/2` but faster. `v` "wraps" between 0.0 and 1.0.

square(v, duty)

Converts a sawtooth waveform `v` to a square wave using the provided `duty` cycle where `duty` is a number between 0.0 and 1.0. `v` "wraps" between 0.0 and 1.0.

triangle(v)

Converts a sawtooth waveform `v` between 0.0 and 1.0 to a triangle waveform between 0.0 to 1.0. `v` "wraps" between 0.0 and 1.0.

mix(low, high, weight)

Returns linear interpolation between `low` and `high` given a `weight` between 0.0 and 1.0.

smoothstep(low, high, v)

Returns a smooth Hermite interpolation between 0.0 and 1.0 based on `v` crossing between `low` and `high` thresholds. `High` must be >= `low`, `v` may exceed this range and clamps at 0.0 or 1.0 respectively. This can be used to ease in and out while comparing a value in an arbitrary range.

bezierQuadratic(t, p0, p1, p2)

Return a quadratic bezier curve at `t` given the start point `p0`, control point `p1` and end point `p2`.

bezierCubic(t, p0, p1, p2)

Return a cubic bezier curve at `t` given the start point `p0`, control points `p1`, `p2`, and end point `p3`.

perlin(x, y, z, seed)

Generate 3D Perlin noise. Every integer value produces a different random result, with smooth transitions between them. The values will repeat every 256 or as specified with `setPerlinWrap()`, and will seamlessly wrap.

perlinFbm(x, y, z, lacunarity, gain, octaves)

Generate 3D fractal Perlin noise (fractial Brownian Motion). The values will repeat every 256 or as specified with `setPerlinWrap()`, and can seamlessly wrap. The `lacunarity` controls the distance between octaves and should be set to 2 or another integer value if wrapping is desired. The `gain` controls the strength between each octave, try values around 0.5-0.8.

perlinRidge(x, y, z, lacunarity, gain, offset, octaves)

Generate 3D fractal ridged Perlin noise. The values will repeat every 256 or as specified with `setPerlinWrap()`, and can seamlessly wrap. The `lacunarity` controls the distance between octaves and should be set to 2 or another integer value if wrapping is desired. The `gain` controls the strength between each octave, try values around 0.5-0.8. The `offset` is used to invert the ridges, values around 1.0 work well.

perlinTurbulence(x, y, z, lacunarity, gain, octaves)

Generate 3D fractal turbulent Perlin noise. The values will repeat every 256 or as specified with `setPerlinWrap()`, and can seamlessly wrap. The `lacunarity` controls the distance between octaves and should be set to 2 or another integer value if wrapping is desired. The `gain` controls the strength between each octave, try values around 0.5-0.8.

setPerlinWrap(x, y, z)

Causes perlin functions to wrap at the given integer intervals between 2 and 256. Useful for creating textures that can repeat smoothly, while controlling density.

Pixel / Color Functions

hsv(`hue`, `saturation`, `value`)

Sets the current pixel by calculating the RGB values based on the HSV color space. `Hue` "wraps" between 0.0 and 1.0. Negative values wrap backwards. For LEDs that support it, this uses 24-bit color plus an additional 5 bits of brightness control giving a high dynamic range especially at lower light levels and reduces posterization.

hsv24(`hue`, `saturation`, `value`)

Sets the current pixel by calculating the RGB values based on the HSV color space. `Hue` "wraps" between 0.0 and 1.0. Negative values wrap backwards. This uses 24-bit color only, even if the LEDs support additional resolution and may reduce flickering in some LEDs.

rgb(`red`, `green`, `blue`)

Sets the current pixel to the RGB value provided. Values range between 0.0 and 1.0.

setPalette(`array`)

Sets the palette to a gradient based on an array of positions and RGB values. For example, this fades from black to magenta to cyan:

```
var rgbGradient = [
    0,    0, 0, 0, //position start, rgb(0,0,0)
    0.75, 1, 0, 1, //position 75%, rgb(1,0,1)
    1,    0, 1, 1  // position end, rgb(0,1,1)
  ]
  setPalette(rgbGradient)
```

paint(`value`, [`brightness = 1`])

Sets the current pixel to a color based on the `value`'s position in the current palette. `Value` "wraps" between 0.0 and 1.0. Negative values wrap backwards. Optionally `brightness` can be specified.

Coordinate Transformation Functions

Coordinate transformations allow you to manipulate the pixel map coordinates by translating (moving), scaling, and rotating. Up to 31 transformations can be applied. These APIs affect the next render cycle, and can be called in `beforeRender` or in the main body of code.

resetTransform()

Resets coordinate transforms to the default. Use this before setting up new transformations.

transform(`m11`, `m21`, `m31`, `m41`, `m12`, `m22`, `m32`, `m42`, `m13`, `m23`, `m33`, `m43`, `m14`, `m24`, `m34`, `m44`)

Applies an arbitrary 4x4 matrix transform.

translate(`x`, `y`)

Move in 2D space.

scale(`x`,`y`)

Scale in 2D space. This increases the density of pixels, such that `scale(2,2)` will cause things to appear half as large.

rotate(`angleRads`)

Rotate 2D space (around the Z axis), by an angle (in radians).

translate3D(`x`, `y`, `z`)

Move in 3D space.

scale3D(`x`, `y`, `z`)

Scale in 3D space. This increases the density of pixels, such that `scale3D(2,2,2)` will cause things to appear half as large.

rotateX(`angleRads`)

Rotate 3D space around the X axis by an angle (in radians).

rotateY(`angleRads`)

Rotate 3D space around the Y axis by an angle (in radians).

rotateZ(`angleRads`)

Rotate 3D space around the Z axis by an angle (in radians).

Pixel Map Functions

pixelMapDimensions()

Return the number of dimensions in the pixel map, e.g. 2 for 2D, or 0 for no map.

has2DMap()

Returns true if there is a 2D pixel map.

has3DMap()

Returns true if there is a 3D pixel map.

mapPixels(`fn`)

Walk through the pixels with pixel map coordinates. The given `fn` is invoked with 4 arguments: (`index`, `x`, `y`, `z`). If no pixel map is installed, `x` will be the same as `index/pixelCount`, and `y` and `z` will be 0. For 2D pixel maps `z` will be 0.

For 2D and 3D maps the current coordinate transformations are applied before `fn` is called.

Input / Output Functions

readAdc()

Reads the value from the ADC as a number between 0.0 and 1.0. *This function is only available on V2 devices*

analogRead(`pin`)

Reads the value from the pin as a number between 0.0 and 1.0. *This function is only available on V3 devices*

pinMode(`pin`,`mode`)

Set the pin mode as an `INPUT`, `INPUT_PULLUP`, `INPUT_PULLDOWN`, `OUTPUT`, `OUTPUT_OPEN_DRAIN`, or `ANALOG`.

V2 device notes: `INPUT_PULLUP` does not work for GP16, instead `INPUT_PULLDOWN_16` is supported. This can be used with a button, connected between GP16 and 3.3v. Pins will read `HIGH` or 1.0 while the button is pressed. On Pixelblaze V2+ pin GP12 is connected to the orange LED. GP2 is used for NeoPixel and WS2811/12/13 support and can't be used unless the trace on the bottom is cut.

digitalWrite(`pin`,`state`)

Set a pin `HIGH` or `LOW`. Any non-zero value will set the pin `HIGH`.

digitalRead(`pin`)

Read a pin state, returns 1.0 if the pin is `HIGH`, 0.0 for `LOW` otherwise.

touchRead(`pin`)

Detect touch and proximity on a pin using capacitive sensing techniques. Returns a value between 0.0 and 1.0 depending on how much capacitance is detected on the pin.

V3 device notes: You can also specify one of these constants: T0, T2, T4, T6, T7.

Clock / Time Functions

When the discovery service is enabled and Pixelblaze is connected to the internet, it will know what time it is and these functions can be used.

clockYear()

clockMonth()

clockDay()

clockHour()

24 hour format. `13` = 1pm.

clockMinute()

clockSecond()

clockWeekday()

Sunday = 1, Monday = 2, etc.

Sync Group Functions

nodeId()

Returns the integer node ID as configured in settings. This can be used to change the behavior within a group.

Sensor Expansion Board

Pixelblaze supports a sensor expansion board that adds:

- A microphone and signal processing that gives:
  - `frequencyData` - 32 element array with frequency magnitude data ranging from 12.5-10khz
  - `energyAverage` - total audio volume
  - `maxFrequency` and `maxFrequencyMagnitude` - detects the strongest tones with resolution of about 39Hz
- A 3-axis 16G `accelerometer` - 3 element array with [x,y,z]
- An ambient light sensor - `light` can be used to automatically dim displays for nightlights
- 5 `analogInputs` - 5 element array with analog values from A0-A4

Each of these can be accessed in a pattern by using the `export var` syntax, with optional defaults if the board is not connected.

```
export var frequencyData
  export var energyAverage
  export var maxFrequencyMagnitude
  export var maxFrequency
  export var accelerometer
  export var light
  export var analogInputs
```