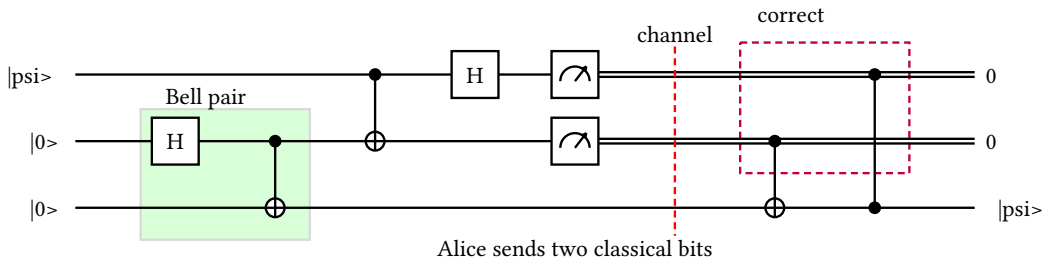
The qrab Manual

A readable language for quantum circuit diagrams,
and the compiler that turns it into LaTeX, Typst, and SVG



Version 0.1.1

github.com/inmzhang/qrab · playground

Contents

1 Introduction 3
1.1 What qrab is 3
1.2 Design philosophy 3
1.3 The three backends 3
2 Installation 4
3 The command line 5
3.1 Checking 5
3.2 Compiling 5
3.3 From source to PDF 5
3.4 Diagnostics 5
4 A guided tour 6
4.1 Wires and a gate 6
4.2 A controlled gate 6
4.3 Labels and measurement 6
5 Wires 8
5.1 Kinds 8
5.2 Labels 8
5.3 Arrays and ranges 8
5.4 Declaring wires implicitly 9
6 Operations 10
6.1 Built-in gates 10
6.2 Controls 10
6.3 Named boxes 11
6.4 Custom target operators 11
6.5 Phase gates 12
6.6 Measurement 12
6.7 Swap and barrier 13
7 Layout and scheduling 14
7.1 How columns are chosen 14
7.2 `parallel` 14
7.3 `overlay` 15
7.4 `touch` 15
7.5 `space` 16
7.6 Geometry 16
7.7 Vertical circuits 17
8 Wire lifecycle 19
8.1 Starting and ending 19
8.2 Changing kind 19
8.3 Bundles 19
8.4 Permutations 20
9 Annotations 21
9.1 `label` and `labels` 21
9.2 Braces 21
9.3 `equals` 22
9.4 Notes 22
9.5 Cuts 22
9.6 Marks and groups 23
10 Programming the source 24
10.1 `let` 24
10.2 `style` 24
10.3 Functions 24

10.4 `repeat` and `reverse` 25
10.5 Imports 26
11 Styling 28
11.1 Fields 28
11.2 Colours 28
11.3 Shapes 30
11.4 Where each field applies 30
11.5 Links 30
12 Backend escapes 32
13 How the picture is built 33
13.1 Coordinates 33
13.2 Growing, not clipping 33
13.3 Vertical circuits 33
14 Worked examples 34
14.1 Quantum Fourier transform 34
14.2 A decomposition 34
14.3 Teleportation, annotated 35
14.4 A sketch with omitted registers 36
14.5 Syndrome extraction 37
15 Library API 39
16 Relationship to qpic 40
17 Appendix: statement index 41

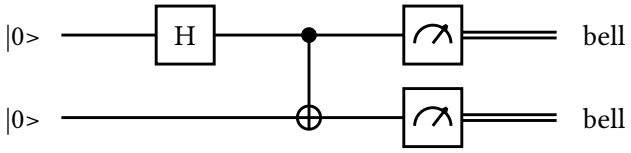
1 Introduction

1.1 What qrab is

`qrab` compiles a small language for describing quantum circuits into three finished formats: standalone LaTeX/TikZ, standalone Typst using Quill, and SVG. One source file produces all three, and the three agree because they are rendered from a single checked model rather than from the text you wrote.

The language is a descendant of qpic and draws the same pictures, but a circuit is written as statements rather than as positional commands, and the compiler checks what you wrote before it draws anything.

```
circuit bell_pair {
  qubit q[2]: "|0>" -> "bell"

  h q[0]
  x q[1] if q[0]
  measure q[0], q[1]
}
```



Everything in that snippet is ordinary language: `qubit` declares two wires and gives them an input and an output label, `h` and `x` are gates, `if` makes a gate controlled, and `measure` switches a wire to a classical double rule.

1.2 Design philosophy

A circuit is a program, not a picture. Positional macro languages make the easy diagram quick and the interesting diagram unmaintainable. `qrab` gives you the constructs you already reach for — naming a thing, calling it twice, importing it from another file — and checks that you used them correctly.

Say it once, get every format. The parser produces one semantic model, and each backend renders that model without ever seeing the source text.

Portable styling or none at all. Style options exist only where every backend can honour them. Anything genuinely backend-specific goes in a `backend` block, where it is visibly quarantined rather than silently ignored somewhere else.

Errors are part of the language. Every diagnostic knows where it came from, across `import` boundaries, and says what to do about it.

1.3 The three backends

Target	What it produces	Needs
`latex`	A standalone `.tex` document using TikZ, with absolute coordinates in centimetres. Alias: `tikz`.	A TeX engine
`typst`	A standalone `.typ` document using Quill's grid, so the circuit participates in Typst's own layout. Alias: `quill`.	Typst 0.15.1
`svg`	Finished SVG. No external toolchain at all, which is what the browser playground runs on.	nothing

The first two emit *source*, not pictures: hand the file to its own compiler. Quill lays out its own grid, so the Typst output is the same circuit drawn by a different engine rather than a copy of the other two; Section 13 has the details.

2 Installation

The distribution channels below are configured but stay pending until the first public release.

Channel	Command
Cargo	`cargo install qrab`
cargo-binstall	`cargo binstall qrab`
npm / Bun	`npm i -g qrab` · `npx qrab` · `bunx qrab`
Homebrew	`brew install --formula .../qrab.rb`
Shell	`curl -LsSf .../qrab-installer.sh | sh`
PowerShell	`irm .../qrab-installer.ps1 | iex`

Until then, install from a checkout:

```
cargo install --path . --locked
```

Nothing else is required to check a circuit, to generate LaTeX or Typst source, or to render SVG. A TeX engine and Typst are needed only to turn the generated sources into PDFs.

> **No installation at all**
> The playground runs the compiler as WebAssembly in your browser. Nothing is uploaded and nothing is installed; it is the fastest way to try anything in this manual.

3 The command line

`qrab` has two subcommands.

3.1 Checking

```
qrab check circuit.qrab
```

`check` parses the file, resolves its imports, runs every semantic check, and prints a one-line summary. It renders nothing, so it is the fastest way to find out whether a circuit is well formed:

```
bell_pair: 2 wire(s), 6 operation(s)
```

3.2 Compiling

```
qrab compile circuit.qrab                   # circuit.tex, circuit.typ, circuit.svg
qrab compile circuit.qrab --target svg      # circuit.svg
qrab compile circuit.qrab -t latex -o out.tex
qrab compile circuit.qrab -t typst -o -     # write to stdout
```

`--target` (`-t`) takes `latex` (alias `tikz`), `typst` (alias `quill`), `svg`, or `all`, and defaults to `all`. `--output` (`-o`) names a single output file and therefore requires a single backend; without it, each output is written next to the input with the matching extension.

3.3 From source to PDF

```
qrab compile circuit.qrab -t latex -o build/circuit.tex
tectonic build/circuit.tex --outdir build

qrab compile circuit.qrab -t typst -o build/circuit.typ
typst compile build/circuit.typ build/circuit.pdf
```

Both generated documents are standalone and crop to the circuit, so they drop straight into a paper with `\includegraphics` or Typst's `image`. Typst downloads Quill 0.8.0 on its first build.

3.4 Diagnostics

Errors carry a line, a column, a labelled span, and, where there is something useful to say, a suggestion. A run reports every error it can recover from rather than stopping at the first:

```
Error:   × 2 errors found

Error:
  × unknown wire `q[3]`
   ╭─[circuit.qrab:4:5]
 3 │
 4 │   h q[3]
   ·     ─
 5 │   gate "U" on q[0] if q[0]
   ╰────
  help: declare it before use or enable `autowires`
```

The second error — `q[0]` used as both a target and a control on line 5 — is reported in the same run. Recovery happens at statement boundaries, so one mistake does not hide the rest of the file.

Spans survive `import`: an error inside an imported module is reported against that module's own file and line, not against the position the text ended up at after expansion.

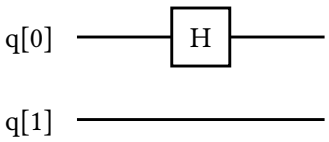
4 A guided tour

This chapter builds one circuit from nothing. Every intermediate step is a complete, compilable file.

4.1 Wires and a gate

A circuit is a named block. Inside it, declarations come first and operations follow in the order they happen.

```
circuit bell_pair {
  qubit q[2]

  h q[0]
}
```

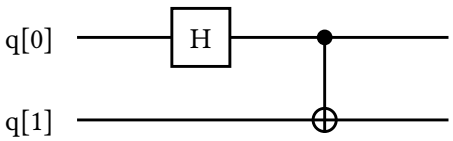


`qubit q[2]` declares a fixed-size array of two quantum wires, addressed `q[0]` and `q[1]`. `h q[0]` puts a Hadamard on the first of them. The second wire is drawn even though nothing acts on it, because it was declared.

4.2 A controlled gate

Any gate becomes controlled by naming its controls after `if`.

```
circuit bell_pair {
  qubit q[2]

  h q[0]
  x q[1] if q[0]
}
```

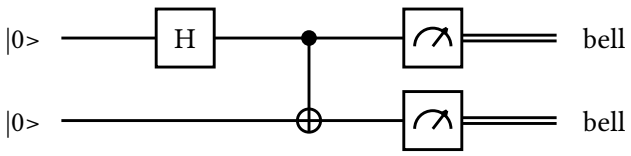


`x q[1] if q[0]` is the familiar CNOT: `qrab` draws a control dot on `q[0]` and the target notation on `q[1]`, because a controlled `X` has a conventional rendering. Everything else is drawn as a labelled box.

4.3 Labels and measurement

Wires can carry an input label, an output label, or both, and `measure` turns a quantum wire into a classical one.

```
circuit bell_pair {
  qubit q[2]: "|0>" -> "bell"

  h q[0]
  x q[1] if q[0]
  measure q[0], q[1]
}
```



That is the whole Bell-pair circuit. Compile it with `qrab compile bell.qrab` and you get `bell.tex`, `bell.typ`, and `bell.svg`, all describing this picture.

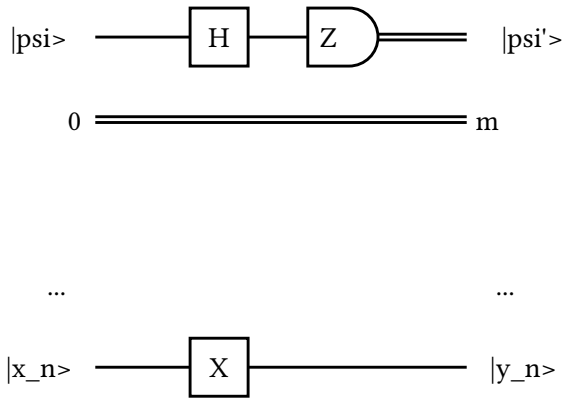
5 Wires

5.1 Kinds

Four declaration keywords cover every row a diagram can have.

Keyword	Draws
`qubit`	A single quantum line.
`bit`	A classical double line.
`hidden`	Nothing, until something makes the wire visible. Useful for a row that only exists for part of the diagram.
`ellipsis`	One wireless row labelled `...` at both ends, which makes an omitted register range explicit without inventing a wire name.

```
circuit wire_kinds {
  qubit data: "|psi>" -> "|psi'>"
  bit result: "0" -> "m"
  hidden scratch
  ellipsis omitted
  qubit last: "|x_n>" -> "|y_n>"

  h data
  measure data as "Z"
  x last
}
```



5.2 Labels

A declaration may carry an input label after `:` and an output label after `->`. Either may be omitted.

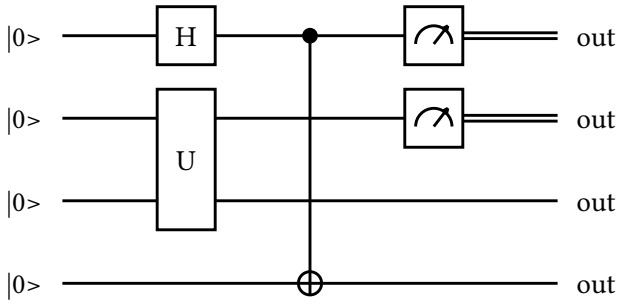
```
qubit message: "|psi>" -> "|psi>"
qubit work: "|0>"
qubit result -> "m"
qubit plain
```

Labels are plain text in every backend. They are not LaTeX maths: write `"|psi>"` rather than `"$\ket{\psi}$"`, and the same string appears in the SVG, the TikZ, and the Typst output. If you need real mathematics in a paper, reach for a `backend latex` block (Section 12).

5.3 Arrays and ranges

A declaration may be an array of fixed size, and a range selects several of its wires at once. Ranges are end-exclusive, so `q[1..4]` is `q[1]`, `q[2]`, and `q[3]`.

```
circuit wire_arrays {
  qubit q[4]: "|0>" -> "out"

  h q[0]
  gate "U" on q[1..3]
  x q[3] if q[0]
  measure q[0..2]
}
```



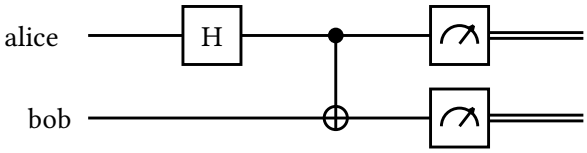
Ranges work anywhere a wire list is accepted: targets, controls, annotation spans, and function arguments. A statement that takes exactly one wire, such as `h`, rejects a range rather than silently picking the first.

An array declaration shares one input and output label across all its wires, which is why the picture above repeats `|0>` and `out` on every row.

5.4 Declaring wires implicitly

Strict declaration is the default: an undeclared name is an error, which is what catches a typo. For a quick sketch, `autowires` opts the rest of the circuit into creating an unknown quantum wire on first use, labelled with its own name.

```
circuit sketch {
  autowires

  h alice
  x bob if alice
  measure alice, bob
}
```



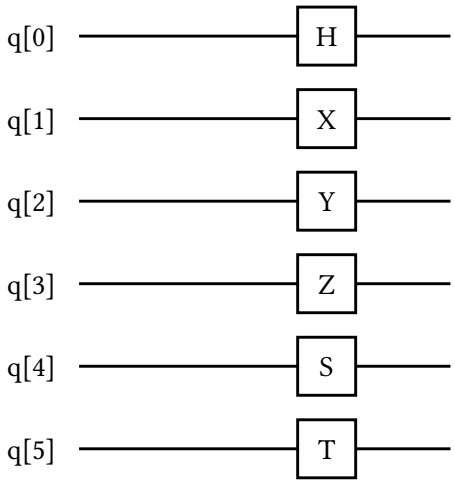
Use it for a sketch and turn it off for anything you intend to keep; with `autowires` on, a misspelled wire quietly becomes a new one.

6 Operations

6.1 Built-in gates

`h`, `x`, `y`, `z`, `s`, and `t` take one wire each and draw a box with the uppercase letter in it.

```
circuit builtin_gates {
  qubit q[6]

  parallel {
    h q[0]
    x q[1]
    y q[2]
    z q[3]
    s q[4]
    t q[5]
  }
}
```

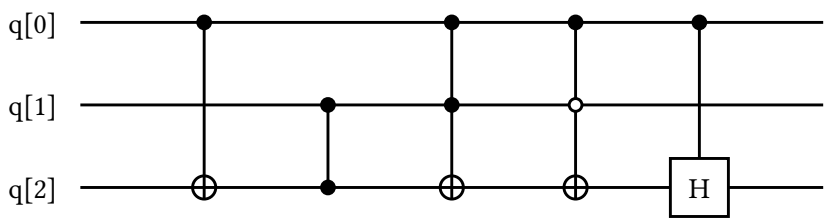


The `parallel` block in that example is explained in Section 7; without it, the six gates would pack into the earliest free column each and produce the same picture here anyway.

6.2 Controls

Controls follow `if`, separated by commas. A control prefixed with `!` is an open (negative) control and is drawn as a hollow dot.

```
circuit controls {
  qubit q[3]

  x q[2] if q[0]
  z q[2] if q[1]
  x q[2] if q[0], q[1]
  x q[2] if q[0], !q[1]
  h q[2] if q[0]
}
```

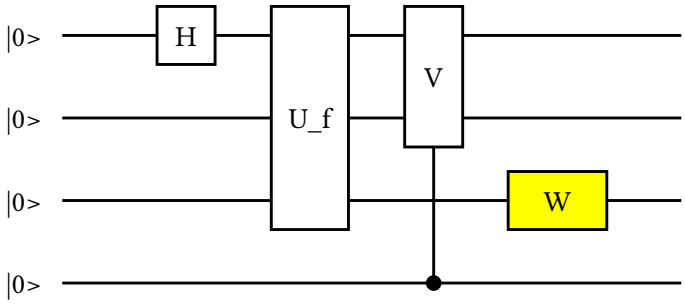


A controlled `X` with no styling and no link is drawn with the conventional target circle, and a controlled `Z` with the conventional dot; every other gate is drawn as a labelled box joined to its controls by a vertical line. A wire cannot be both a target and a control of the same operation.

6.3 Named boxes

`gate "label" on wires` draws an arbitrary box. Given several targets it spans them, and it may take controls like any other gate.

```
circuit boxes {
  qubit q[4]: "|0>"

  gate "H" on q[0]
  gate "U_f" on q[0..3]
  gate "V" on q[0], q[1] if q[3]
  gate "W" on q[2] with fill: yellow, width: 34
}
```

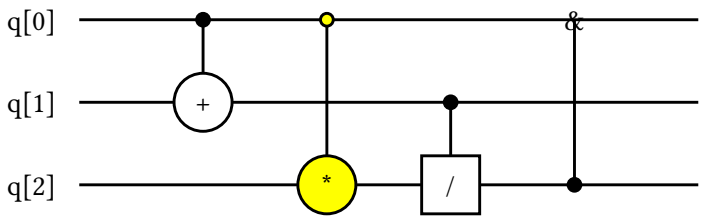


A box grows to fit its label rather than clipping it, and the column grows with it, so a long name spreads the diagram out instead of overlapping its neighbour. Set `width` to ask for more room than the label needs.

6.4 Custom target operators

The shape and the label are yours, so any operator notation is available without new syntax.

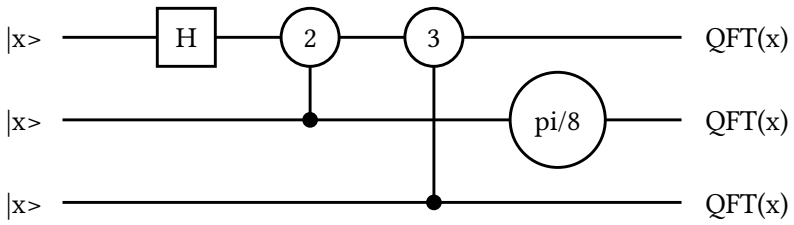
```
circuit custom_operators {
  qubit q[3]

  gate "+" on q[1] if q[0] with shape: circle
  gate "*" on q[2] if q[0] with shape: circle, fill: yellow
  gate "/" on q[2] if q[1] with shape: box
  gate "&" on q[0] if q[2] with shape: none
}
```



6.5 Phase gates

`phase` draws its label verbatim in a circle, exactly as qpic does.

```
circuit phases {
  qubit q[3]: "|x>" -> "QFT(x)"

  h q[0]
  phase "2" on q[0] if q[1]
  phase "3" on q[0] if q[2]
  phase "pi/8" on q[1] with size: 28
}
```

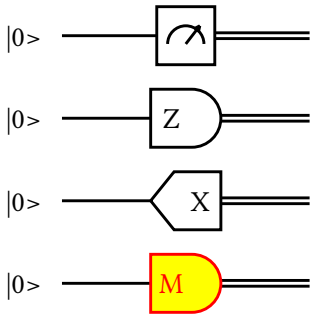


The conventional label is the index k of R_k, not the angle, which is why the QFT examples in this manual read `phase "2"` rather than `phase "pi/2"`. Unlike qpic the circle grows to fit, so a long label widens the gate instead of being cut off; see Section 13.2 for what that costs.

6.6 Measurement

An unlabelled `measure` draws a meter. `measure q as "Z"` draws a D-shaped result marker, and `using tag` switches it to a pointed tag. All three switch each target to a classical wire.

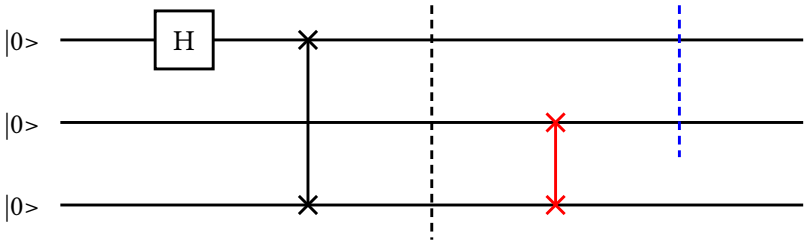
```
circuit measurements {
  qubit q[4]: "|0>"

  measure q[0]
  measure q[1] as "Z"
  measure q[2] as "X" using tag
  measure q[3] as "M" with fill: yellow, stroke: red
}
```



6.7 Swap and barrier

`swap` connects two wires with the crossed notation. `barrier` draws a dashed line across its selected wires; with no selection it covers every currently active wire.

```
circuit swaps_and_barriers {
  qubit q[3]: "|0>"

  h q[0]
  swap q[0], q[2]
  barrier
  swap q[1], q[2] with stroke: red, width: 10
  barrier q[0], q[1] with stroke: blue
}
```

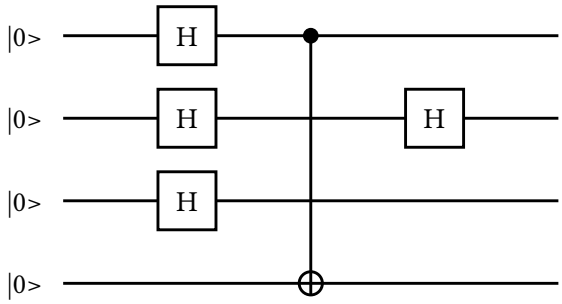


7 Layout and scheduling

7.1 How columns are chosen

You never write a column number. The compiler packs each operation into the earliest column that is free across every wire it occupies, while preserving source order on each wire.

```
circuit packing {
  qubit q[4]: "|0>"

  h q[0]
  h q[1]
  h q[2]
  x q[3] if q[0]
  h q[1]
}
```

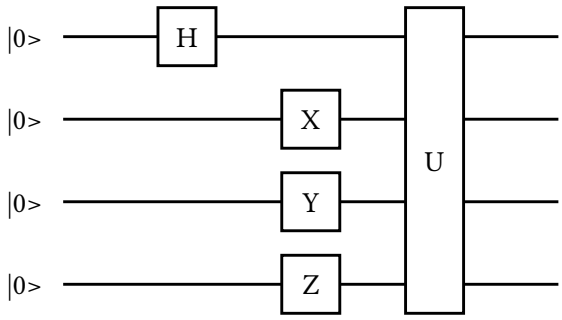


The three Hadamards are independent, so they share the first column. The CNOT occupies rows 0 through 3, so it has to wait for the column after the Hadamard on `q[0]`. The second Hadamard on `q[1]` waits for the CNOT, because the CNOT's span crosses `q[1]` even though it does not act on it.

7.2 `parallel`

A `parallel` block aligns its operations after all prior work and aligns everything after the block behind it, which is how you draw a layer.

```
circuit parallel_layer {
  qubit q[4]: "|0>"

  h q[0]
  parallel {
    x q[1]
    y q[2]
    z q[3]
  }
  gate "U" on q[0..4]
}
```

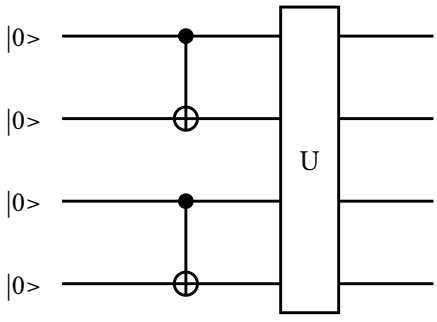


Operations whose visual spans overlap are still serialised inside the block, so `parallel` cannot produce an overlapping picture.

7.3 `overlay`

When two operations genuinely belong in one column even though their spans collide, say so explicitly.

```
circuit overlaid_column {
  qubit q[4]: "|0>"

  overlay {
    x q[1] if q[0]
    x q[3] if q[2]
  }
  gate "U" on q[0..4]
}
```

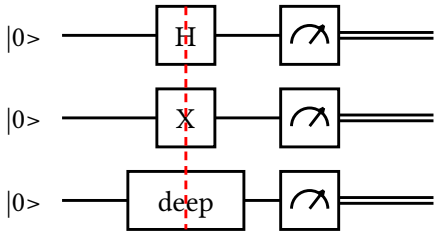


Two gates still cannot share a wire, and lifecycle changes and permutations are rejected inside an overlay.

7.4 `touch`

`touch` consumes no column of its own; it aligns everything after it with the deepest column reached so far. Give it a `stroke` to draw the alignment as a visible slice.

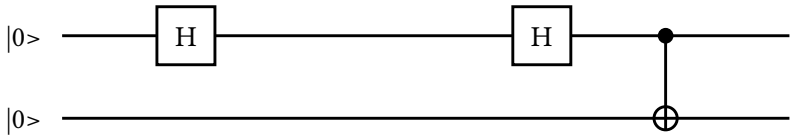
```
circuit touch_alignment {
  qubit q[3]: "|0>"

  h q[0]
  x q[1]
  gate "deep" on q[2] with width: 40
  touch with stroke: red, dash: dashed
  measure q[0..3]
}
```



7.5 `space`

`space` reserves invisible room. A `width` wider than the layout's `column_gap` widens that column and shifts everything after it; a narrower one changes nothing, because the abstract gap is a floor.

```
circuit reserved_space {
  qubit q[2]: "|0>"

  h q[0]
  space q[0] with width: 80
  h q[0]
  x q[1] if q[0]
}
```

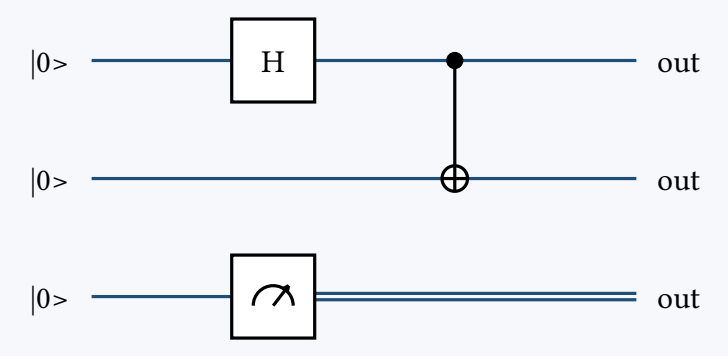


7.6 Geometry

`layout` sets the shared geometry every backend reads. Sizes are in points; gaps are abstract units that each backend maps to its own grid.

Property	Default	Meaning
`orientation`	`horizontal`	`horizontal` or `vertical`.
`scale`	`1`	Overall scale factor.
`column_gap`	`1.5`	Distance between columns.
`wire_gap`	`1`	Distance between wires.
`gate_size`	`20`	Default box size, in points.
`corner_radius`	`4`	Corner rounding of permutation curves, in points.
`comment_width`	`144`	Wrapping width of `note` text, in points.
`background`	`white`	Background colour of the whole diagram.

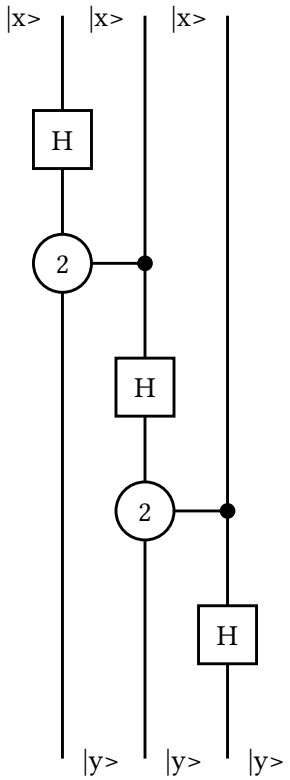
```
circuit tuned_layout {
  layout {
    scale: 1.1
    column_gap: 2.0
    wire_gap: 1.3
    gate_size: 26
    background: "#F7F8FC"
  }

  qubit q[3]: "|0>" -> "out" with stroke: "#1F4E79"

  h q[0]
  x q[1] if q[0]
  measure q[2]
}
```



7.7 Vertical circuits

A vertical circuit reads top to bottom, as qpic's does: the input labels sit above the first column, time runs down the page, and the wires run left to right in declaration order.

```
circuit vertical_reading {
  layout { orientation: vertical }

  qubit q[3]: "|x>" -> "|y>"

  h q[0]
  phase "2" on q[0] if q[1]
  h q[1]
  phase "2" on q[1] if q[2]
  h q[2]
}
```



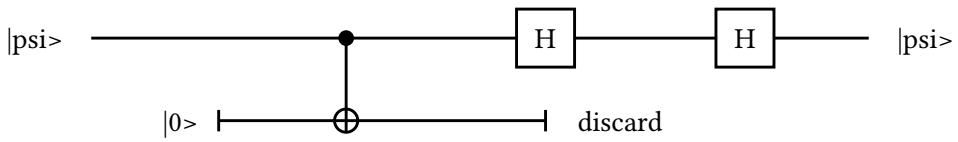
8 Wire lifecycle

Wires do not have to exist for the whole diagram, they do not have to keep the same kind, and they do not have to stay in one row.

8.1 Starting and ending

`start` defers a wire until the column it appears in; `end` stops it. Either may carry an `as "label"`, which is drawn at the tick.

```
circuit lifetimes {
  qubit main: "|psi>" -> "|psi>"
  qubit ancilla

  start ancilla as "|0>"
  x ancilla if main
  h main
  end ancilla as "discard"
  h main
}
```

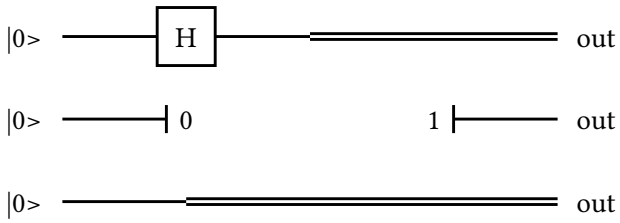


An omitted wire list on `start` selects every wire that has not started yet; on `end` it selects every active wire.

8.2 Changing kind

`set wires to quantum | classical | hidden` changes what the line draws from that column on. A transition to `hidden` or `quantum` may add `as "value"` to draw a known-value exit or entry marker.

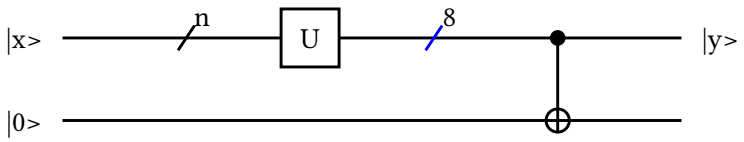
```
circuit wire_kinds_change {
  qubit q[3]: "|0>" -> "out"

  h q[0]
  set q[0] to classical
  set q[1] to hidden as "0"
  space q[1] with width: 30
  set q[1] to quantum as "1"
  set q[2] to classical
}
```



8.3 Bundles

`bundle` draws the multiplicity slash used for a register drawn as one line.

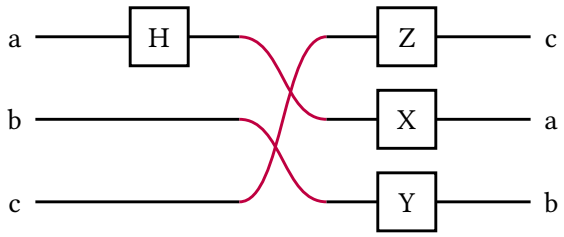
```
circuit bundles {
  qubit register: "|x>" -> "|y>"
  qubit single: "|0>"

  bundle "n" on register
  gate "U" on register
  bundle "8" on register with stroke: blue
  x single if register
}
```



8.4 Permutations

`permute` lists the selected wires in their new visual order. Every later operation, and every output label, follows the new order.

```
circuit reordering {
  qubit a: "a" -> "a"
  qubit b: "b" -> "b"
  qubit c: "c" -> "c"

  h a
  permute c, a, b with stroke: purple, width: 30
  x a
  y b
  z c
}
```



The `width` of a permutation controls how much horizontal room the crossing curves get, and `layout.corner_radius` controls how sharply they turn.

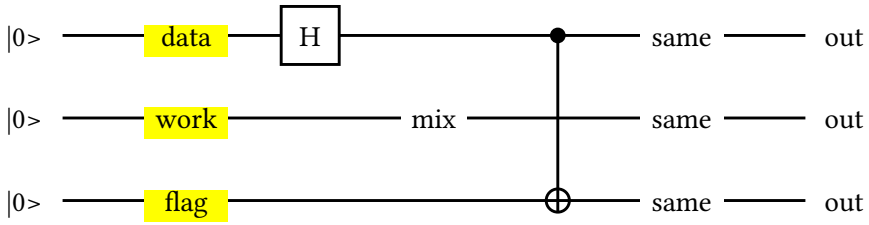
9 Annotations

Annotations describe the circuit without becoming part of it. Most of them share a column with the operation they annotate rather than consuming one.

9.1 `label` and `labels`

`label` centres one piece of text across a wire span; `labels` puts text on each wire of a selection, taking either one repeated label or one per wire.

```
circuit wire_labels {
  qubit q[3]: "|0>" -> "out"

  labels "data", "work", "flag" on q[0..3] with fill: yellow
  h q[0]
  label "mix" on q[0..3]
  x q[2] if q[0]
  labels "same" on q[0..3]
}
```

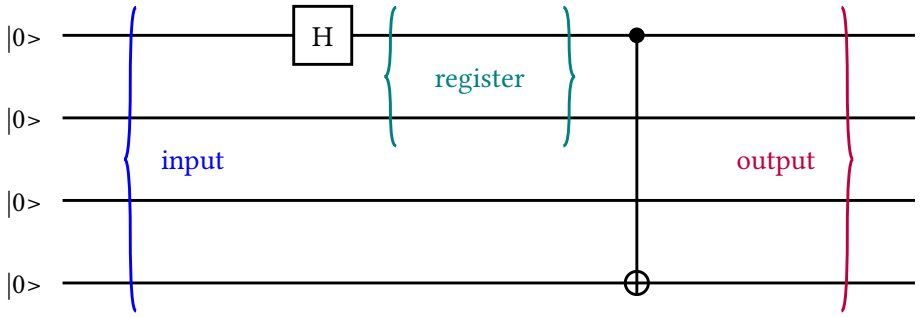


Label text is drawn on the diagram background, so the wire breaks around it rather than striking through it. That is what qpic does too. A `gate` with `shape: none` takes no fill, also as in qpic, so its wire does run behind the text.

9.2 Braces

`brace left | right | both` draws a curly brace beside or around a label spanning the selected wires.

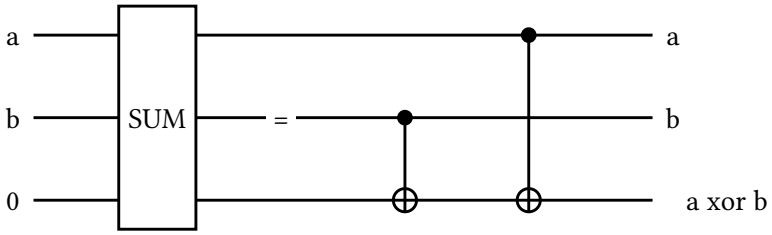
```
circuit braces {
  qubit q[4]: "|0>"

  brace left "input" on q[0..4] with stroke: blue
  h q[0]
  brace both "register" on q[0..2] with stroke: teal
  x q[3] if q[0]
  brace right "output" on q[0..4] with stroke: purple
}
```



9.3 `equals`

`equals` centres `=` across the wires, which is how a decomposition is written: the composite gate, then `=`, then what it expands to. It accepts a label of its own, an `on` selection, and `braced left | right | both`.

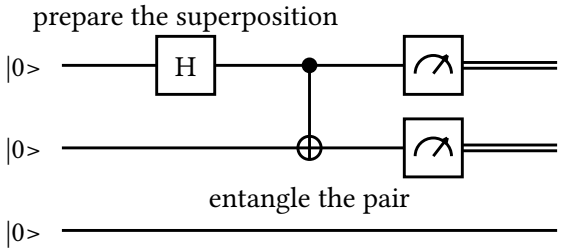
```
circuit decomposition {
  qubit a: "a" -> "a"
  qubit b: "b" -> "b"
  qubit sum: "0" -> "a xor b"

  gate "SUM" on a, b, sum
  equals
  x sum if b
  x sum if a
}
```



9.4 Notes

`note above | below` attaches free text to the preceding slice without consuming a column of its own. The text wraps at `layout.comment_width`.

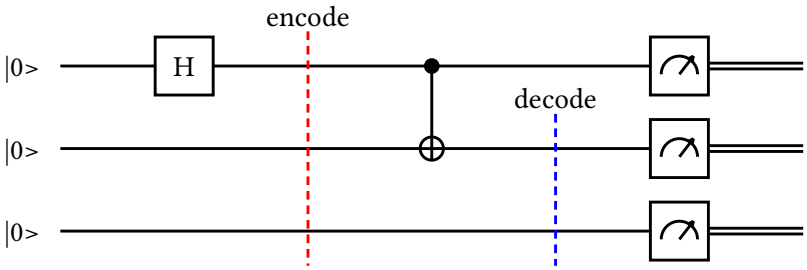
```
circuit notes {
  layout { comment_width: 90 }

  qubit q[3]: "|0>"

  h q[0]
  note above "prepare the superposition" on q[0]
  x q[1] if q[0]
  note below "entangle the pair" on q[1]
  measure q[0], q[1]
}
```



9.5 Cuts

A `cut` is a stage separator and does occupy its own column.

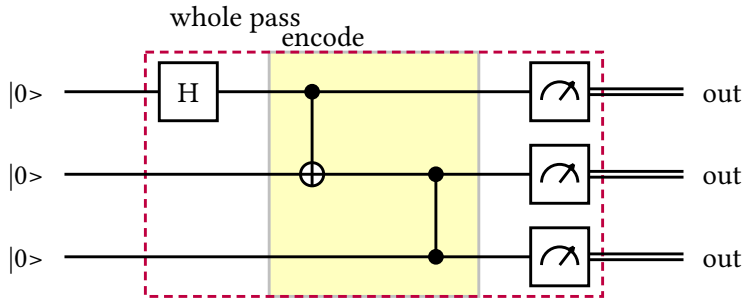
```
circuit stages {
  qubit q[3]: "|0>"

  h q[0]
  cut as "encode" with stroke: red
  x q[1] if q[0]
  cut on q[1..3] as "decode" with stroke: blue
  measure q[0..3]
}
```



9.6 Marks and groups

A `mark` is a named position in the operation stream; it is not an operation and draws nothing. A `group` highlights everything between two marks.

```
circuit regions {
  qubit q[3]: "|0>" -> "out"

  mark start
  h q[0]
  mark encode
  x q[1] if q[0]
  z q[2] if q[1]
  mark encoded

  group "encode" from encode to encoded on q[0..3] with fill: yellow, opacity: 0.25

  measure q[0..3]
  group "whole pass" from start to here on q[0..3] with stroke: purple, dash: dashed
}
```



The end mark is exclusive, and `here` means every operation parsed so far. Groups may overlap or nest, and are drawn behind the circuit.

10 Programming the source

Everything in this chapter happens before a picture exists. `let`, `style`, `fn`, and `import` are resolved into the same typed operation tree that a literal circuit produces — none of them performs textual substitution, so a parameter name that also appears inside a gate label stays ordinary label text.

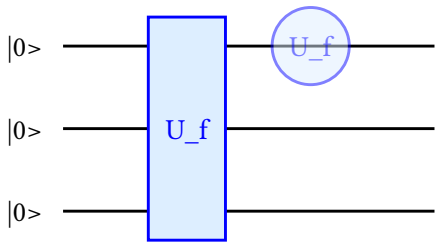
10.1 `let`

`let` names a label or a colour.

10.2 `style`

`style` names a set of checked style fields, which any operation can then apply with `with`. A use may add comma-separated overrides after the style name.

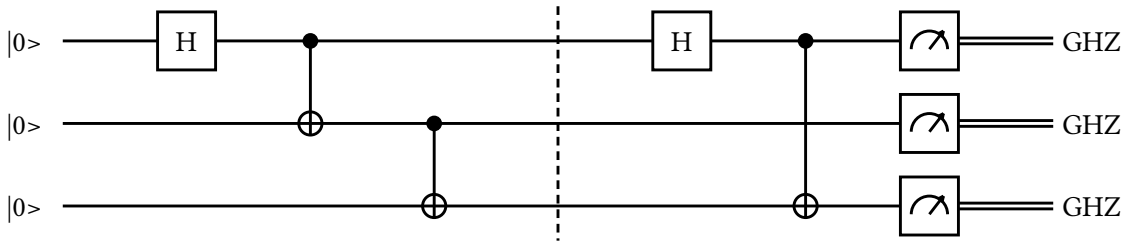
```
let oracle_name = "U_f"
let accent = "#DDEEFF"

style oracle {
  fill: accent
  stroke: blue
  shape: box
}

circuit named_values {
  qubit q[3]: "|0>"

  gate oracle_name on q[0..3] with oracle
  gate oracle_name on q[0] with oracle, opacity: 0.5, shape: circle
}
```



10.3 Functions

`fn` names a sequence of operations over wire parameters. Functions may call functions defined before them, arity is checked at the call site, and two parameters cannot be bound to the same wire.

```
fn entangle(control, target) {
  h control
  x target if control
}

fn ghz(a, b, c) {
  entangle(a, b)
  x c if b
}

circuit composed {
  qubit q[3]: "|0>" -> "GHZ"

  ghz(q[0], q[1], q[2])
  barrier
  entangle(q[0], q[2])
  measure q[0..3]
}
```

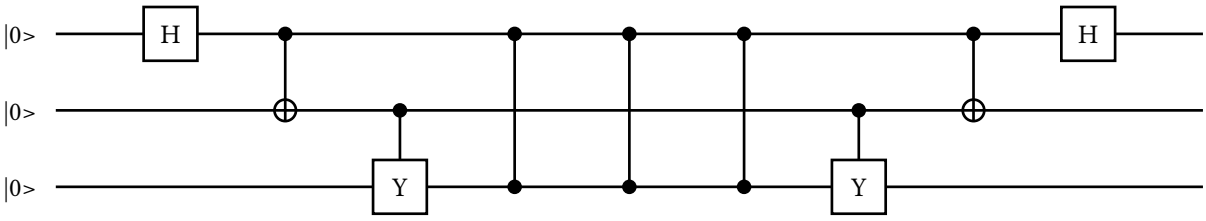


Declarations, captures, forward calls, and recursion are deliberately not part of the current function subset. A function is a reusable block of operations, not a general procedure.

10.4 `repeat` and `reverse`

`repeat count { ... }` repeats a parsed block, including function calls. `reverse { ... }` emits its operations in reverse source order, and `reverse from start_mark to end_mark` replays an earlier marked range backwards.

```
circuit structured {
  qubit q[3]: "|0>"

  mark encode
  h q[0]
  x q[1] if q[0]
  y q[2] if q[1]
  mark encoded

  repeat 3 {
    z q[2] if q[0]
  }

  reverse from encode to encoded
}
```



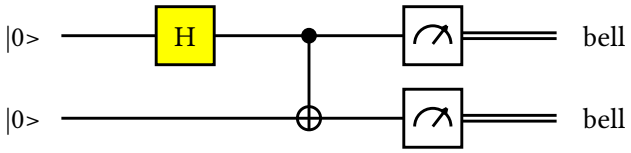
> **What `reverse` does and does not do**
> `reverse` reverses the order of statements. It does not invert the mathematical action of each gate: an `s` stays an `s`, it does not become its adjoint. For a self-inverse block — most encoders, most syndrome circuits — that is exactly the uncomputation you want, and for anything else you should spell out the adjoint gates.

10.5 Imports

A module holds declarations — `import`, `let`, `style`, and `fn` — and the root file owns the single `circuit`. Paths are relative to the importing file, duplicate imports load once, and cycles are rejected.

```
// modules/prep.qrab
style prepared {
  fill: yellow
}

fn make_bell(control, target) {
  h control with prepared
  x target if control
}
```

```
import "modules/prep.qrab"

circuit imported {
  qubit q[2]: "|0>" -> "bell"

  make_bell(q[0], q[1])
  measure q[0], q[1]
}
```



11 Styling

Every operation accepts a trailing `with` clause. The grammar is shared so a named `style` can be reused anywhere, but each visual consumes only the fields that mean something for it; the rest are accepted and ignored.

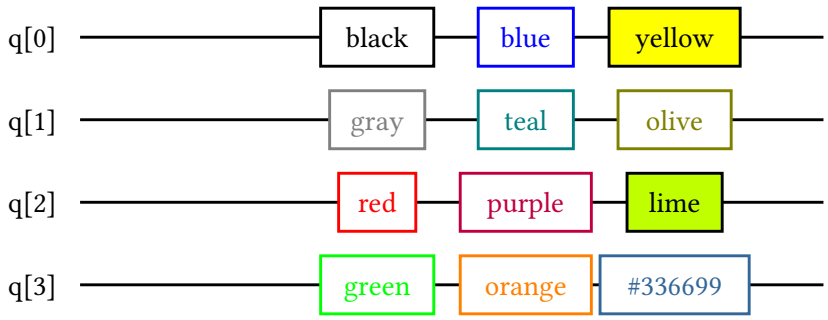
11.1 Fields

Field	Value	Meaning
`stroke`	colour	Outline and text colour.
`fill`	colour	Interior colour.
`width`	points	Extent along the time axis.
`height`	points	Extent across the wires.
`size`	points	Sets `width` and `height` at once.
`shape`	name	`box`, `circle`, `ellipse`, or `none`.
`dash`	name	`solid` or `dashed`.
`opacity`	0 to 1	Transparency of the shape.
`link`	URL	Wraps the visible text in a hyperlink.

11.2 Colours

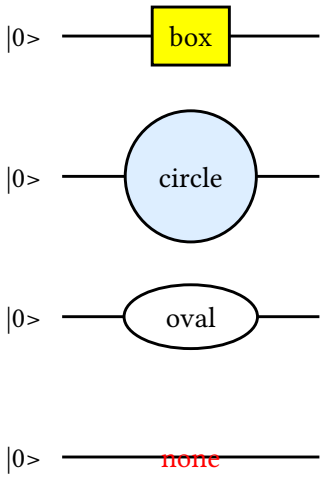
Twelve named colours are portable across all three backends: `black`, `white`, `gray`, `red`, `green`, `blue`, `teal`, `purple`, `orange`, `yellow`, `olive`, and `lime`. Anything else is written as a quoted six-digit hex value such as `"#336699"`. A name that is not on the list is an error rather than a silent fallback, because the point of the list is that every backend draws the same colour.

```
circuit palette {
  layout { column_gap: 1.8 }

  qubit q[4]

  parallel {
    gate "black" on q[0] with stroke: black
    gate "gray" on q[1] with stroke: gray
    gate "red" on q[2] with stroke: red
    gate "green" on q[3] with stroke: green
  }
  parallel {
    gate "blue" on q[0] with stroke: blue
    gate "teal" on q[1] with stroke: teal
    gate "purple" on q[2] with stroke: purple
    gate "orange" on q[3] with stroke: orange
  }
  parallel {
    gate "yellow" on q[0] with fill: yellow
    gate "olive" on q[1] with stroke: olive
    gate "lime" on q[2] with fill: lime
    gate "#336699" on q[3] with stroke: "#336699"
  }
}
```



11.3 Shapes

```
circuit shapes {
  layout { wire_gap: 1.7 }

  qubit q[4]: "|0>"

  gate "box" on q[0] with shape: box, fill: yellow
  gate "circle" on q[1] with shape: circle, fill: "#DDEEFF"
  gate "oval" on q[2] with shape: ellipse, width: 46, height: 22
  gate "none" on q[3] with shape: none, stroke: red
}
```



`shape: none` draws the label with no outline and no fill, which is qpic's behaviour for a shapeless operator, and which is why the wire is visible through the text.

11.4 Where each field applies

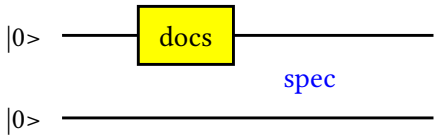
Visual	Honours
Gates, measurements, value markers	the full set
Wire declarations	`stroke`, `dash`, `opacity`
`space`	`width`, `height`
`barrier`, `cut`	`stroke`, `opacity` (always dashed)
`touch`	`stroke`, `dash`, `opacity`
`swap`, `permute`	`stroke`, `dash`, `opacity`, and an applicable `width`
Labels, endpoints	`stroke`, `fill`, `opacity`, `link`

In particular, `width` or `link` on a wire declaration, `fill` or `dash: solid` on a barrier or a cut, and `shape` on a label or a `touch` have no effect.

11.5 Links

A link must be an absolute `http`, `https`, or `mailto` URL, and is checked. It wraps the visible text of a gate or a label in every backend that can carry one.

```
circuit linked {
  qubit q[2]: "|0>"

  gate "docs" on q[0] with link: "https://example.com/gate", fill: yellow
  label "spec" on q[0..2] with link: "https://example.com/spec", stroke: blue
}
```



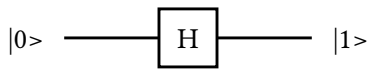
One backend-specific style
`bundle` styling is not fully portable: the LaTeX backend applies `stroke`, `dash`, and `opacity` to the slash, while Quill 0.8 exposes no style parameters for its `nwire`, so the Typst output uses Quill's default slash and label appearance.

12 Backend escapes

Backend-only code is deliberately isolated. A `backend` block names one target and carries verbatim strings that are emitted only for it.

```
circuit backend_hooks {
  backend latex {
    preamble: "\\newcommand{\\tag}{TikZ only}"
    before: "\\node[anchor=south west] at (0,0.4) {\\tag};"
  }

  backend typst {
    preamble: "#let tag = [Typst only]"
    before: "#align(center)[#tag]"
  }

  qubit q: "|0>" -> "|1>"
  h q
}
```

Each block may repeat `preamble`, `before`, or `after`. `preamble` lands in the document preamble, `before` just inside the picture, and `after` just before it closes. The SVG backend has no escape block and ignores both of these, so the picture below is what every backend draws with the hooks removed:



Escapes are the explicit replacement for qpic's preamble and TikZ hooks. Reach for one when you need real LaTeX mathematics in a label, a package only one document needs, or an effect the common model cannot express — and not otherwise, because anything you put here exists in exactly one output.

13 How the picture is built

13.1 Coordinates

The LaTeX and SVG backends place everything at absolute coordinates: wires are `wire_gap` apart, columns are `column_gap` apart, and the compiler computes both. The Typst backend instead hands the circuit to Quill, which lays out its own grid from the contents of each cell. This is why the Typst output is not pixel-identical to the other two: it is the same circuit, drawn by a different engine, and Quill's grid will size a column to its contents where the coordinate backends use the abstract gap.

Anything that needs more room than the column gap widens its own column and shifts everything after it — a `space`, a gate with an explicit `width`, a box whose label outgrew the default size, or a lifecycle label that hangs into the gap. The result is that a long label spreads a diagram out rather than overlapping its neighbour.

13.2 Growing, not clipping

Shapes grow to fit their labels. qpic clips instead, which keeps the layout exact and loses the end of the label; `qrab` keeps the label and moves the layout.

Along the time axis this is free, because the column grows too. Across the wires there is nowhere to grow into: a circle or an ellipse whose label is wider than `layout.wire_gap` will reach into the row above and below. When that happens, give the gate an explicit `size` or widen `wire_gap`.

13.3 Vertical circuits

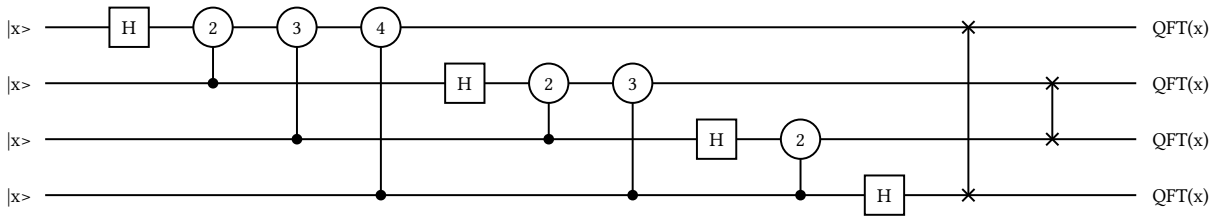
A vertical circuit is laid out horizontally and then turned a quarter turn, so that time runs down the page and the first-declared wire stays on the left. Text is turned back the other way and stays upright. Wire labels are drawn horizontally, so in a vertical circuit with long labels they can crowd each other; shorten them or widen `wire_gap`.

14 Worked examples

14.1 Quantum Fourier transform

Nothing here is new: `phase` with a control, and `swap` for the bit reversal at the end. The label of each phase gate is the index of R_k, as in qpic.

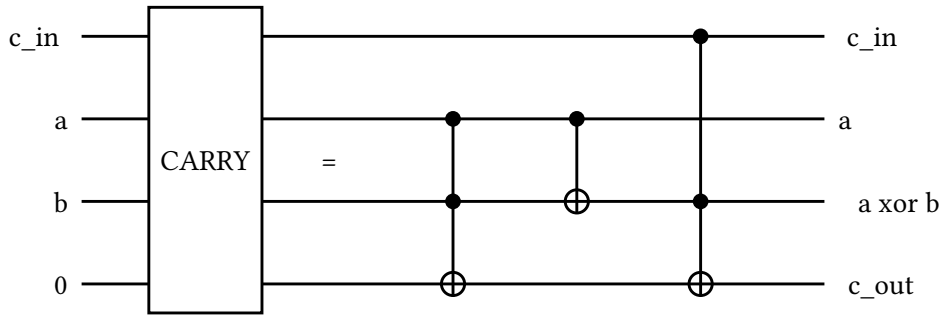
```
circuit qft4 {
  qubit q[4]: "|x>" -> "QFT(x)"

  h q[0]
  phase "2" on q[0] if q[1]
  phase "3" on q[0] if q[2]
  phase "4" on q[0] if q[3]
  h q[1]
  phase "2" on q[1] if q[2]
  phase "3" on q[1] if q[3]
  h q[2]
  phase "2" on q[2] if q[3]
  h q[3]
  swap q[0], q[3]
  swap q[1], q[2]
}
```



14.2 A decomposition

`equals` is what makes a decomposition figure read as one. The composite gate is drawn on the left, then `=`, then the circuit it expands to, all in a single diagram.

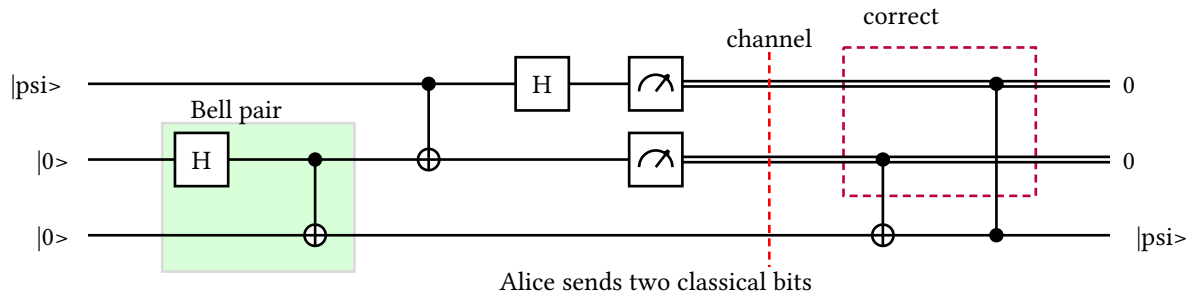
```
circuit vbe_carry {
  qubit cin: "c_in" -> "c_in"
  qubit a: "a" -> "a"
  qubit b: "b" -> "a xor b"
  qubit cout: "0" -> "c_out"

  gate "CARRY" on cin, a, b, cout
  equals
  x cout if a, b
  x b if a
  x cout if cin, b
}
```



14.3 Teleportation, annotated

Groups highlight a marked range, a `cut` separates the classical channel from the quantum part, and a `note` explains it. Notice that the groups are declared at the end, after the marks they refer to exist, but are drawn behind everything.

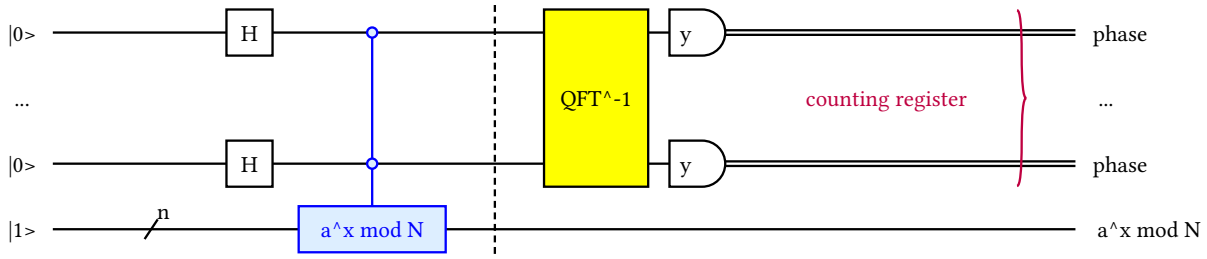
```
circuit annotated_teleportation {
  layout { comment_width: 130 }

  qubit message: "|psi>" -> "0"
  qubit alice: "|0>" -> "0"
  qubit bob: "|0>" -> "|psi>"

  mark entangle
  h alice
  x bob if alice
  mark entangled

  x alice if message
  h message
  measure message, alice
  note below "Alice sends two classical bits" on bob

  cut as "channel" with stroke: red
  mark correct
  x bob if alice
  z bob if message

  group "Bell pair" from entangle to entangled on alice, bob with fill: green,
opacity: 0.15
  group "correct" from correct to here on message, alice with stroke: purple, dash:
dashed
}
```



14.4 A sketch with omitted registers

`ellipsis` stands in for the rows a real circuit would have, `bundle` says how many wires each drawn line really carries, and `brace` picks out the register the algorithm reads.

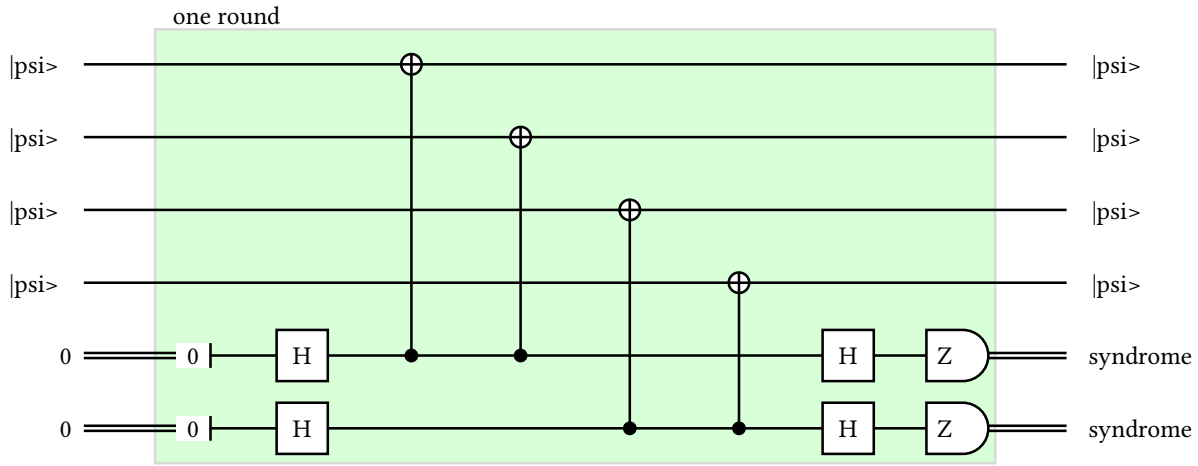
```
circuit shor_sketch {
  qubit first: "|0>" -> "phase"
  ellipsis omitted
  qubit last: "|0>" -> "phase"
  qubit work: "|1>" -> "a^x mod N"

  bundle "n" on work
  parallel {
    h first
    h last
  }
  gate "a^x mod N" on work if first, last with fill: "#DDEEFF", stroke: blue
  barrier
  gate "QFT^-1" on first, last with fill: yellow
  measure first, last as "y"
  brace right "counting register" on first, last with stroke: purple
}
```



14.5 Syndrome extraction

A `hidden`-to-`quantum` transition brings the ancillas in with a known value, `parallel` draws the layers, and a `group` marks one round.

```
circuit syndrome_extraction {
  qubit d[4]: "|psi>" -> "|psi>"
  bit s[2]: "0" -> "syndrome"

  mark round
  set s[0] to quantum as "0"
  set s[1] to quantum as "0"
  parallel {
    h s[0]
    h s[1]
  }
  x d[0] if s[0]
  x d[1] if s[0]
  x d[2] if s[1]
  x d[3] if s[1]
  parallel {
    h s[0]
    h s[1]
  }
  measure s[0], s[1] as "Z"
  mark measured

  group "one round" from round to measured on d[0..4], s[0..2] with fill: green,
opacity: 0.15
}
```



15 Library API

The crate exposes the same pipeline the binary uses.

```
use qrab::{Target, compile, load_source, parse, render};

// Shortest path, for source already in memory.
let svg = compile("circuit c { qubit q\n h q }", Target::Svg)?;

// With relative imports resolved from a path on disk.
let source = load_source("circuit.qrab")?;
let circuit = parse(source.as_str())?;
let latex = render(&circuit, Target::Latex);
let typst = render(&circuit, Target::Typst);
```

`compile` is `parse` followed by `render`. `load_source` expands `import` statements and remembers where each expanded line came from, which is what lets a diagnostic point into the right module: `LoadedSource::origin` maps a line of the expanded source back to its file and line.

`parse` returns `Result<Circuit, Diagnostic>`, and `Diagnostic` implements `std::error::Error` as well as `miette::Diagnostic`, so it prints with source context when a `miette` reporter is installed.

Public AST structs are `#[non_exhaustive]` and should be obtained through `parse`, with public fields available for adjustments afterwards. Enums stay exhaustive so that additions to the model break downstream matches at compile time; before 1.0, minor releases may make such changes.

16 Relationship to qpic

`qrab` draws qpic's visual model. All 44 of qpic's golden circuits and its 64 documented examples are translated into the test corpus, and every one of them is compiled through both TeX and Typst on every run.

The differences are deliberate:

qpic	**qrab**
Positional commands, wire names in the first columns	Statements with keywords, wires named in declarations
Token substitution macros	`let`, `style`, and `fn`, resolved into a typed tree
`\input` of another file	`import`, path-aware and cycle-checked
Labels are LaTeX	Labels are plain text in every backend; LaTeX goes in a `backend` block
Uppercase directives (`VERTICAL`, `SCALE`, …)	A `layout` block
Shapes clip their labels	Shapes grow to fit them (Section 13.2)
One output: TikZ	Three outputs from one model

Parity and its evidence are tracked in `docs/qpic-coverage.md`.

17 Appendix: statement index

Newlines terminate statements, `;` separates several on one line, and `//` starts a comment. An omitted wire list selects every active wire on `end`, `barrier`, `label`, `labels`, `equals`, `brace`, `note`, `cut`, `space`, and `touch`, and every *inactive* wire on `start`; a statement is rejected when that default selection turns out to be empty.

Statement	Summary
`import "path"`	Load a declaration-only module, relative to this file.
`let name = value`	Name a label or a colour.
`style name { ... }`	Name a set of style fields.
`fn name(a, b) { ... }`	Name a sequence of operations over wire parameters.
`circuit name { ... }`	The circuit. One per root file.
`layout { ... }`	Shared geometry.
`backend latex|typst { ... }`	Verbatim code for one target only.
`qubit`, `bit`, `hidden`	Declare a wire or a fixed-size array.
`ellipsis name`	Declare a wireless `...` row.
`autowires`	Create unknown quantum wires on first use.
`h x y z s t`	Built-in single-wire gates.
`gate "L" on ...`	An arbitrary box.
`phase "L" on ...`	A labelled circle.
`measure ... [as "L"] [using tag]`	Meter, D marker, or tag marker.
`swap a, b`	Swap notation.
`barrier [wires]`	Dashed barrier.
`set ... to quantum|classical|hidden [as "v"]`	Change wire kind.
`start ... [as "L"]`	Begin a deferred wire.
`end ... [as "L"]`	Stop a wire.
`bundle "n" on wire`	Multiplicity slash.
`permute ...`	Reorder rows from here on.
`space ... with width: n`	Reserve invisible room.
`touch [wires]`	Align later work; optionally draw a slice.
`label "L" on ...`	Text centred across a span.
`labels "A", "B" on ...`	Text on each selected wire.
`brace left|right|both "L" on ...`	Brace around a span.
`equals ["L"] [on ...] [braced ...]`	Decomposition separator.
`note above|below "L" [on ...]`	Wrapped free text beside a slice.
`cut [on ...] [as "L"]`	Stage separator in its own column.
`mark name`	A named position in the operation stream.
`group "L" from m to m|here on ...`	Highlight a marked range.
`repeat n { ... }`	Repeat a block.
`reverse { ... }`	Emit a block backwards.
`reverse from m to m|here`	Replay a marked range backwards.
`parallel { ... }`	Align a layer.
`overlay { ... }`	Force one column.