Digidraw

A package for drawing digital timing diagrams

Digidraw is a Typst package to help drawing digital timing diagrams. The syntax tries to be like from *WaveDrom*. The package allows for additional styling such as colors and fonts.

The package uses CeTZ to draw everthing.

Caution

Not all symbol combinations have been implemented so far (probably), so expect some undefined or unexpected behaviour.

Please report them in the package repository:
https://codeberg.org/joelvonrotz/typst-#package-name/

Contents

Quickstart

From Universe (aka. Stable Version)

To import the Universe version, insert following snippet

```
#import "@preview/digidraw:0.9.0"
```

From Repository (aka. Development Version)

The repository will always be the *development* version of Digidraw. This will also include the latest changes, but might break here and there.

I. Clone the Repository into your project (for example into a `./packages/` folder or at root level).

```
cd /path/to/project/
git clone https://codeberg.org/joelvonrotz/typst-digidraw.git
```

OR if your project is a git project, you can add it as a submodule, saving a little bit of space in your repository:

```
cd /path/to/project/
git submodule add https://codeberg.org/joelvonrotz/typst-digidraw.git
```

II. Include the package into your document

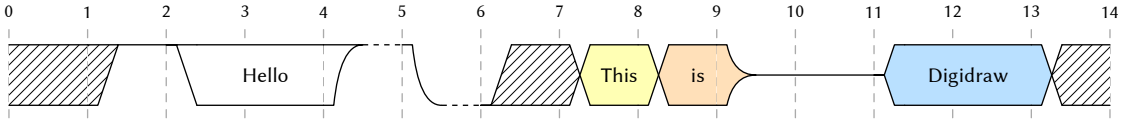
```
#import "./path/to/typst-digidraw/lib.typ" as digidraw
```

Usage

Drawing Diagrams

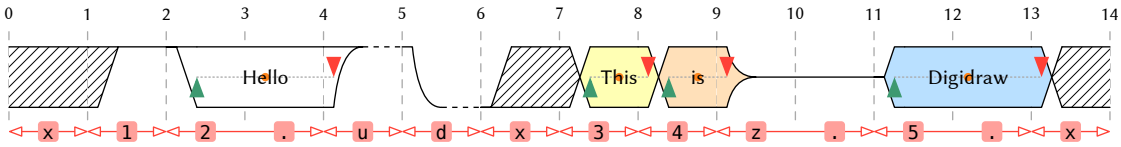
Drawing a diagram is easy:

```
#digidraw.wave(
  (signal: (
    (wave: "x12.udx34z.5.x",data: "Hello This is Digidraw"),)
  )
)
```



You can turn on Debug mode by setting `debug : true`. This shows where and which symbol is placed. Additionally, it shows the exact placement of bus labels: start reference ▲ and end reference ▼ and inbetween the label position ● are shown.

```
#digidraw.wave(
  (signal: (
    (wave: "x12.udx34z.5.x",data: "Hello This is Digidraw"),)
  ),
  debug: true
)
```



Check out the chapters *Symbols* and *API Reference* for more.

Data Structure

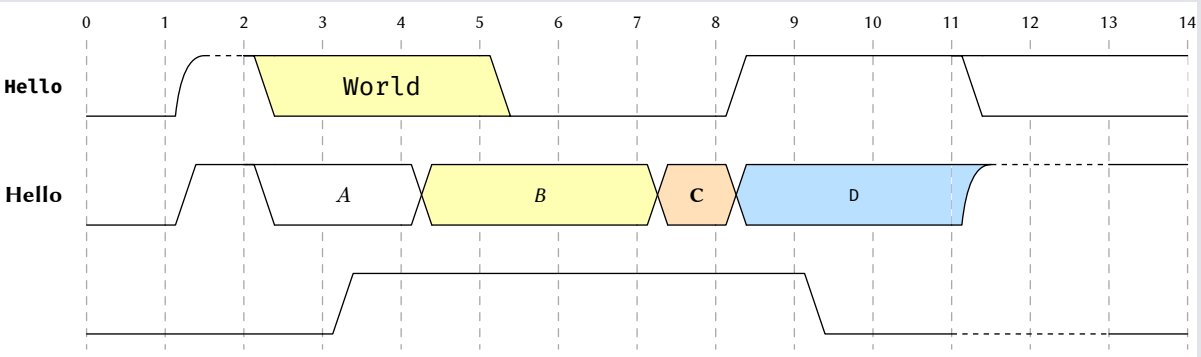
Let's talk about wave data structures! The data syntax is the one from *WaveDrom*. You can copy WaveDrom scripts and insert those into your document. Well, almost... You have to format it according to the JSON standard!

There's two ways on how to define the wave data: as dictionaries or as JSON structures.

Dictionaries

This allows for fancier styling labels. `.wave` and `.data` can contain `content`. *Raw*'em, *Strong*'em, *Emph*'em, do what you want, as long as it turns into a `content`!

```
#let data = (signal: (
  (wave: "0u3..0..1..2..", name: `Hello`, data: (text(1.5em,`World`),)),
  (wave: "012.3..45..u.1", name: "Hello", data: ($A$, [_B_], [*C*], `D`)),
  (wave: "0..1.....0.d.0"),
))
#digidraw.wave(data)
```

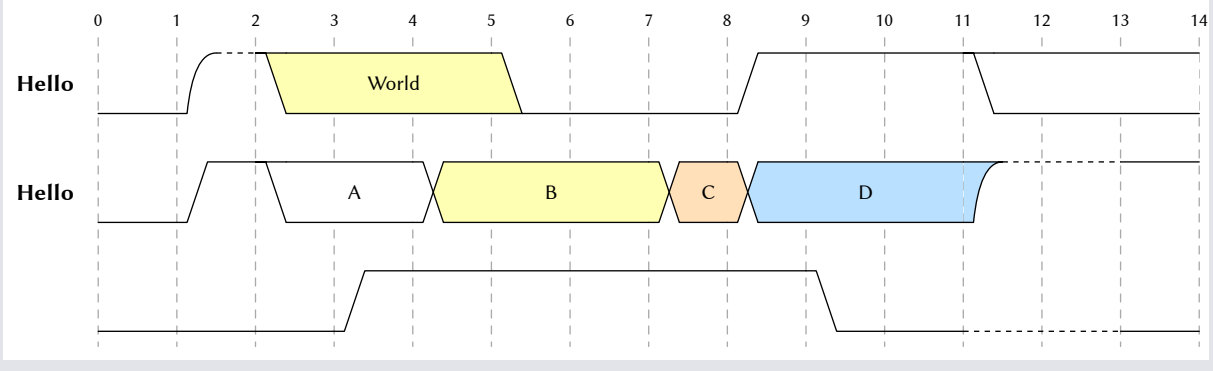


JSON

Easier when copying from WaveDrom with the downside of not having much styling options.

```
{"signal": [
  {"wave": "0u3..0..1..2..", "name": "Hello", "data": ["World"]},
  {"wave": "012.3..45..u.1", "name": "Hello", "data": "A B C"},
  {"wave": "0..1.....0.d.0"}
  // more waves
]}
```

```
#let data = json(bytes(`{"signal": [
  {"wave": "0u3..0..1..2..", "name": "Hello", "data": ["World"]},
  {"wave": "012.3..45..u.1", "name": "Hello", "data": "A B C D"},
  {"wave": "0..1.....0.d.0"}
]}`.text)) // or json(read("file.json"))
#digidraw.wave(data)
```



`.signal` `array` of `dictionary`

A list/array containing the wave objects.

`.wave` `string`

The wave's contents built using the *Symbols*.

Example: `"10..u.10"`

`.name` `string` or `content`

The name of the wave.

Example: `"Mike"`

`.data` `string` or `array` of `content` or `array` of `string`

JSON, Typst | Typst | JSON, Typst

When buses are defined in the wave, labels can be inserted. The index of the label defines its position: first entry goes into the first bus, second in the second one, etc. If `data` is a string, labels are split by spaces. `"Hello World"` is equal to `("Hello", "World")`

> **Important**
>
> Everything else from the WaveDrom syntax either hasn't been implemented yet or will not see the light of day. Help is very much appreciated or check out the README todo list!

Configuration

Digidraw offers a lot of configurations through a configuration dictionary. For options and their effects, see Section *API Reference*.

Global Configurations

Global configurations are done via `#wave.with()` and placed after the import statement.

Well you can say this isn't really how «global» should be. This was done, so I didn't have to use any `state` components!

```
#let wave = digidraw.wave.with(
  wave-height: 2,
  symbol-width: 3,
  /* ... */
)
```

> **Using in Multi-document project**
>
> Usually in multi document projects, you probably have a *preamble* file containing global imports. In there you define the global configuration.

Local Configurations

Local configurations are done in the `#wave`-function.

```
#digidraw.wave(
  wave-height: 2,
  symbol-width: 3,
  /* ... */

  (signal: /* ... */)
)
```

Reseting Configurations

Configurations can be reset to their default value by passing `auto` to the respective parameter.

```
#let wave = digidraw.wave.with(
  wave-height: 2,
)
/* ... */
#let wave = wave.with(
  wave-height: auto
)
```

Symbols

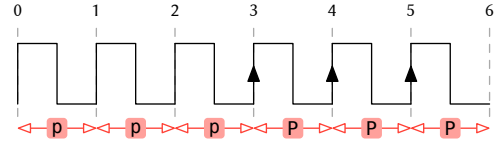
Type	Symbols
Clocks	`p` `P` `n` `N`
Buses	`x` `=` `2` `3` `4` `5` `6` `7` `8` `9`
Wires	`l` `L` `h` `H` `0` `1` `z` `u` `d`
Extender	`.`
Time Skip	`|`
Invalid Symbols	`/` and everything else

Clocks

Normal Clock `p` `P`

Capitalizing the letters adds an arrow mark to the line pointing up.

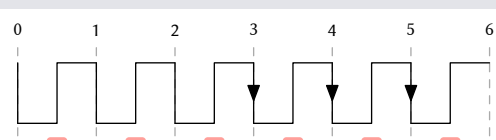
```
#digidraw.wave(
  (signal: ((wave: "pppPPP",),)),
  debug: true
)
```



Inverted Clock `n` `N`

Capitalizing the letters adds an arrow mark to the line pointing down.

```
#digidraw.wave(
  (signal: ((wave: "nnnNNN",),)),
  debug: true
)
```

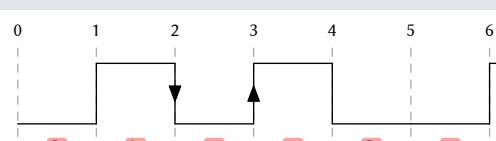


Wires

Straight Signals `l` `L` `h` `H`

Capitalizing the letters adds an arrow mark to the line pointing down.

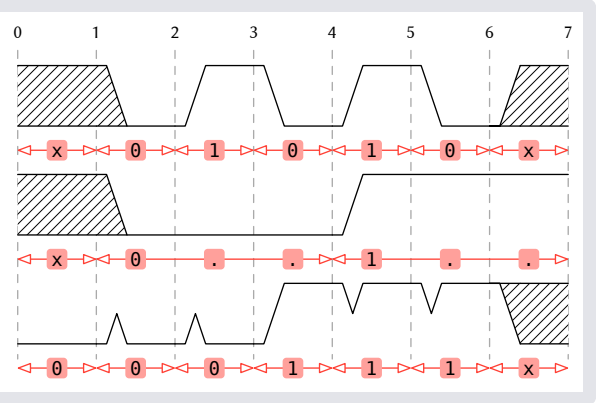
```
#digidraw.wave(
  (signal: ((wave: "lhLHlLh",),)),
  debug: true
)
```



Flanked Signals `0` `1`

Using `0` and `1` instead of `l`/`L` and `h`/`H` inserts flanked/sloped signals.

```
#digidraw.wave(
  (signal: (
    (wave: "x01010x",),
    (wave: "x0..1..",),
    (wave: "000111x",),
  )),
  debug: true
)
```

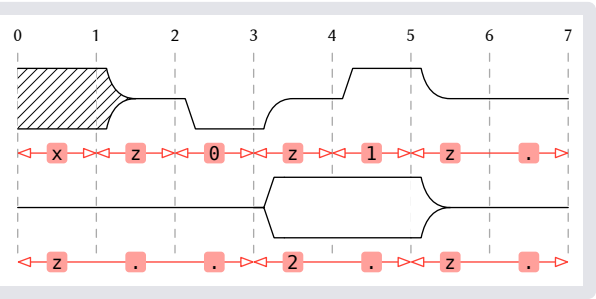


High-Impedance Signal `z`

The high impedance signal introduces curved transitions from any signal to high impedance.

Note that multiples of the symbols are combined
Example: `"zzz2.zz"` → `"z..2.z."`

```
#digidraw.wave(
  (signal: (
    (wave: "xz0z1z.",),
    (wave: "zzz2.zz",)
  )),
  debug: true
)
```

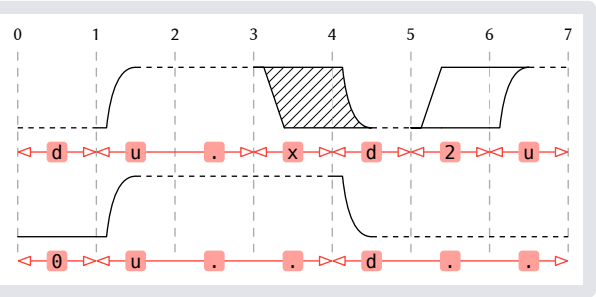


Transition Edges `u` `d`

If the transition has of unknown duration/delay, a transition edge can be used.

Note that multiples of the symbols are combined
Example: `"uuuddduuu"` → `"u..d..u.."`

```
#digidraw.wave(
  (signal: (
    (wave: "du.xd2u",),
    (wave: "0uuudd",),
  )),
  debug: true
)
```



Buses

Buses are grouped into two groups: *don't care* and data buses.

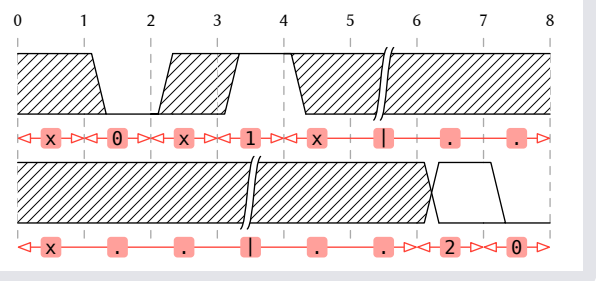
Don't Care `x`

Indicates a section, where the signal value is not important, aka. *don't care*.

Note that multiples of the symbols are combined

Example: `"xxx|xxx20"` → `"x.. | ... 20"`

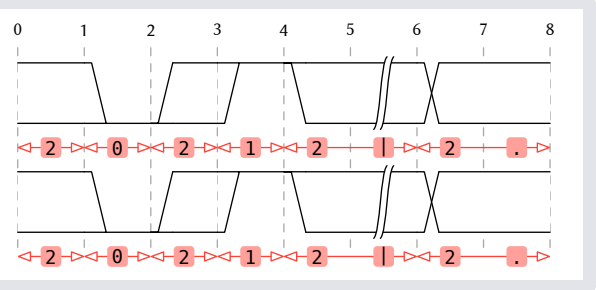
```
#let data = (signal: (
  (wave: "x0×1x|x."),
  (wave: "xxx|xx20"),
),)

#digidraw.wave(data, symbol-width: 1.1,
debug: true)
```



Simple Bus `=` `2`

= and 2 are both the same and respectively handled that way (= gets turned into 2).

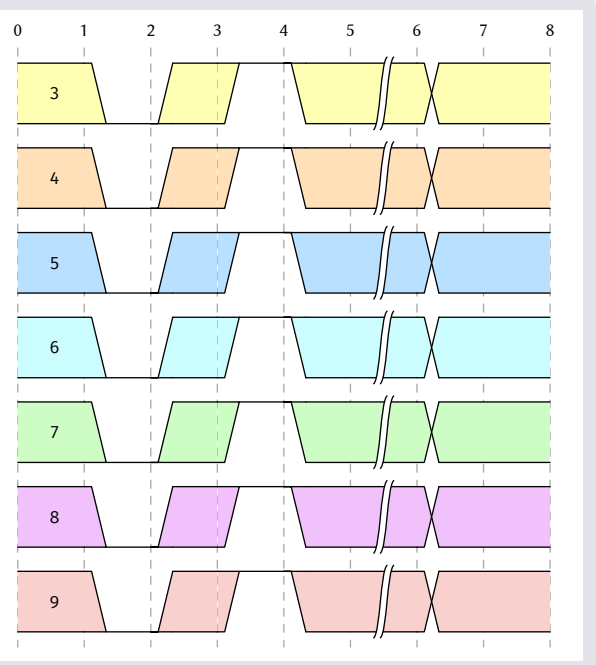
```
#let data = (signal: (
  (wave: "20212|2."),
  (wave: "=0=1=|=."),
),)

#digidraw.wave(data, symbol-width: 1.1,
debug: true)
```



Coloured Buses `3` `4` `5` `6` `7` `8` `9`

The databus can be also colored using numbers `"3"` through `"9"`.

```
#let data = (signal: (
  (wave: "30313|3.", data: (`3`,)),
  (wave: "40414|4.", data: (`4`,)),
  (wave: "50515|5.", data: (`5`,)),
  (wave: "60616|6.", data: (`6`,)),
  (wave: "70717|7.", data: (`7`,)),
  (wave: "80818|8.", data: (`8`,)),
  (wave: "90919|9.", data: (`9`,)),
),)

#digidraw.wave(data, symbol-width: 1.1,
wave-gutter: 0.4)
```



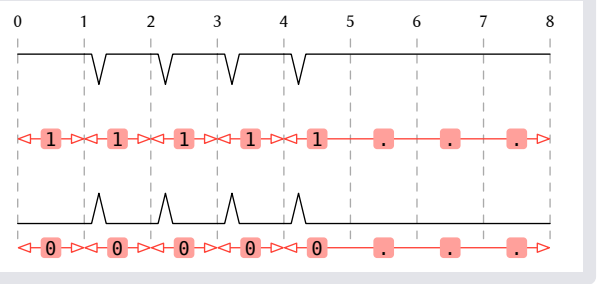
Extender `.`

The extender can be used to extend signals. If a signal doesn't look right, you might have to replace some of the symbols with the extender. And if that doesn't work, report it!

For example, while a `"1111"` inserts high signals with impulses, `"1...."` stretches.

Note Some signals are automatically modified with extenders, such as `"zzz"` → `"z.."`

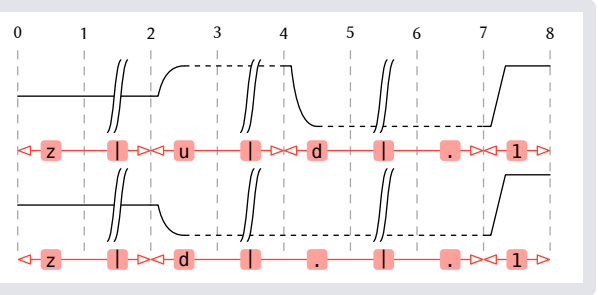
```
#let data = (signal: (
  (wave: "11111...."),
  (wave: "00000...."),
))

#digidraw.wave(data, symbol-width: 1.1,
debug: true)
```



Time Skip `|`

To insert a timeskip to indicate the passing of a long time, insert the symbol `|`.

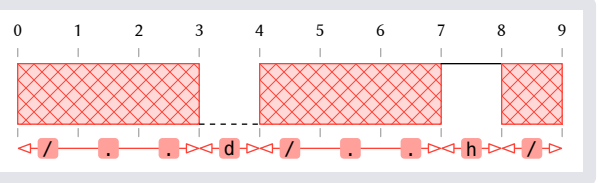
```
#let data = (signal: (
  (wave: "z|u|d|.1"),
  (wave: "z|d|d|.1"),
),)

#digidraw.wave(data, symbol-width: 1.1,
debug: true)
```



Invalid Symbols `/`

Any other symbol will result in the `/` symbol. This indicates a default or invalid character.

```
#let data = (signal: (
  (wave: "abcdefghi"),
),)

#digidraw.wave(data, symbol-width: 1,
debug: true)
```



API Reference

wave

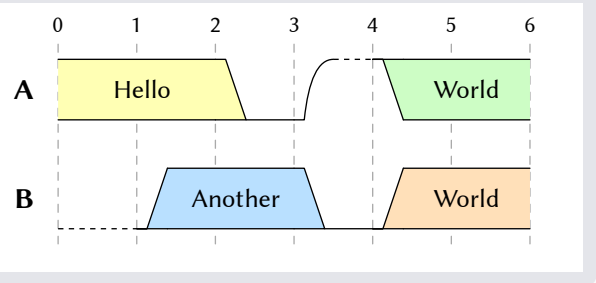
Parameters

```
wave(
  data: dictionary,
  size: length,
  debug: boolean,
  tick-overshoot: ratio,
  wave-width: auto integer,
  symbol-width: float,
  wave-height: float,
  inset-1: ratio float,
  inset-2: ratio float,
  wave-gutter: float,
  name-gutter: ratio,
  s-spacing: float,
  s-outside: float,
  mark-scale: float,
  show-guides: boolean,
  stroke: dictionary stroke,
  stroke-dashed: dictionary stroke,
  show-tick-lines: boolean,
  ticks-format: function,
  name-format: function,
  data-format: function,
  stroke-tick-lines: dictionary stroke,
  stroke-guides: dictionary stroke,
  debug-offset: float,
  bus-colors: dictionary
) → content
```

data `dictionary`

The wave data à la WaveDrom syntax.

```
#let data = (signal: (
  (wave: "3.0u7.", name: "A", data: "Hello
  World"),
  (wave: "d5.04.", name: "B", data:
  ([Another],[World],)),
))

#wave(data)
```



size `length`

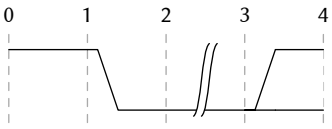
Sets the size of one «unit» of the diagram → it's just the *length* parameter of *cetz.canvas()*.

To calculate the actual sizes, use following:

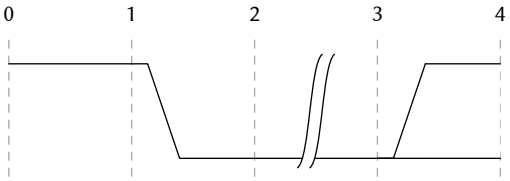
- Actual symbol width `size` · `symbol-width`
- Actual wave height `size` · `wave-height`

```
#let data = (signal: ((wave: "10|2"),))

*size: 8mm* (default)
#wave(data)

*size: 1.25cm*
#wave(data, size: 1.25cm)
```

size: 8mm (default)



size: 1.25cm



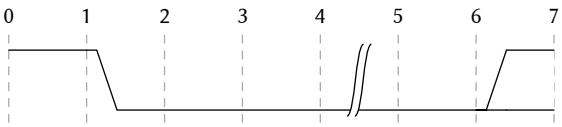
Default: `8mm`

debug `boolean`

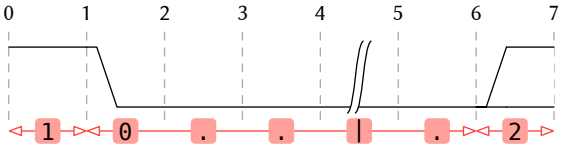
If enabled, shows which characters are assigned to each tick slot. Helps a lot with figuring out which symbol is inserted where.

```
#let data = (signal: ((wave:
"10..|.2"),))

*Debug OFF*
#wave(data, debug: false)

*Debug ON*
#wave(data, debug: true)
```

Debug OFF



Debug ON



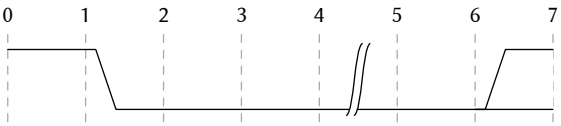
Default: `false`

tick-overshoot `ratio`

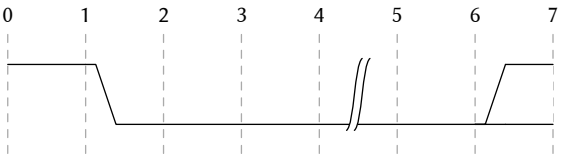
When tick lines are enabled, defines the amount of overshoot times `size` added to the top and bottom of the tick line.

```
#let data = (signal: ((wave:
"10..|.2"),))

*tick-overshoot: 25%* (default)
#wave(data)

*tick-overshoot: 50%*
#wave(data, tick-overshoot: 50%)
```

tick-overshoot: 25% (default)



tick-overshoot: 50%



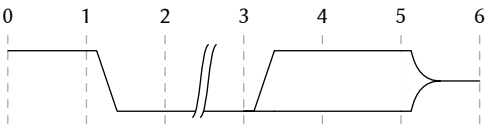
Default: `25%`

wave-width `auto` or `integer`

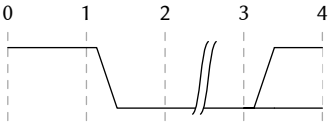
The width of the timing diagram in tick counts. If `auto` is given, uses the largest wave as tick count. If set to an integer, symbols after the given value are not rendered.

```
#let data = (signal: ((wave: "10|
2.z"),))

*wave-width: auto* (default)
#wave(data)

*wave-width: 4*
#wave(data, wave-width: 4)
```

wave-width: auto (default)



wave-width: 4



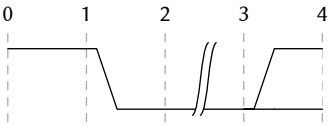
Default: `auto`

symbol-width `float`

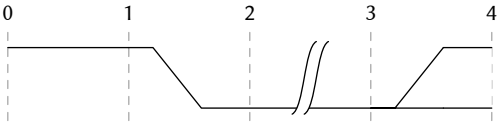
The width of one symbol. The actual symbol width equals to `size` · `symbol-width`

```
#let data = (signal: ((wave: "10|2"),))

*symbol-width: 1.3* (default)
#wave(data)

*symbol-width: 2*
#wave(data, symbol-width: 2)
```

symbol-width: 1.3 (default)



symbol-width: 2



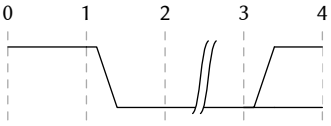
Default: `1.3`

wave-height `float`

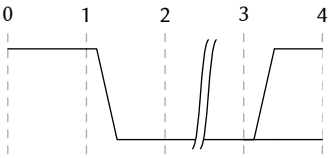
The height of one wave. The actual wave height equals to `size` · `wave-height`

```
#let data = (signal: ((wave: "10|2"),))

*wave-height: 1* (default)
#wave(data)

*wave-height: 1.5*
#wave(data, wave-height: 1.5)
```

wave-height: 1 (default)



wave-height: 1.5



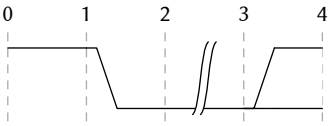
Default: `1`

inset-1 `ratio` or `float`

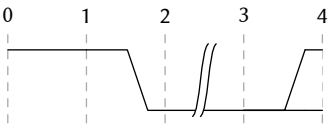
Ratio of the first inset and usually applied as a transition delay.

```
#let data = (signal: ((wave: "10|2"),))

*inset-1: 10%* (default)
#wave(data)

*inset-1: 40%*
#wave(data, inset-1: 40%)
```

inset-1: 10% (default)



inset-1: 40%



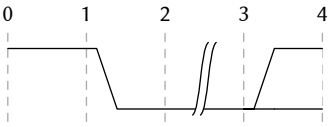
Default: `10%`

inset-2 `ratio` or `float`

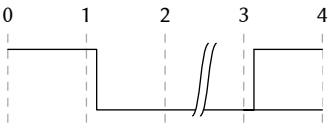
Ratio of the second inset and usually applied to transition.

```
#let data = (signal: ((wave: "10|2"),))

*inset-2: 20%* (default)
#wave(data)

*inset-2: 40%*
#wave(data, inset-2: 0%)
```

inset-2: 20% (default)



inset-2: 40%



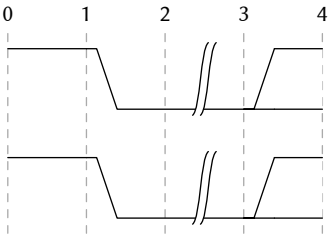
Default: `20%`

wave-gutter `float`

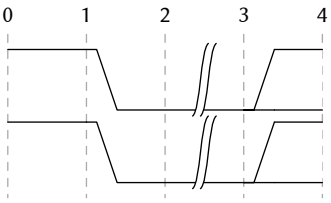
The spacing between waves. Not included when an empty wave is inserted (→ `(:)`).

```
#let data = (signal: ((wave: "10|2"),
(wave: "10|2"),))

*wave-gutter: 0.8* (default)
#wave(data)

*wave-gutter: 0.2*
#wave(data, wave-gutter: 0.2)
```

wave-gutter: 0.8 (default)



wave-gutter: 0.2



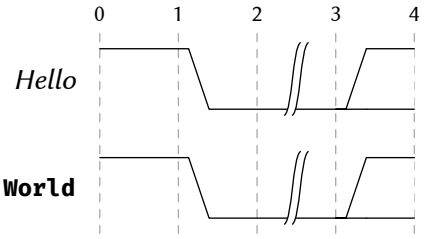
Default: `0.8`

name-gutter `ratio`

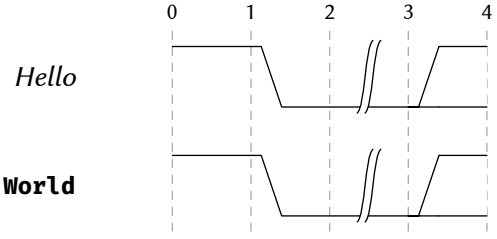
The spacing between wave and the wave name.

```
#let data = (signal: (
   (wave: "10|2", name: emph[Hello]),
   (wave: "10|2", name: raw("World")),
 ))

*name-gutter: 0.4* (default)
#wave(data)

*name-gutter: 1.6*
#wave(data, name-gutter: 1.6)
```

name-gutter: 0.4 (default)



name-gutter: 1.6



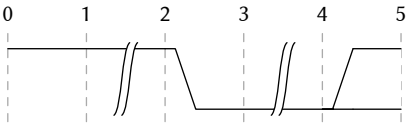
Default: `0.4`

s-spacing `float`

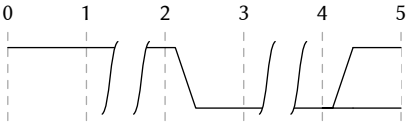
Spacing of the time skip symbol ($\int\int$).

```
#let data = (signal: (
   (wave: "1|0|2"),
 ))

*s-spacing: 0.1* (default)
#wave(data)

*s-spacing: 0.4*
#wave(data, s-spacing: 0.4)
```

s-spacing: 0.1 (default)



s-spacing: 0.4



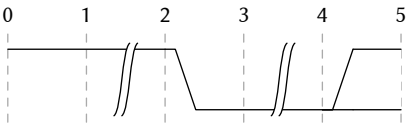
Default: `0.1`

s-outside `float`

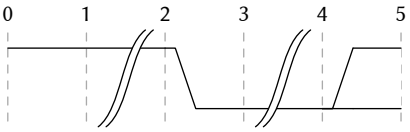
How much the time skip symbol reaches outside the wave.

```
#let data = (signal: (
   (wave: "1|0|2"),
 ))

*s-outside: 0.1* (default)
#wave(data)

*s-outside: 0.3*
#wave(data, s-outside: 0.3)
```

s-outside: 0.1 (default)



s-outside: 0.3



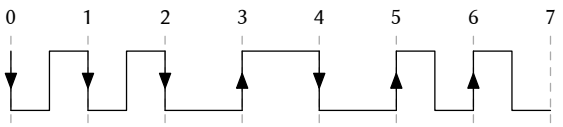
Default: `0.1`

mark-scale `float`

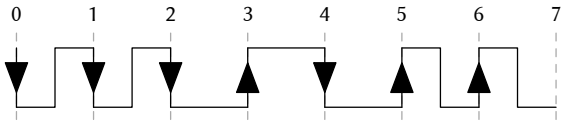
The size of the arrow marks applied to clocks, .

```
#let data = (signal: (
   (wave: "NNLHLPP"),
 ))

*mark-scale: 1* (default)
#wave(data)

*mark-scale: 2*
#wave(data, mark-scale: 2)
```

mark-scale: 1 (default)



mark-scale: 2



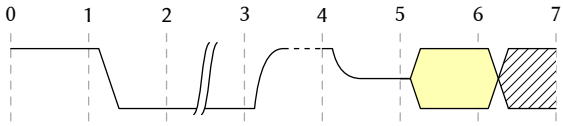
Default: 1

show-guides `boolean`

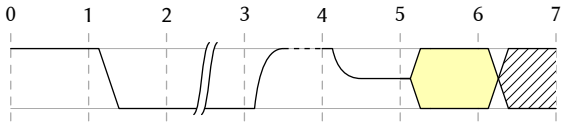
Renders two lines the «0» level and «1» level of the wave and uses `stroke-guides` for the stroke styling.

```
#let data = (signal: (
   (wave: "10|uz3x"),
 ))

*show-guides: false* (default)
#wave(data)

*show-guides: true*
#wave(data, show-guides: true)
```

show-guides: false (default)



show-guides: true



Default: false

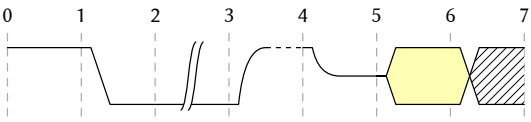
stroke `dictionary` or `stroke`

Changes/sets the styling of the wave strokes.

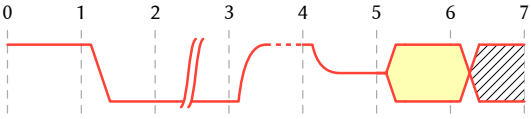
Note it is recommended to set at least `(cap: "round")`, so the line segments of the wave are correctly connect. It is a WIP fix!

```
#let data = (signal: (
   (wave: "10|uz3x"),
 ))

*Default*
#wave(data)

#raw("(paint: red, join: \"round\", cap:
\"round\")", lang: "typc")
#wave(data, stroke: (paint: red, join:
"round", cap: "round"))
```

Default



`(paint: red, join: "round", cap: "round")`



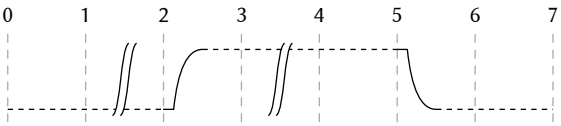
Default: `(thickness: 0.5pt, paint: black, join: "round", cap: "round")`

stroke-dashed `dictionary` or `stroke`

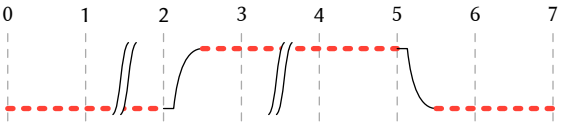
An extension of `stroke` for the transition edges `"u"` and `"d"`.

```
#let data = (signal: (
   (wave: "d|u|.d."),
 ))

*Default*
#wave(data)

#raw("(paint: red, dash: \"loosely-
dotted\",
 thickness: 2pt)", lang: "typc")
#wave(data, stroke-dashed: (paint: red,
dash: "loosely-dotted", thickness: 2pt))
```

Default



`(paint: red, dash: "loosely-dotted",
 thickness: 2pt)`



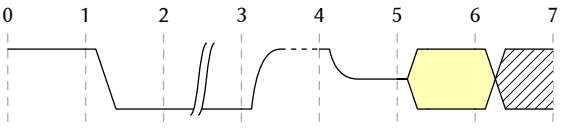
Default: `(cap: "butt", join: "round", dash: (2pt, 2pt))`

show-tick-lines `boolean`

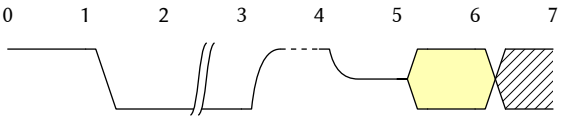
Renders the tick lines uses `stroke-tick-lines` for the stroke styling.

```
#let data = (signal: (
   (wave: "10|uz3x"),
 ))

*show-tick-lines: true* (default)
#wave(data)

*show-tick-lines: false*
#wave(data, show-tick-lines: false)
```

show-tick-lines: true (default)



show-tick-lines: false



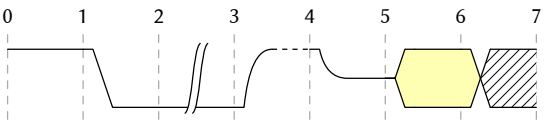
Default: `true`

ticks-format `function`

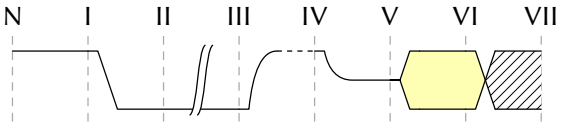
A rendering function with one parameter representing the tick number. Must return `content` and essentially describes how the label numbers are rendered. Setting it to `none` removes the labels.

```
#let data = (signal: (
   (wave: "10|uz3x"),
 ))

*Default*
#wave(data)

*ticks-format:* #raw("(n) =>
numbering(\"I\", n)", lang: "typc")
#wave(data, ticks-format: (n) =>
numbering("I", n))

*ticks-format:* #raw("none", lang: "typc")
#wave(data, ticks-format: none)
```

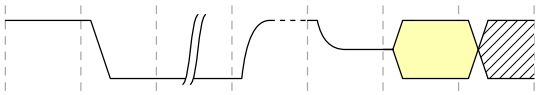
Default



ticks-format: `(n) => numbering("I", n)`



ticks-format: `none`



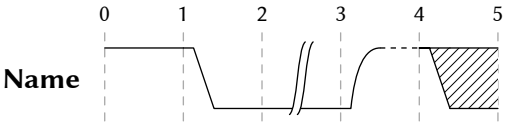
Default: `(n) => text(0.8em, numbering("1", n))`

name-format `function`

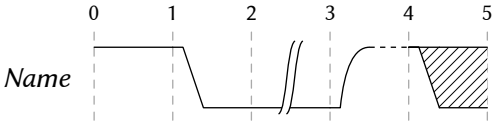
A rendering function with the wave name as parameter. Set it to `none` to not render any label (and removes `name-gutter`)

```
#let data = (signal: (
   (wave: "10|ux", name: [Name]),
 ))

*Default*
#wave(data)

*name-format:* #raw("(name) => emph(name)",
lang: "typc")
#wave(data, name-format: (name) =>
emph(name))

*name-format:* #raw("none", lang: "typc")
#wave(data, name-format: none)
```

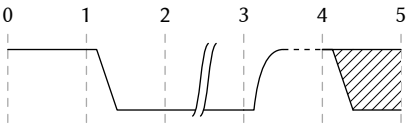
Default



name-format: `(name) => emph(name)`



name-format: `none`



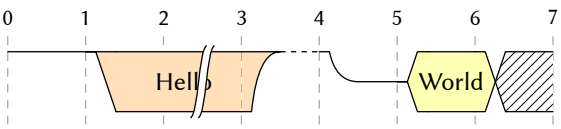
Default: `(name) => text(1em, weight: "bold", top-edge: "cap-height", bottom-edge: "baseline", if type(name) != str [#name] else [#eval(name, mode: "markup")])`

data-format `function`

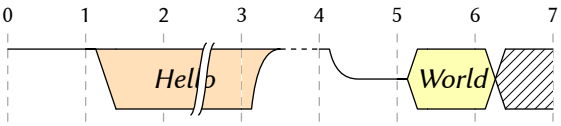
A rendering function with the respective data name as parameter. Set it to none to not render.

```
#let data = (signal: (
   (wave: "14|uz3x", data: "Hello World"),
 ))

*Default*
#wave(data)

*data-format:* #raw("(data) ⇒ emph(data)",
lang: "typc")
#wave(data, data-format: (data) ⇒
emph(data))

*data-format:* #raw("none", lang: "typc")
#wave(data, data-format: none)
```

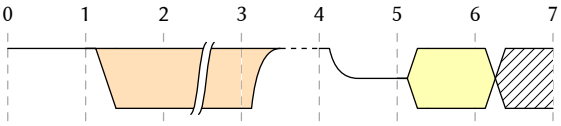
Default



data-format: `(data) ⇒ emph(data)`



data-format: none



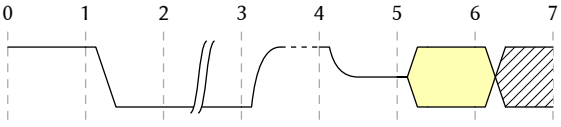
Default: `(data) ⇒ text(`
```
    0.9em, top-edge: "cap-height", bottom-edge: "baseline",
    if type(data) ≠ str [#data] else [#eval(data, mode: "markup")],
  )
```

stroke-tick-lines `dictionary` or `stroke`

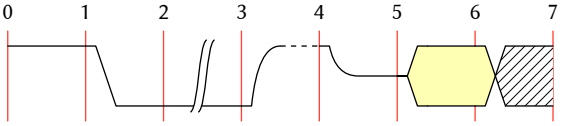
The stroke styling of the tick lines.

```
#let data = (signal: (
   (wave: "10|uz3x"),
 ))

*Default*
#wave(data)

*stroke-tick-lines: red*
#wave(data, stroke-tick-lines: red)
```

Default



stroke-tick-lines: red



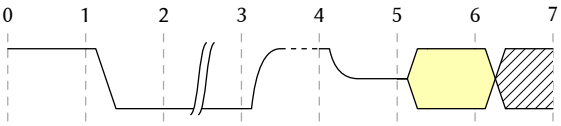
Default: `(cap: "round", thickness: 0.5pt, paint: gray, dash: "dashed")`

stroke-guides `dictionary` or `stroke`

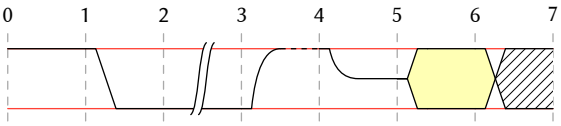
The stroke styling of the guide lines.

```
#let data = (signal: (
   (wave: "10|uz3x"),
 ))

*Default*
#wave(data)

*stroke-guides: red*
#wave(data, stroke-guides: red, show-
guides: true)
```

Default



stroke-guides: red



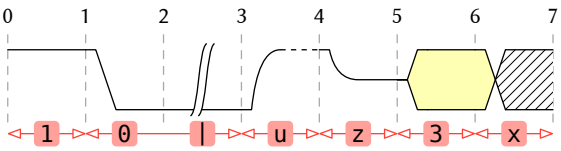
Default: `(cap: "round", thickness: 0.25pt, paint: gray)`

debug-offset `float`

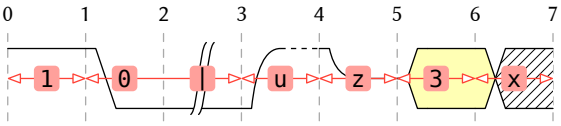
In case the debuging symbols are a bit off or hard to read, the debug-offset can be used to move the debug information vertically.

```
#let data = (signal: (
   (wave: "10|uz3x"),
 ))

*Default*
#wave(data, debug: true)

*debug-offset: -0.5*
#wave(data, debug-offset: -0.5, debug:
true)
```

Default



debug-offset: -0.5



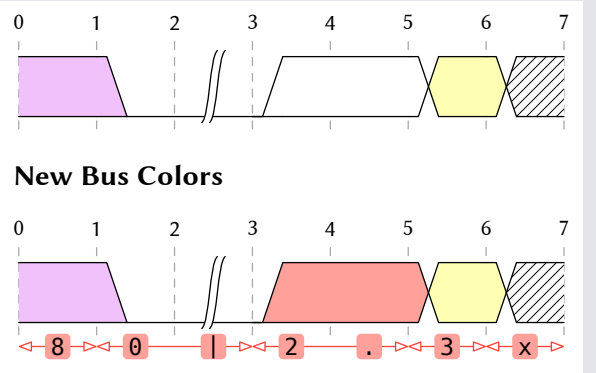
Default: `0.4`

bus-colors `dictionary`

A dictionary containing the bus number/type as the key and fill colors, tilings or gradients as value.

Note When not all bus colors are configured, the remaining ones **will** be set to the default values.

```
#let data = (signal: (
  (wave: "80|2.3x"),)
)
#let bus-colors = ("2": red.lighten(50%))

#wave(data)

*New Bus Colors*
#wave(data, bus-colors: bus-colors, debug: true)
```



__digidraw-x-pattern is the gray diagonal lines (as seen in the example below).

Default: (

```
    "2": white,
    "3": rgb("#ffffb4"),
    "4": rgb("#ffe0b9"),
    "5": rgb("#b9e0ff"),
    "6": rgb("#ccfdfe"),
    "7": rgb("#cdfdc5"),
    "8": rgb("#f0c1fb"),
    "9": rgb("#f8d0ce"),
    "x": __digidraw-x-pattern,
  )
```