The string-diagrams package*
Draw string diagrams using TikZ

Paolo Brasolin
paolo.brasolin@gmail.com

v0.4.0 (2026/11/12)

This is major version zero, for initial development: *anything MAY change at any time*. So this is the best time to contribute! You can also keep the `sty` file with your code, and not care at all.

1 Documentation

Load the package with `\usepackage{string-diagrams}`. It loads Ti*k*Z, with its `fit` and `quotes` libraries. The examples below need nothing else, unless they say otherwise.

1.1 Boxes

`/pgf/box`
New: 2023-05-31
Updated: 2026-11-07

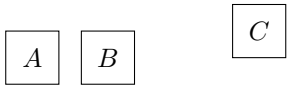
To draw a box, use this style on a node. Its label is in math mode already, so write it without dollars: `{f}`, since `{$f$}` breaks the node.

```
\begin{tikzpicture}
  \node[box] {A};
\end{tikzpicture}
```



Place boxes with any positioning technique of Ti*k*Z, and name them, so that wires can refer to them. The last box below is placed with the `calc` library of Ti*k*Z: load it with `\usetikzlibrary{calc}`.

```
\begin{tikzpicture}
  \node[box] (A) at (0,0) {A};
  \node[box, right of=A] (B) {B};
  \node[box] (C) at ($(B)+(2cm,1em)$) {C};
\end{tikzpicture}
```



*Thanks!

/pgf/box␣ports␣north /pgf/box ports north=⟨*integer*⟩
/pgf/box␣ports␣east /pgf/box ports east=⟨*integer*⟩
/pgf/box␣ports␣south /pgf/box ports south=⟨*integer*⟩
/pgf/box␣ports␣west /pgf/box ports west=⟨*integer*⟩

New: 2023-06-12
Updated: 2026-11-07

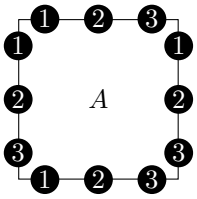
These keys open ports on the sides of a box: a box has none until you open them. As other Ti*k*Z keys, you can set them on a picture or a scope, for all the boxes in it. Refer to a port by its side and number: ports are numbered from left to right on the north and south sides, and from top to bottom on the east and west sides. A port that a box doesn't have is an error.

```
\begin{tikzpicture}[
  marker/.style={circle, fill, inner sep=1pt, text=white},
]

\node[
  box,
  box ports north=3,
  box ports east=3,
  box ports south=3,
  box ports west=3,
  minimum width=6em,
  minimum height=6em,
] (A) {A};

\foreach \side in {north,east,south,west}
  \foreach \index in {1,...,3}
    \node[marker] at (A.\side.\index) {\index};

\end{tikzpicture}
```



/pgf/box␣ports /pgf/box ports=⟨*integer*⟩/⟨*integer*⟩/⟨*integer*⟩/⟨*integer*⟩

New: 2023-06-12
Updated: 2026-11-07

The `box ports` key sets the ports of all sides at once. It takes exactly four counts, for the north, east, south and west sides.

```
\begin{tikzpicture}[
  marker/.style={circle, fill, inner sep=1pt},
]

\node[box, box ports=1/2/3/4] (A) {A};

\foreach \side/\n in {north/1,east/2,south/3,west/4}
  \foreach \index in {1,...,\n}
    \node[marker] at (A.\side.\index) {};

\end{tikzpicture}
```



You can also give the same value to the `box` key itself. Without a value, `box` keeps the counts in force.

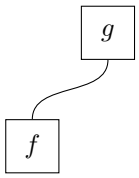
1.2 Wires

`\wires` — `\wires[⟨TikZ keys⟩]{⟨wires⟩}`

New: 2023-05-31
Updated: 2026-11-11

To connect nodes, use `\wires`. Its optional argument holds Ti*k*Z keys for all its wires. Its mandatory argument lists the wires, separated by commas. A wire goes from an anchor of a node to an anchor of a node: ⟨*node*⟩.⟨*anchor*⟩ = ⟨*node*⟩.⟨*anchor*⟩.

```
\begin{tikzpicture}
  \node[box=1/0/0/0] (f) at (0,0) {f};
  \node[box=0/0/1/0] (g) at (1,1.5) {g};
  \wires{ f.north.1 = g.south.1 }
\end{tikzpicture}
```



Wires start and end at these anchors:

- the sides of a node: `north`, `east`, `south` and `west`;
- its corners: `north east` and so on;
- the ports of a box, such as `east.1`.

Wires leave sides and ports straight out, and corners diagonally. Other anchors, such as `center`, have no direction, so wires can't start or end there.

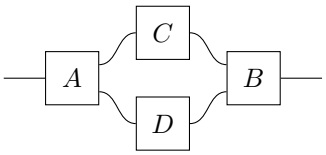
The boundary of the picture is a node too, called `@`. Wires to its sides are the loose ends of the diagram. They go straight out to their side, or bend towards it.

Wires that begin alike can share their beginning. Write it once, then `=` and a group in braces, with the rest of each wire. A beginning ends after the name of a node, or at the `=` of a wire. So a group can hold:

- the anchors of the node the wires start from, as in `A = { east.1 = C.west, west = @.west }`;
- the targets of an anchor, as in `f.north.1 = { g.south.1, h.south.1 }`;
- the anchors of the node of a target, as in `f.north.1 = { g = { south.1, south.2 } }`.

Groups can nest, and every grouping of the same wires draws the same wires. Only beginnings are shared: wires that end alike can't share their end. A node can appear in more than one entry, and an anchor can carry several wires, in and out.

```
\begin{tikzpicture}[scale=0.6]
  \node[box=0/2/0/1] (A) at (-2, 0) {A};
  \node[box=0/1/0/2] (B) at (+2, 0) {B};
  \node[box=0/1/0/1] (C) at ( 0,+1) {C};
  \node[box=0/1/0/1] (D) at ( 0,-1) {D};
  \wires{
    A = { east.1 = C.west
        , east.2 = D.west
        , west = @.west },
    B.east = @.east,
    C.east = B.west.1,
    D.east = B.west.2,
  }
\end{tikzpicture}
```

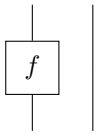


The first `\wires` of a picture draws the boundary around the nodes drawn so far, with a margin. Later calls in the same picture use the same boundary. So draw all nodes before the wires.

A wire across the boundary runs through its middle. To place it elsewhere, for example beside a box:

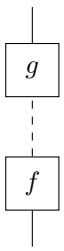
1. Place a coordinate there, with a path as below: `\coordinate` doesn't count towards the size of the picture, so the boundary might miss it.
2. Wire the coordinate to both sides of the boundary.

```
\begin{tikzpicture}
  \node[box=1/0/1/0] (f) at (0,0) {f};
  \path (0.8,0) coordinate (i);
  \wires{
    f = { north.1 = @.north
        , south.1 = @.south },
    i = { north = @.north
        , south = @.south },
  }
\end{tikzpicture}
```



The keys of a call apply to all its wires. Keys in brackets can also begin any entry, group or target, as in `f.north.1 = [dashed] g.south.1`. They apply to every wire they reach, and add up from the call down to each wire, so the keys closest to a wire win. As in Ti*k*Z, keys in brackets end at the first `]` outside braces: so a key with brackets goes in braces, as in `[-{Stealth[length=6pt]}]`.

```
\begin{tikzpicture}
  \node[box=1/0/1/0] (f) at (0,0) {f};
  \node[box=1/0/1/0] (g) at (0,1.5) {g};
  \wires{
    f.south.1 = @.south,
    f.north.1 = [dashed] g.south.1,
    g.north.1 = @.north,
  }
\end{tikzpicture}
```



All keys come after the angles that `\wires` computes, so `out` and `in` override those angles. `bend left` and `bend right` bend a wire between its ends. Each wire is a Ti*k*Z edge, so its keys style its stroke too.

Wires take their directions from the names of anchors, in the frame of the picture. So you can shift and scale a picture, but not rotate or mirror it. Rotated nodes aren't supported either.

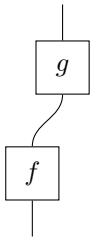
1.3 Anchors left out

`/tikz/wire␣flow` `/tikz/wire flow=⟨direction⟩`

New: 2026-11-11

Sets the direction of wires between ends without anchors: `north`, `east`, `south` or `west`. The synonyms `up`, `right`, `down` and `left` work as for the `grow` key of Ti*k*Z. `none` removes the flow. Set it on a picture, a scope, a call, a group, a wire or a path, as any key.

```
\begin{tikzpicture}[wire flow=north]
  \node[box=1/0/1/0] (f) at (0,0) {f};
  \node[box=1/0/1/0] (g) at (.4,1.4) {g};
  \wires{ @ = f, f = g, g = @ }
\end{tikzpicture}
```

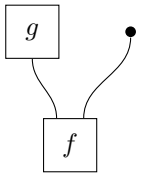


You can leave out the anchor of either end of a wire, or of both, as in `f = g`. An end without anchor is the side that the wire crosses going straight on. If that side has one port, the wire ends at the port. If it has more, the package can't choose: it reports an error, so name the port.

The direction of the wire comes from:

- the anchored end, if there is one. A wire from a corner goes diagonally, so it turns to the axis of the flow, if set, or else to the axis that separates its ends more.
- the flow, if both ends are bare.
- else, the axis that separates the centers of the two nodes more. Two bare ends on nodes with the same center need a flow.

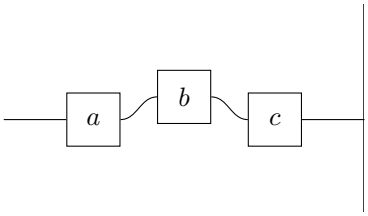
```
\begin{tikzpicture}
  \node[box=2/0/0/0] (f) at (0,0) {f};
  \node[box=0/0/1/0] (g) at (-.5,1.5) {g};
  \node[dot] (d) at (.8,1.5) {};
  \wires{ f = { north.1 = g, north.2 = d } }
\end{tikzpicture}
```



The flow decides even when the target lies behind: then the wire leaves along the flow, and loops round. A wire goes from its start to its target, so swap them to turn it. Without a flow, the axis flips when two nodes are near a diagonal: a small move of a node can turn their wire. Set a flow to keep it.

With a flow, loose ends need no anchors either: `@ = f` leaves the boundary behind `f`, and `f = @` reaches it ahead of `f`.

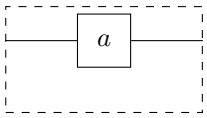
```
\begin{tikzpicture}[wire flow=east]
  \node[box=0/1/0/1] (a) at (0,0) {a};
  \node[box=0/1/0/1] (b) at (1.2,.3) {b};
  \node[box=0/1/0/1] (c) at (2.4,0) {c};
  \wires{ @ = a, a = b, b = c, c = @ }
\end{tikzpicture}
```



1.4 Frames

An end faces inwards when its node holds the other node: its box contains the box of the other, borders that touch counting as inside. Then the wire leaves or enters that end the other way, and a side without ports is reached straight across. This is how loose ends work, since the boundary holds the nodes of the picture. A frame drawn by hand works the same way.

```
\begin{tikzpicture}
  \node[box=0/1/0/1] (a) at (0,.25) {a};
  \node[draw, dashed, minimum width=2.6cm,
        minimum height=1.4cm] (F) at (0,0) {};
  \wires{ F.west = a, a = F.east }
\end{tikzpicture}
```



Nodes apart both face out, and a wire between them may loop round. Nodes that overlap otherwise, or have the same box, face out too. But then the package can't tell which way a bare end goes, so give both ends anchors.

A wire from a node to itself loops round the node, and a wire from the boundary to itself crosses the picture. With both ends bare, such a wire needs a flow: with `wire flow=east`, `f = f` is `f.east = f.west`, and `@ = @` is `@.west = @.east`.

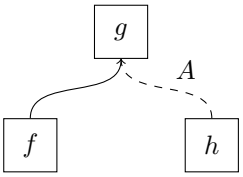
1.5 The wire key

`/tikz/wire` `\draw (⟨node⟩.⟨anchor⟩) to[wire] (⟨node⟩.⟨anchor⟩);`
`/tikz/every␣wire` `\path (⟨node⟩.⟨anchor⟩) edge[wire] (⟨node⟩.⟨anchor⟩);`

New: 2026-11-11

The `wire` key draws a `to` path or an edge of Ti*k*Z as a wire, as `\wires` does. The `every wire` style applies right after it.

```
\begin{tikzpicture}
  \node[box=1/0/0/0] (f) at (0,0) {f};
  \node[box=0/0/1/0] (g) at (1.2,1.5) {g};
  \node[box=1/0/0/0] (h) at (2.4,0) {h};
  \draw[->] (f.north.1) to[wire] (g.south.1);
  \path (h) edge[wire, dashed, "$A$"'] (g);
\end{tikzpicture}
```



Ends can be bare, as in `\wires`. Loose ends need the boundary, which only `\wires` draws: so call `\wires` first in the picture. The key sets the `out` and `in` keys of Ti*k*Z, so

Ti*k*Z draws the curve and places its labels. Keys after `wire` override its angles, and keys before it don't. A wrong end raises an error, and the wire isn't drawn.

On an edge, `every wire` styles the stroke, as `every edge` does. On a `to` path it doesn't, as `every to` doesn't: there, keys shape the curve, but only the keys of the path draw it.

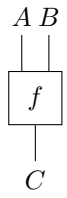
1.6 Labels

`/tikz/beyond`
`/tikz/beside`
`/tikz/beyond␣end`
`/tikz/beyond␣start`
`/tikz/beside␣end`
`/tikz/beside␣start`

New: 2026-11-11

These keys place a label as the `auto` key of Ti*k*Z does, but past the end of a wire, or beside it and clear of the box. `beyond end` is short for `at end, beyond`, and so on.

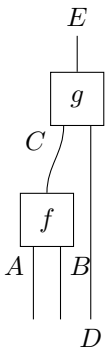
```
\begin{tikzpicture}
  \node[box=2/0/1/0] (f) {f};
  \wires{ f = {
    north.1 = ["$A$" beyond end] @,
    north.2 = ["$B$" beyond end] @,
    south.1 = ["$C$" beyond end] @ } }
\end{tikzpicture}
```



Wires take three kinds of labels:

- The type of a wire, along it. Use the quotes of Ti*k*Z, as in `f.north.1 = ["$A$"] g.south.1`. The label goes to the left of the wire, and `"$A$"'` puts it to the right.
- The interface of the diagram, at its loose ends. Use `beyond end`, or `beyond start` on a wire from the boundary: the label goes past the end.
- The type of a port, at its box. Use `beside start` or `beside end` on the wire, or `beside` in the label of the box, as in `label={[beside]north.1:$A$}`. The label goes beside the port, clear of the box, to the left of a wire leaving it; `swap` puts it to the right. In the label of the box, it belongs to the port, so it shows once, however many wires the port has.

```
\begin{tikzpicture}[wire flow=north]
  \node[box=1/0/2/0,
    label={[swap, beside]south.1:$A$},
    label={[beside]south.2:$B$}] (f) at (0,0)
        {f};
  \node[box=1/0/2/0] (g) at (.4,1.6) {g};
  \wires{
    @ = { f.south.1, f.south.2,
          ["$D$" beyond start] g.south.2 },
    f.north.1 = ["$C$"] g.south.1,
    g.north.1 = ["$E$" beyond end] @,
  }
\end{tikzpicture}
```



The keys only choose the anchor of the label, from the direction of the wire there; `at end`, `at start` and `pos` still place it. On a path, they act as at the end the label is nearer. In the label of a node, they read the name of the anchor. Where a label touches a wire, swap it.

1.7 Dots

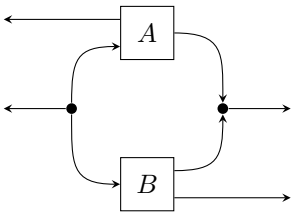
/pgf/dot

New: 2023-05-31

Updated: 2026-11-07

To split and join wires, use dots. Wires attach to their sides and corners, and leave corners diagonally. Style wires as you like.

```
\begin{tikzpicture}
  \node[box=0/1/0/2] (A) at ( 0,+1) {A};
  \node[box=0/2/0/1] (B) at ( 0,-1) {B};
  \node[dot] (x) at (+1, 0) {};
  \node[dot] (y) at (-1, 0) {};
  \wires[looseness=1.5,-stealth]{
    A = { east.1 = x.north
        , west.1 = @.west },
    B = { east.1 = x.south
        , east.2 = @.east },
    y = { north = A.west.2
        , south = B.west
        , west = @.west },
    x = { east = @.east },
  }
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \node[dot] (c) at (0,0) {};
  \node[dot] (m) at (0,1.2) {};
  \wires{
    c = { south = @.south
        , north west = m.south west
        , north east = m.south east },
    m = { north = @.north },
  }
\end{tikzpicture}
```



1.8 Ti*k*Z pitfalls

Some behaviors of Ti*k*Z surprise in diagrams.

- *Styles of the picture.* `\wires` draws its wires as edges, but with the default `every edge` of Ti*k*Z: so only `every wire` and `every curve to` reach them, not `every to` or `every edge`. A `to[wire]` or `edge[wire]` written by hand follows Ti*k*Z.
- *Options of quotes.* The options of a path end at the first `]` outside braces, so `"x"[blue]` fails. Write `"x" blue`, or `"x" {pos=1, below}`.
- *Quotes and babel.* Languages of babel that make `"` a shorthand break quotes: load the `babel` library of Ti*k*Z.
- *Arrow tips.* `\draw[->]` with edges adds a stray tip to the path itself: write `\path[->]` instead.
- *Nodes on a path.* After a node on a path, the next `to` or edge starts at that node, not at the coordinate before it.

- *Wires in a row.* In `(f.north.1) to[wire] (g.west.1) to[wire] (h.west.1)`, the second wire leaves `g.west.1` westwards, back the way the first came.
- *Feedback loops.* A wire between two level ends of a node, as `f.east.1 = f.west.1`, leaves and enters straight, so Ti*k*Z draws it as a line through the node. Give it angles, as in `f = [out=60, in=120, looseness=2.5] f`.
- *Nodes drawn late.* A node drawn after the first `\wires` of a picture can stick out of the boundary. Then it overlaps the boundary, and its wires to `@` face out.
- *Markings.* The markings of the `decorations.markings` library fail on wires of no length, with "Dimension too large".

1.9 Changes from earlier versions

Version 0.4.0 changes some behaviors of 0.3.0:

- `every to` no longer styles the wires of `\wires`: use `every wire`.
- An end without anchor is no longer an error.
- The boundary faces inwards only towards the nodes it holds, as any node does.

Up to version 0.2.1, `\wires` took a third argument, listing the loose ends. Loose ends are now wires to the boundary, and `\wires` ignores a third argument.

2 Implementation

Open the DocStrip guards and set the internal namespace prefix (as per LaTeX3 DocStrip convention).

```
1 ⟨*package⟩
2 ⟨@@=stridi⟩
```

Require a LaTeX with hooks that take arguments, and load the essential support (expl3, and Ti*k*Z with the `fit` library that the boundary of `\wires` needs, and the `quotes` library that labels wires) "up-front".

```
3 \NeedsTeXFormat{LaTeX2e}[2023/06/01]
4 \RequirePackage{expl3}[2023/06/05]
5 \RequirePackage{tikz}[2023/01/15]
6 \usetikzlibrary{fit}
7 \usetikzlibrary{quotes}
```

Identify the package and give the over all version information.

```
8 \ProvidesExplPackage
9   {string-diagrams}
10  {2026/11/12}
11  {0.4.0}
12  {Draw string diagrams using TikZ}
```

/pgf/box␣ports␣north
/pgf/box␣ports␣east
/pgf/box␣ports␣south
/pgf/box␣ports␣west
/pgf/box␣ports

Define high level keys to configure the number of ports on each side. Boxes start with no ports. The shortcut checks that it gets four counts, and sets the keys of the sides to their

values; given nothing, it leaves them alone, so that the box style without counts keeps the ones in force.

```
\msg_new:nnn { string-diagrams } { port-counts }
  {
    Box~ports~take~four~counts,~for~the~north,~east,~south~and~west~sides,~
    as~in~1/2/1/2,~not~‘#1’.
  }
\seq_new:N \l__stridi_port_counts_seq
\pgfkeys{
  /pgf/box~ports~north/.initial=0,
  /pgf/box~ports~east/.initial=0,
  /pgf/box~ports~south/.initial=0,
  /pgf/box~ports~west/.initial=0,
  /pgf/box~ports/.code={
    \tl_if_blank:nF { #1 }
      {
        \seq_set_split:Nnn \l__stridi_port_counts_seq { / } { #1 }
        \int_compare:nNnTF { \seq_count:N \l__stridi_port_counts_seq } = { 4 }
          {
            \exp_args:Ne \pgfkeysalso
              {
                /pgf/box~ports~north = \seq_item:Nn \l__stridi_port_counts_seq { 1 } ,
                /pgf/box~ports~east  = \seq_item:Nn \l__stridi_port_counts_seq { 2 } ,
                /pgf/box~ports~south = \seq_item:Nn \l__stridi_port_counts_seq { 3 } ,
                /pgf/box~ports~west  = \seq_item:Nn \l__stridi_port_counts_seq { 4 } ,
              }
          }
          { \msg_error:nnn { string-diagrams } { port-counts } { #1 } }
      }
  },
}
```

(*End of definition for* `/pgf/box ports north` *and others. These functions are documented on page* 2.)

`\__stridi_intersect_hv_lines_through:NN` Calculates the intersection of two lines parallel to axes passing through given points on the plane.

`#1` : Point through which the vertical line passes
`#2` : Point through which the horizontal line passes

```
\cs_new_protected:Nn \__stridi_intersect_hv_lines_through:NN {
  \pgfextractx { \pgf@xa } { #1 }
  \pgfextracty { \pgf@ya } { #2 }
  \pgfpoint { \pgf@xa } { \pgf@ya }
}
```

(*End of definition for* `\__stridi_intersect_hv_lines_through:NN`.)

`\__stridi_subdivide_segment:nNNNNN` Defines the port anchors of a side of a box: equally spaced points on a segment. Anchors are defined globally, by the first box with enough ports on that side, so each one checks the port count of the box it is asked of. The box is named in the error when an anchor of a named node is asked for, but not while placing a node by one of its own anchors.

#1 : Side of the box
#2 : Points count
#3 : Point containing the x coordinate of the starting point
#4 : Point containing the y coordinate of the starting point
#5 : Point containing the x coordinate of the ending point
#6 : Point containing the y coordinate of the ending point

```
\msg_new:nnn { string-diagrams } { missing-port }
  { #1~has~no~port~#2.#3,~since~its~#2~side~has~#4. }
\cs_new_protected:Nn \__stridi_subdivide_segment:nNNNNN {
  \int_step_inline:nnnn { #2 } { -1 } { 1 } {
    \cs_if_exist:cTF
      { pgf@anchor@box@#1.##1 }
      { \prg_break: }
      { \prg_do_nothing: }
    \cs_new_nopar:cpn
      { pgf@anchor@box@#1.##1 }
      {
        \int_compare:nNnTF { ##1 } > { #2 }
          {
            \msg_error:nneeee { string-diagrams } { missing-port }
              {
                \cs_if_exist:NTF \pgfreferencednodename
                  { Box~`\pgfreferencednodename' }
                  { This~box }
              }
              { #1 } { ##1 } { #2 }
            \pgfpointlineattime
              { 0.5 }
              { \__stridi_intersect_hv_lines_through:NN { #3 } { #4 } }
              { \__stridi_intersect_hv_lines_through:NN { #5 } { #6 } }
          }
          {
            \pgfmathdivide
              { 2 * ##1 - 1 }
              { 2 * #2 }
            \pgfpointlineattime
              { \pgfmathresult }
              { \__stridi_intersect_hv_lines_through:NN { #3 } { #4 } }
              { \__stridi_intersect_hv_lines_through:NN { #5 } { #6 } }
          }
      }
  }
}
```

(*End of definition for* `\__stridi_subdivide_segment:nNNNNN`.)

`box` Define a rectangular shape with configurable ports.

```
\pgfdeclareshape{box}{

  % Inherit all the structure of rectangle
  \inheritsavedanchors[from=rectangle]
  \clist_map_inline:nn
```

```
    {
      north~west,  north, north~east,
            west, center,       east,
        mid~west,    mid,   mid~east,
       base~west,   base,  base~east,
      south~west,  south, south~east,
    }
    { \inheritanchor[from=rectangle]{#1} }
  \inheritanchorborder[from=rectangle]
  \inheritbackgroundpath[from=rectangle]

  % Dump port counts into saved macros
  \savedmacro\portsnorth
    {\pgfmathtruncatemacro\portsnorth{\pgfkeysvalueof{/pgf/box~ports~north}}}
  \savedmacro\portseast
    {\pgfmathtruncatemacro\portseast{\pgfkeysvalueof{/pgf/box~ports~east}}}
  \savedmacro\portssouth
    {\pgfmathtruncatemacro\portssouth{\pgfkeysvalueof{/pgf/box~ports~south}}}
  \savedmacro\portswest
    {\pgfmathtruncatemacro\portswest{\pgfkeysvalueof{/pgf/box~ports~west}}}

  % Add ports definitions to shape definition
  \expandafter\pgfutil@g@addto@macro\csname pgf@sh@s@box\endcsname{
    \__stridi_subdivide_segment:nNNNNN { north } { \portsnorth }
      { \southwest } { \northeast } { \northeast } { \northeast }
    \__stridi_subdivide_segment:nNNNNN { east } { \portseast }
      { \northeast } { \northeast } { \northeast } { \southwest }
    \__stridi_subdivide_segment:nNNNNN { south } { \portssouth }
      { \southwest } { \southwest } { \northeast } { \southwest }
    \__stridi_subdivide_segment:nNNNNN { west } { \portswest }
      { \southwest } { \northeast } { \southwest } { \southwest }
  }

}
```

(*End of definition for* box.)

/pgf/box Define style to draw boxes.

```
\ExplSyntaxOff
\tikzset{
  box/.default={},
  box/.style args={#1}{
    shape=box,
    draw,
    inner sep=.5em,
    minimum width=2em,
    minimum height=2em,
    execute at begin node=$,
    execute at end node=$,
    /pgf/box ports=#1,
  },
}
\ExplSyntaxOn
```

(*End of definition for* `/pgf/box`. *This function is documented on page* 1.)

`/pgf/dot` Define style to draw dots.

```
\ExplSyntaxOff
\tikzset{
  dot/.style={
    shape=circle,
    fill,
    inner sep=0,
    minimum width=0.4em,
  },
}
\ExplSyntaxOn
```

(*End of definition for* `/pgf/dot`. *This function is documented on page* 8.)

`\__stridi_clamp_to_node:n` Moves the point in the internal `pgf` variables to the nearest point of the bounding rectangle of the given node, if it lies outside it.

```
\dim_new:N \l__stridi_x_dim
\dim_new:N \l__stridi_y_dim
\cs_new_protected:Nn \__stridi_clamp_to_node:n {
  \dim_set:Nn \l__stridi_x_dim { \pgf@x }
  \dim_set:Nn \l__stridi_y_dim { \pgf@y }
  \pgfpointanchor { #1 } { north~east }
  \dim_set:Nn \l__stridi_x_dim { \dim_min:nn { \l__stridi_x_dim } { \pgf@x } }
  \dim_set:Nn \l__stridi_y_dim { \dim_min:nn { \l__stridi_y_dim } { \pgf@y } }
  \pgfpointanchor { #1 } { south~west }
  \dim_set:Nn \l__stridi_x_dim { \dim_max:nn { \l__stridi_x_dim } { \pgf@x } }
  \dim_set:Nn \l__stridi_y_dim { \dim_max:nn { \l__stridi_y_dim } { \pgf@y } }
  \pgfpoint { \l__stridi_x_dim } { \l__stridi_y_dim }
}
```

(*End of definition for* `\__stridi_clamp_to_node:n`.)

`\__stridi_calctarget:nnnn` Performs the core calculation for `\__stridi_end_coordinate:NnnnnN`. The result is stored in the internal `pgf` variables. A wire from a corner to a side named in it, such as south west to south or west, ends offset along its diagonal. A bend that would end past a corner of the boundary ends at that corner.

`#1` : source node
`#2` : source anchor
`#3` : target node (can be boundary)
`#4` : target anchor

```
\cs_new_protected:Nn \__stridi_calctarget:nnnn {
  \pgfpointanchor{#1}{#2} \pgf@xa=\pgf@x \pgf@ya=\pgf@y
  \pgfpointanchor{#3}{#4} \pgf@xb=\pgf@x \pgf@yb=\pgf@y
  \regex_match_case:nn
  {
    { \A north ; north \  west \b } { \pgfpoint { \pgf@xa-.618*(\pgf@yb-\pgf@ya) } { \pgf@yb
    { \A north ; north \  east \b } { \pgfpoint { \pgf@xa+.618*(\pgf@yb-\pgf@ya) } { \pgf@yb
    { \A south ; south \  west \b } { \pgfpoint { \pgf@xa-.618*(\pgf@ya-\pgf@yb) } { \pgf@yb
```

```
    { \A south ; south \  east \b } { \pgfpoint { \pgf@xa+.618*(\pgf@ya-\pgf@yb) } { \pgf@yb
    { \A east  ; north \  east \b } { \pgfpoint { \pgf@xb } { \pgf@ya+.618*(\pgf@xb-\pgf@xa)
    { \A east  ; south \  east \b } { \pgfpoint { \pgf@xb } { \pgf@ya-.618*(\pgf@xb-\pgf@xa)
    { \A west  ; north \  west \b } { \pgfpoint { \pgf@xb } { \pgf@ya+.618*(\pgf@xa-\pgf@xb)
    { \A west  ; south \  west \b } { \pgfpoint { \pgf@xb } { \pgf@ya-.618*(\pgf@xa-\pgf@xb)
    { \A north ; north \b } { \pgfpoint { \pgf@xa } { \pgf@yb } }
    { \A north ; south \b } { \pgfpoint { \pgf@xa } { \pgf@yb } }
    { \A south ; north \b } { \pgfpoint { \pgf@xa } { \pgf@yb } }
    { \A south ; south \b } { \pgfpoint { \pgf@xa } { \pgf@yb } }
    { \A north ; east  \b } { \pgfpoint { \pgf@xa+.618*(\pgf@yb-\pgf@ya) } { \pgf@yb } }
    { \A south ; east  \b } { \pgfpoint { \pgf@xa-.618*(\pgf@yb-\pgf@ya) } { \pgf@yb } }
    { \A north ; west  \b } { \pgfpoint { \pgf@xa-.618*(\pgf@yb-\pgf@ya) } { \pgf@yb } }
    { \A south ; west  \b } { \pgfpoint { \pgf@xa+.618*(\pgf@yb-\pgf@ya) } { \pgf@yb } }
    { \A east  ; east  \b } { \pgfpoint { \pgf@xb } { \pgf@ya } }
    { \A east  ; west  \b } { \pgfpoint { \pgf@xb } { \pgf@ya } }
    { \A west  ; east  \b } { \pgfpoint { \pgf@xb } { \pgf@ya } }
    { \A west  ; west  \b } { \pgfpoint { \pgf@xb } { \pgf@ya } }
    { \A east  ; north \b } { \pgfpoint { \pgf@xb } { \pgf@ya+.618*(\pgf@xb-\pgf@xa) } }
    { \A west  ; north \b } { \pgfpoint { \pgf@xb } { \pgf@ya-.618*(\pgf@xb-\pgf@xa) } }
    { \A east  ; south \b } { \pgfpoint { \pgf@xb } { \pgf@ya-.618*(\pgf@xb-\pgf@xa) } }
    { \A west  ; south \b } { \pgfpoint { \pgf@xb } { \pgf@ya+.618*(\pgf@xb-\pgf@xa) } }
  } { #4 ; #2 }
  \__stridi_clamp_to_node:n { #3 }
}
```

(*End of definition for* `\__stridi_calctarget:nnnn`.)

`\l__stridi_angle_int` Scratch variable for `\__stridi_anchor_to_angle:Nn`.

```
\int_new:N \l__stridi_angle_int
```

(*End of definition for* `\l__stridi_angle_int`.)

`\__stridi_anchor_to_angle:Nn` Calculates the angle at which a wire leaves an anchor outwards, from its name. Wires leave sides and their ports straight out, and corners diagonally; other anchors have no angle, and leave the variable empty. An end that faces inwards turns this angle round (see `\__stridi_wire:`).

`#1` : token list variable
`#2` : anchor

```
\cs_new_protected:Nn \__stridi_anchor_to_angle:Nn {
  \regex_match_case:nnTF
    {
      { \A north \  east \b } { \int_set:Nn \l__stridi_angle_int {   45 } }
      { \A north \  west \b } { \int_set:Nn \l__stridi_angle_int {  135 } }
      { \A south \  east \b } { \int_set:Nn \l__stridi_angle_int {  -45 } }
      { \A south \  west \b } { \int_set:Nn \l__stridi_angle_int { -135 } }
      { \A north \b } { \int_set:Nn \l__stridi_angle_int {  90 } }
      { \A east  \b } { \int_set:Nn \l__stridi_angle_int {   0 } }
      { \A south \b } { \int_set:Nn \l__stridi_angle_int { -90 } }
      { \A west  \b } { \int_set:Nn \l__stridi_angle_int { 180 } }
    }
    { #2 }
```

```
    { \tl_set:Nf #1 { \int_use:N \l__stridi_angle_int } }
    { \tl_clear:N #1 }
}
```

(*End of definition for* `\__stridi_anchor_to_angle:Nn`.)

`\l__stridi_oangle_tl`
`\l__stridi_iangle_tl`
`\l__stridi_source_tl`
`\l__stridi_target_tl`

Initialize scratch variables required by `\__stridi_wire:`.

```
\tl_new:N \l__stridi_oangle_tl
\tl_new:N \l__stridi_iangle_tl
\tl_new:N \l__stridi_source_tl
\tl_new:N \l__stridi_target_tl
```

(*End of definition for* `\l__stridi_oangle_tl` *and others.*)

`\__stridi_draw_wire:nnnnn`
`\__stridi_draw_wire_edge:nnnnn`
`\__stridi_wire_coordinate:nn`

Draws a wire as an edge with the `wire` key, followed by the keys of the call, so that it is drawn as on any path of Ti*k*Z: the key sets its angles and moves its ends on the boundary, and raises an error for each end without an angle. The edge is an implementation detail, so it gets Ti*k*Z's default `every edge`, which only draws it: the wires of `\wires` are styled by `every wire` and the keys of the call, not by the `every edge` of the picture. The coordinates are expanded first, since Ti*k*Z hands the target to the key as written; an end without an anchor is a bare node. Ti*k*Z resolves the start of a path while reading it, before any key, so a start on a node that doesn't exist, or at an anchor its node lacks, would get only the error of pgf, which for an anchor is about a mathematical function: the start is checked first, and such a wire is reported and not drawn.

`#1` : Ti*k*Z keys
`#2` : source node
`#3` : source anchor
`#4` : target node
`#5` : target anchor

```
\cs_new_protected:Nn \__stridi_draw_wire:nnnnn {
  \cs_if_exist:cTF { pgf@sh@ns@ #2 }
    {
      \tl_if_empty:nTF { #3 }
        { \__stridi_draw_wire_edge:nnnnn { #1 } { #2 } { #3 } { #4 } { #5 } }
        {
          \__stridi_if_anchor:nnTF { #2 } { #3 }
            { \__stridi_draw_wire_edge:nnnnn { #1 } { #2 } { #3 } { #4 } { #5 } }
            { \__stridi_no_anchor:nn { #2 } { #3 } }
        }
    }
    {
      \msg_error:nne { string-diagrams } { end-no-node }
        { \__stridi_wire_coordinate:nn { #2 } { #3 } }
    }
}
\cs_new_protected:Nn \__stridi_draw_wire_edge:nnnnn {
  \use:e
    {
      \exp_not:N \path [ every~edge/.style = { draw } ]
        ( \__stridi_wire_coordinate:nn { #2 } { #3 } )
```

```
        edge [ wire, \exp_not:n { #1 } ]
        ( \__stridi_wire_coordinate:nn { #4 } { #5 } ) ;
    }
}
\cs_new:Nn \__stridi_wire_coordinate:nn
  { \exp_not:n { #1 } \tl_if_blank:nF { #2 } { . \exp_not:n { #2 } } }
```

(*End of definition for* `\__stridi_draw_wire:nnnnn`, `\__stridi_draw_wire_edge:nnnnn`, *and* `\__stridi_wire_coordinate:nn`.)

`\__stridi_if_boundary_in_picture_p:`
`\__stridi_if_boundary_in_picture:TF`

Tests whether the current picture has a boundary, by the picture that pgf records for each node: node names outlive their pictures.

```
\prg_new_conditional:Nnn \__stridi_if_boundary_in_picture: { p, T, F, TF }
  {
    % NOTE: the space keeps docstrip from turning the __stridi of the name of node @ into th
    \bool_lazy_and:nnTF
      { \cs_if_exist_p:c { pgf@sh@pi@ @ } }
      { \str_if_eq_p:ee { \use:c { pgf@sh@pi@ @ } } { \pgfpictureid } }
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End of definition for* `\__stridi_if_boundary_in_picture:TF`.)

`\__stridi_draw_boundary:`
`\__stridi_draw_boundary_around:nn`

Creates the boundary node around the picture so far, or around the origin if nothing has been drawn yet (as pgf does, it tells an empty picture by its untouched bounds). A picture gets a single boundary: later calls of `\wires` in it reuse the first one.

`#1` : north east corner to surround
`#2` : south west corner to surround

```
\cs_new_protected:Nn \__stridi_draw_boundary: {
  \__stridi_if_boundary_in_picture:F
    {
      \dim_compare:nNnTF { \pgf@picmaxx } = { -16000pt }
        { \__stridi_draw_boundary_around:nn { 0pt, 0pt } { 0pt, 0pt } }
        {
          \__stridi_draw_boundary_around:nn
            { current~bounding~box.north~east }
            { current~bounding~box.south~west }
        }
    }
}
\cs_new_protected:Nn \__stridi_draw_boundary_around:nn {
  \node [%draw, box=2/2/0/1, dashed, red,
    name=@, node~contents={},
    fit={
      ([shift={(+2em,+1em)}]#1)
      ([shift={(-2em,-1em)}]#2)
    },
  ];
}
```

(*End of definition for* `\__stridi_draw_boundary:` *and* `\__stridi_draw_boundary_around:nn`.)

`\l__stridi_split_seq` Scratch variable for `\__stridi_split_node_anchor:NNn`.

```
\seq_new:N \l__stridi_split_seq
```

(*End of definition for* `\l__stridi_split_seq`.)

`\__stridi_split_node_anchor:NNn` Parses a token list into a node/anchor pair.

`#1` : token list variable to store node
`#2` : token list variable to store anchor
`#3` : token list to be parsed

```
\cs_new_protected:Nn \__stridi_split_node_anchor:NNn {
  \regex_extract_once:nnN { ([^\.]*) \.? (.*) } { #3 } \l__stridi_split_seq
  \seq_pop:NN \l__stridi_split_seq #1  % discard match
  \seq_pop:NN \l__stridi_split_seq #1  % keep 1st matching group (node)
  \seq_pop:NN \l__stridi_split_seq #2  % keep 2nd matching group (anchor)
}
```

(*End of definition for* `\__stridi_split_node_anchor:NNn`.)

`\g__stridi_anchors_seq` The anchors resolved in the outermost picture so far, pictures nested in it included, newest first: each as the picture it was resolved in, its node, its anchor and the two coordinates of its point.

```
\seq_new:N \g__stridi_anchors_seq
```

(*End of definition for* `\g__stridi_anchors_seq`.)

`\__stridi_record_anchor:nn` Records an anchor that pgf has just resolved in a picture, with the point it gave. Ti*k*Z gives a to path its start as a point, not as the anchor that was written, so the `wire` key looks the point up in this record. It runs after every `\pgfpointanchor`; Ti*k*Z passes it node names wrapped in their prefix and suffix, and anchors that may hold macros, such as the variable of a `\foreach`, so both are expanded. The record is cleared when an outermost picture begins: pgf defines `\pgfpictureid` only inside pictures, and restores it when a nested one ends.

`#1` : node
`#2` : anchor

```
\cs_new_protected:Nn \__stridi_record_anchor:nn {
  \cs_if_exist:NT \pgfpictureid
    {
      \seq_gput_left:Ne \g__stridi_anchors_seq
        {
          { \pgfpictureid } { #1 } { #2 }
          { \dim_use:N \pgf@x } { \dim_use:N \pgf@y }
        }
    }
}
\AddToHookWithArguments { cmd/pgfpointanchor/after }
```

```
  { \__stridi_record_anchor:nn {#1} {#2} }
\AddToHook { cmd/pgfpicture/before }
  { \cs_if_exist:NF \pgfpictureid { \seq_gclear:N \g__stridi_anchors_seq } }
```

(*End of definition for* `\__stridi_record_anchor:nn`.)

`\__stridi_find_anchor:NNw`
`\__stridi_find_anchor_check:NNnnnnnnn`

Finds the newest anchor recorded at a point in the current picture, within 0.01pt, and stores its node and anchor; it stores nothing if there is none.

`#1` : token list variable to store the node
`#2` : token list variable to store the anchor
`#3` : the point, as $\langle x \rangle$,$\langle y \rangle$, ended by `\q_stop`

```
\cs_new_protected:Npn \__stridi_find_anchor:NNw #1#2#3 , #4 \q_stop {
  \tl_clear:N #1
  \tl_clear:N #2
  \seq_map_inline:Nn \g__stridi_anchors_seq
    { \__stridi_find_anchor_check:NNnnnnnnn #1 #2 { #3 } { #4 } ##1 }
}
\cs_new_protected:Nn \__stridi_find_anchor_check:NNnnnnnnn {
  \bool_lazy_all:nT
    {
      { \str_if_eq_p:ee { #5 } { \pgfpictureid } }
      { \dim_compare_p:n { \dim_abs:n { #8 - ( #3 ) } < 0.01pt } }
      { \dim_compare_p:n { \dim_abs:n { #9 - ( #4 ) } < 0.01pt } }
    }
    {
      \tl_set:Nn #1 { #6 }
      \tl_set:Nn #2 { #7 }
      \seq_map_break:
    }
}
```

(*End of definition for* `\__stridi_find_anchor:NNw` *and* `\__stridi_find_anchor_check:NNnnnnnnn`.)

`\__stridi_if_anchor:nn`*TF*

Tests whether a node has an anchor: a name its shape defines, or an angle, which pgf reads as a point of the border. The ports of boxes are defined for all boxes at once, by the first box with enough ports on a side, so a port of a box is checked against the count of that box.

```
\prg_new_protected_conditional:Nnn \__stridi_if_anchor:nn { TF }
  {
    \__stridi_port_of_box:nn { #1 } { #2 }
    \seq_if_empty:NTF \l__stridi_port_seq
      {
        \cs_if_exist:cTF { pgf@anchor@ \use:c { pgf@sh@ns@ #1 } @ #2 }
          { \prg_return_true: }
          {
            \regex_match:nnTF { \A [+\-]? [0-9.]+ \Z } { #2 }
              { \prg_return_true: }
              { \prg_return_false: }
          }
      }
      {
```

```
        \__stridi_side_ports:neN { #1 } { \seq_item:Nn \l__stridi_port_seq { 2 } } \l__strid
        \int_compare:nTF
          { 1 <= \seq_item:Nn \l__stridi_port_seq { 3 } <= \l__stridi_ports_int }
          { \prg_return_true: }
          { \prg_return_false: }
      }
  }
\prg_generate_conditional_variant:Nnn \__stridi_if_anchor:nn { nV , VV } { TF }
```

(*End of definition for* `\__stridi_if_anchor:nnTF`.)

`\__stridi_side_ports:nnN` `\__stridi_port_of_box:nn` `\__stridi_no_anchor:nn`

`\__stridi_side_ports:nnN` counts the ports of a side of a node, none unless it is a box, from the counts saved in the node. `\__stridi_port_of_box:nn` splits the anchor of a box into side and number, in `\l__stridi_port_seq`, if it is a port, and leaves the sequence empty otherwise. `\__stridi_no_anchor:nn` reports an anchor that a node doesn't have, as a missing port for a box asked for a port beyond its count, and pgf would otherwise fail to read it, with an error about a mathematical function.

```
\int_new:N \l__stridi_ports_int
\int_new:N \g__stridi_ports_int
\cs_new_protected:Nn \__stridi_side_ports:nnN {
  \int_zero:N #3
  \str_if_eq:eeT { \use:c { pgf@sh@ns@ #1 } } { box }
    {
      \group_begin:
        \use:c { pgf@sh@ma@ #1 }
        \int_gset:Nn \g__stridi_ports_int { \use:c { ports #2 } }
      \group_end:
      \int_set_eq:NN #3 \g__stridi_ports_int
    }
}
\msg_new:nnn { string-diagrams } { no-anchor }
  { Node~‘#1’~has~no~anchor~‘#2’,~so~the~wire~is~not~drawn. }
\seq_new:N \l__stridi_port_seq
\cs_new_protected:Nn \__stridi_port_of_box:nn {
  \seq_clear:N \l__stridi_port_seq
  \str_if_eq:eeT { \use:c { pgf@sh@ns@ #1 } } { box }
    {
      \regex_extract_once:nnN { \A (north|east|south|west) \. ([0-9]+) \Z }
        { #2 } \l__stridi_port_seq
    }
}
\cs_new_protected:Nn \__stridi_no_anchor:nn {
  \__stridi_port_of_box:nn { #1 } { #2 }
  \seq_if_empty:NTF \l__stridi_port_seq
    { \msg_error:nnnn { string-diagrams } { no-anchor } { #1 } { #2 } }
    {
      \__stridi_side_ports:neN { #1 } { \seq_item:Nn \l__stridi_port_seq { 2 } } \l__stridi_
      \msg_error:nnneeV { string-diagrams } { missing-port } { Box~‘#1’ }
        { \seq_item:Nn \l__stridi_port_seq { 2 } } { \seq_item:Nn \l__stridi_port_seq { 3 }
        \l__stridi_ports_int
    }
}
\cs_generate_variant:Nn \__stridi_side_ports:nnN { ne }
```

```
\cs_generate_variant:Nn \msg_error:nnnnnn { nnneeV }
\cs_generate_variant:Nn \__stridi_no_anchor:nn { VV }
```

(*End of definition for* `\__stridi_side_ports:nnN`, `\__stridi_port_of_box:nn`, *and* `\__stridi_no_anchor:nn`.)

`\__stridi_wire_end:nNNNN`
`\__stridi_wire_end_side:NNN`

Finds the node and anchor of an end of a to path; the anchor is empty for a bare node. An anchor must be on a side, so that the wire has an angle to leave or enter it at; otherwise the end raises an error, whatever the other end. Ti*k*Z gives an end as the text that was written, such as `g.south.1`, or as a point: the start always, unless it is a bare node, and the target when it is computed, as in `++(0,1)` or `([xshift=1em]g.south)`. A point is looked up among the anchors resolved in the picture. An end that is no anchor of a node, or a boundary from another picture, raises an error. An end on a node that doesn't exist, or at an anchor its node doesn't have, raises an error only if Ti*k*Z hasn't resolved the end, and so reported it, already: then it also sets `\l__stridi_end_missing_bool`, so that the end isn't resolved later. Ti*k*Z resolves the start of a to path while reading the path, and a target only if it computes it.

`#1` : end, as Ti*k*Z gives it
`#2` : token list variable to store the node
`#3` : token list variable to store the anchor
`#4` : boolean variable, set true if the end is fine
`#5` : boolean, true if Ti*k*Z has already resolved the end

```
\msg_new:nnn { string-diagrams } { end-not-anchor }
  {
    Wire~end~`(#1)'~is~not~an~anchor~of~a~node,~so~the~wire~is~not~drawn:~
    wires~go~between~anchors,~as~in~(f.north)~to[wire]~(g.south).
  }
\msg_new:nnn { string-diagrams } { end-no-node }
  { Wire~end~`#1'~is~on~a~node~that~doesn't~exist,~so~the~wire~is~not~drawn. }
\msg_new:nnn { string-diagrams } { end-off-sides }
  {
    Wire~end~`#1.#2'~is~not~on~a~side,~so~the~wire~is~not~drawn:~
    wires~end~at~north,~east,~south~or~west,~or~at~ports.
  }
\msg_new:nnn { string-diagrams } { end-boundary-elsewhere }
  {
    Wire~end~`@.#1'~is~on~the~boundary~of~another~picture,~so~the~wire~is~not~drawn:~
    \iow_char:N\\wires~draws~the~boundary~of~this~picture.
  }
\bool_new:N \l__stridi_end_missing_bool
\bool_new:N \l__stridi_end_resolved_bool
\cs_new_protected:Nn \__stridi_wire_end:nNNNN {
  \bool_set_false:N #4
  \bool_set_eq:NN \l__stridi_end_resolved_bool #5
  \regex_match:nnTF { \A [^,]* pt , [^,]* pt \Z } { #1 }
    {
      \bool_set_true:N \l__stridi_end_resolved_bool
      \__stridi_find_anchor:NNw #2 #3 #1 \q_stop
    }
    { \__stridi_split_node_anchor:NNn #2 #3 { #1 } }
  \tl_if_empty:NTF #2
```

```
    { \msg_error:nnn { string-diagrams } { end-not-anchor } { #1 } }
    {
      \cs_if_exist:cTF { pgf@sh@ns@ #2 }
        {
          \bool_lazy_and:nnTF
            { \str_if_eq_p:Vn #2 { @ } }
            { ! \__stridi_if_boundary_in_picture_p: }
            { \msg_error:nnV { string-diagrams } { end-boundary-elsewhere } #3 }
            {
              \tl_if_empty:NTF #3
                { \bool_set_true:N #4 }
                {
                  \__stridi_if_anchor:VVTF #2 #3
                    { \__stridi_wire_end_side:NNN #2 #3 #4 }
                    {
                      \bool_if:NF \l__stridi_end_resolved_bool
                        {
                          \__stridi_no_anchor:VV #2 #3
                          \bool_set_true:N \l__stridi_end_missing_bool
                        }
                    }
                }
            }
        }
        {
          \bool_if:NF \l__stridi_end_resolved_bool
            {
              \regex_match:nnTF { [,:] } { #1 }
                { \msg_error:nnn { string-diagrams } { end-not-anchor } { #1 } }
                {
                  \msg_error:nnn { string-diagrams } { end-no-node } { #1 }
                  \bool_set_true:N \l__stridi_end_missing_bool
                }
            }
        }
    }
}
\tl_new:N \l__stridi_end_angle_tl
\cs_new_protected:Nn \__stridi_wire_end_side:NNN {
  \__stridi_anchor_to_angle:NV \l__stridi_end_angle_tl #2
  \tl_if_empty:NTF \l__stridi_end_angle_tl
    { \msg_error:nnVV { string-diagrams } { end-off-sides } #1 #2 }
    { \bool_set_true:N #3 }
}
\cs_generate_variant:Nn \__stridi_wire_end:nNNNN { V }
\cs_generate_variant:Nn \__stridi_anchor_to_angle:Nn { NV }
```

(*End of definition for* `\__stridi_wire_end:nNNNN` *and* `\__stridi_wire_end_side:NNN`.)

`\l__stridi_start_node_tl`
`\l__stridi_start_anchor_tl`
`\l__stridi_end_node_tl`
`\l__stridi_end_anchor_tl`
`\l__stridi_start_ok_bool`
`\l__stridi_end_ok_bool`
`\l__stridi_start_x_dim`
`\l__stridi_start_y_dim`
`\l__stridi_end_x_dim`
`\l__stridi_end_y_dim`
`\l__stridi_start_in_bool`
`\l__stridi_end_in_bool`

Scratch variables for `\__stridi_wire:`: the two ends of a wire, whether each is fine, their points, and whether each faces inwards.

```
\tl_new:N \l__stridi_start_node_tl
\tl_new:N \l__stridi_start_anchor_tl
```

```
\tl_new:N \l__stridi_end_node_tl
\tl_new:N \l__stridi_end_anchor_tl
\bool_new:N \l__stridi_start_ok_bool
\bool_new:N \l__stridi_end_ok_bool
\dim_new:N \l__stridi_start_x_dim
\dim_new:N \l__stridi_start_y_dim
\dim_new:N \l__stridi_end_x_dim
\dim_new:N \l__stridi_end_y_dim
\bool_new:N \l__stridi_start_in_bool
\bool_new:N \l__stridi_end_in_bool
```

(*End of definition for* `\l__stridi_start_node_tl` *and others.*)

`\__stridi_point_of_end:nnNN` Stores the point of an end: its anchor, or the center of a bare node.

`#1` : node
`#2` : anchor, or nothing
`#3` : dimension variable to store the x coordinate
`#4` : dimension variable to store the y coordinate

```
\cs_new_protected:Nn \__stridi_point_of_end:nnNN {
  \tl_if_empty:nTF { #2 }
    { \pgfpointanchor { #1 } { center } }
    { \pgfpointanchor { #1 } { #2 } }
  \dim_set:Nn #3 { \pgf@x }
  \dim_set:Nn #4 { \pgf@y }
}
\cs_generate_variant:Nn \__stridi_point_of_end:nnNN { VVNN }
```

(*End of definition for* `\__stridi_point_of_end:nnNN`.)

`\l__stridi_start_west_dim` `\l__stridi_start_south_dim` `\l__stridi_start_east_dim` `\l__stridi_start_north_dim` `\l__stridi_end_west_dim` `\l__stridi_end_south_dim` `\l__stridi_end_east_dim` `\l__stridi_end_north_dim` `\l__stridi_start_holds_bool` `\l__stridi_end_holds_bool` Scratch variables for `\__stridi_faces_inwards:`: the bounding boxes of the nodes of the two ends of a wire, and whether each holds the other.

```
\dim_new:N \l__stridi_start_west_dim
\dim_new:N \l__stridi_start_south_dim
\dim_new:N \l__stridi_start_east_dim
\dim_new:N \l__stridi_start_north_dim
\dim_new:N \l__stridi_end_west_dim
\dim_new:N \l__stridi_end_south_dim
\dim_new:N \l__stridi_end_east_dim
\dim_new:N \l__stridi_end_north_dim
\bool_new:N \l__stridi_start_holds_bool
\bool_new:N \l__stridi_end_holds_bool
```

(*End of definition for* `\l__stridi_start_west_dim` *and others.*)

`\__stridi_node_box:nNNNN` Stores the bounding box of a node, from its south west and north east anchors, which lie on the outer edge of its stroke, since Ti*k*Z's default `outer sep` is half the line width.

`#1` : node
`#2` : dimension variable to store the west border
`#3` : dimension variable to store the south border
`#4` : dimension variable to store the east border

`#5` : dimension variable to store the north border

```
\cs_new_protected:Nn \__stridi_node_box:nNNNN {
  \pgfpointanchor { #1 } { south~west }
  \dim_set:Nn #2 { \pgf@x }
  \dim_set:Nn #3 { \pgf@y }
  \pgfpointanchor { #1 } { north~east }
  \dim_set:Nn #4 { \pgf@x }
  \dim_set:Nn #5 { \pgf@y }
}
\cs_generate_variant:Nn \__stridi_node_box:nNNNN { V }
```

(*End of definition for* `\__stridi_node_box:nNNNN`.)

`\__stridi_if_box_within_p:nn` `\__stridi_if_boxes_apart_p:`

`\__stridi_if_box_within_p:nn` tests whether the box of one end of a wire, `start` or `end`, lies within the box of the other, a shared border counting as inside within 0.01pt. `\__stridi_if_boxes_apart_p:` tests whether the two boxes don't overlap, borders that touch included.

```
\prg_new_conditional:Nnn \__stridi_if_box_within:nn { p }
  {
    \bool_lazy_all:nTF
      {
        { \dim_compare_p:n { \use:c { l__stridi_#2_west_dim } <= \use:c { l__stridi_#1_west_
        { \dim_compare_p:n { \use:c { l__stridi_#2_south_dim } <= \use:c { l__stridi_#1_sout
        { \dim_compare_p:n { \use:c { l__stridi_#2_east_dim } >= \use:c { l__stridi_#1_east_
        { \dim_compare_p:n { \use:c { l__stridi_#2_north_dim } >= \use:c { l__stridi_#1_nort
      }
      { \prg_return_true: }
      { \prg_return_false: }
  }
\prg_new_conditional:Nnn \__stridi_if_boxes_apart: { p }
  {
    \bool_lazy_any:nTF
      {
        { \dim_compare_p:n { \l__stridi_start_east_dim <= \l__stridi_end_west_dim + 0.01pt }
        { \dim_compare_p:n { \l__stridi_end_east_dim <= \l__stridi_start_west_dim + 0.01pt }
        { \dim_compare_p:n { \l__stridi_start_north_dim <= \l__stridi_end_south_dim + 0.01pt
        { \dim_compare_p:n { \l__stridi_end_north_dim <= \l__stridi_start_south_dim + 0.01pt
      }
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End of definition for* `\__stridi_if_box_within_p:nn` *and* `\__stridi_if_boxes_apart_p:`.)

`\__stridi_faces_inwards:`

Tells which ends of a wire face inwards, from the boxes of their nodes. The end on a node that holds the other faces inwards. Otherwise both ends face out: the nodes are apart, or they overlap, or have the same box, and then a bare end is an error, since neither node is the outer one, and the wire is not drawn. A wire from a node to itself faces inwards at both ends on the boundary `@`, which it crosses, and outwards on any other node, which it loops round.

```
\msg_new:nnn { string-diagrams } { bare-overlap }
  {
    Wire~from~`#1'~to~`#2'~has~an~end~without~anchor,~and~the~two~nodes~overlap,~
    so~it~is~not~drawn:~give~both~ends~anchors.
  }
\cs_new_protected:Nn \__stridi_faces_inwards: {
  \bool_set_false:N \l__stridi_start_in_bool
  \bool_set_false:N \l__stridi_end_in_bool
  \str_if_eq:VVTF \l__stridi_start_node_tl \l__stridi_end_node_tl
    {
      \str_if_eq:VnT \l__stridi_start_node_tl { @ }
        {
          \bool_set_true:N \l__stridi_start_in_bool
          \bool_set_true:N \l__stridi_end_in_bool
        }
    }
    {
      \__stridi_node_box:VNNNN \l__stridi_start_node_tl
        \l__stridi_start_west_dim \l__stridi_start_south_dim \l__stridi_start_east_dim \l__s
      \__stridi_node_box:VNNNN \l__stridi_end_node_tl
        \l__stridi_end_west_dim \l__stridi_end_south_dim \l__stridi_end_east_dim \l__stridi_
      \bool_set:Nn \l__stridi_start_holds_bool { \__stridi_if_box_within_p:nn { end } { star
      \bool_set:Nn \l__stridi_end_holds_bool { \__stridi_if_box_within_p:nn { start } { end
      \bool_if:nTF { \l__stridi_start_holds_bool && ! \l__stridi_end_holds_bool }
        { \bool_set_true:N \l__stridi_start_in_bool }
        {
          \bool_if:nTF { \l__stridi_end_holds_bool && ! \l__stridi_start_holds_bool }
            { \bool_set_true:N \l__stridi_end_in_bool }
            {
              \bool_lazy_or:nnF
                { \__stridi_if_boxes_apart_p: }
                { ! \tl_if_empty_p:N \l__stridi_start_anchor_tl && ! \tl_if_empty_p:N \l__st
                {
                  \msg_error:nnVV { string-diagrams } { bare-overlap }
                    \l__stridi_start_node_tl \l__stridi_end_node_tl
                  \bool_set_false:N \l__stridi_start_ok_bool
                }
            }
        }
    }
}
```

(*End of definition for* `\__stridi_faces_inwards:`.)

`\l__stridi_flow_tl` The flow of the wires, set by `wire flow`: a side, or nothing.

```
\tl_new:N \l__stridi_flow_tl
```

(*End of definition for* `\l__stridi_flow_tl`.)

`\__stridi_set_flow:n` Sets the flow of the wires: `north`, `east`, `south` or `west`, or `up`, `right`, `down` or `left`, as Ti*k*Z's `grow` takes, or `none` to have none.

```
\msg_new:nnn { string-diagrams } { flow-value }
```

```
  {
    Wire~flow~`#1'~is~not~a~direction:~
    use~north,~east,~south,~west,~up,~right,~down,~left~or~none.
  }
\cs_new_protected:Nn \__stridi_set_flow:n {
  \str_case:nnF { #1 }
    {
      { north } { \tl_set:Nn \l__stridi_flow_tl { north } }
      { east } { \tl_set:Nn \l__stridi_flow_tl { east } }
      { south } { \tl_set:Nn \l__stridi_flow_tl { south } }
      { west } { \tl_set:Nn \l__stridi_flow_tl { west } }
      { up } { \tl_set:Nn \l__stridi_flow_tl { north } }
      { right } { \tl_set:Nn \l__stridi_flow_tl { east } }
      { down } { \tl_set:Nn \l__stridi_flow_tl { south } }
      { left } { \tl_set:Nn \l__stridi_flow_tl { west } }
      { none } { \tl_clear:N \l__stridi_flow_tl }
    }
    { \msg_error:nnn { string-diagrams } { flow-value } { #1 } }
}
```

(*End of definition for* `\__stridi_set_flow:n`.)

`\l__stridi_dir_int` `\l__stridi_side_tl` Scratch variables for the inference of anchors: the direction of travel of a wire, in degrees, and a side.

```
\int_new:N \l__stridi_dir_int
\tl_new:N \l__stridi_side_tl
```

(*End of definition for* `\l__stridi_dir_int` *and* `\l__stridi_side_tl`.)

`\__stridi_side_of:n` `\__stridi_opposite:n` `\__stridi_side_anchor:NNnn` Turn directions into sides and sides into anchors. `\__stridi_side_of:n` expands to the side a direction, a multiple of 90 degrees, points to, and `\__stridi_opposite:n` to the side opposite a side. `\__stridi_side_anchor:NNnn` gives the anchor of a bare end on a side: the side without ports, its only port, and an error asking which if it has more, which leaves the anchor empty and stops the wire.

```
\cs_new:Nn \__stridi_side_of:n
  {
    \int_case:nn { \int_mod:nn { #1 + 720 } { 360 } }
      { { 0 } { east } { 90 } { north } { 180 } { west } { 270 } { south } }
  }
\cs_new:Nn \__stridi_opposite:n
  { \str_case:nn { #1 } { { north } { south } { south } { north } { east } { west } { west }
\msg_new:nnn { string-diagrams } { bare-many-ports }
  {
    Wire~end~`#1'~has~no~anchor,~and~its~#2~side~has~#3~ports,~so~the~wire~is~not~drawn:~
    name~one,~as~in~#1.#2.1.
  }
\cs_new_protected:Nn \__stridi_side_anchor:NNnn {
  \__stridi_side_ports:nnN { #3 } { #4 } \l__stridi_ports_int
  \int_case:nnF { \l__stridi_ports_int }
    {
      { 0 } { \tl_set:Nn #1 { #4 } }
      { 1 } { \tl_set:Nn #1 { #4.1 } }
```

```
    }
    {
      \msg_error:nnnnV { string-diagrams } { bare-many-ports } { #3 } { #4 } \l__stridi_port
      \bool_set_false:N #2
    }
}
\cs_generate_variant:Nn \__stridi_side_anchor:NNnn { NNVe }
\cs_generate_variant:Nn \msg_error:nnnnn { nnnnV }
```

(*End of definition for* `\__stridi_side_of:n`, `\__stridi_opposite:n`, *and* `\__stridi_side_anchor:NNnn`.)

`\__stridi_snap:`
`\__stridi_both_bare_direction:`
`\__stridi_infer_anchors:`

Infer the anchors of bare ends from the direction a wire travels in. With one end anchored, the wire travels as it leaves that end, or as it enters it; a diagonal, from a corner, is snapped to the axis of the flow, if set, else to the axis that separates the two ends more. With both ends bare, it travels along the flow, if set, else along the axis that separates the centers of the two nodes more, and nodes with the same center are an error, as is a wire from a node to itself. Then a bare start is the side the wire leaves through, and a bare target the side it enters through; an end that faces inwards uses the opposite side, so that a wire from a node around the other starts behind it, and a wire to that node ends ahead of it.

```
\msg_new:nnn { string-diagrams } { bare-no-direction }
  {
    Wire~from~‘#1’~to~‘#2’~has~no~anchors,~and~their~centers~coincide,~so~it~is~not~drawn:~
    give~an~end~an~anchor,~or~set~a~wire~flow.
  }
\msg_new:nnn { string-diagrams } { bare-self-no-direction }
  {
    Wire~from~‘#1’~to~itself~has~no~anchors,~so~it~is~not~drawn:~
    give~an~end~an~anchor,~or~set~a~wire~flow.
  }
\cs_new_protected:Nn \__stridi_snap: {
  \int_compare:nNnF { \int_mod:nn { \l__stridi_dir_int + 720 } { 90 } } = { 0 }
    {
      \bool_if:nTF
        {
          \tl_if_empty_p:N \l__stridi_flow_tl
          && \dim_compare_p:n
            {
              \dim_abs:n { \l__stridi_end_x_dim - \l__stridi_start_x_dim }
              >= \dim_abs:n { \l__stridi_end_y_dim - \l__stridi_start_y_dim }
            }
          || \str_if_eq_p:Vn \l__stridi_flow_tl { east }
          || \str_if_eq_p:Vn \l__stridi_flow_tl { west }
        }
        { \int_set:Nn \l__stridi_dir_int { \fp_eval:n { cosd ( \l__stridi_dir_int ) > 0 ? 0
        { \int_set:Nn \l__stridi_dir_int { \fp_eval:n { sind ( \l__stridi_dir_int ) > 0 ? 90
    }
}
\cs_new_protected:Nn \__stridi_both_bare_direction: {
  \tl_if_empty:NTF \l__stridi_flow_tl
    {
      \bool_if:nTF
```

```
        {
          \dim_compare_p:n { \dim_abs:n { \l__stridi_end_x_dim - \l__stridi_start_x_dim } <
          && \dim_compare_p:n { \dim_abs:n { \l__stridi_end_y_dim - \l__stridi_start_y_dim }
        }
        {
          \str_if_eq:VVTF \l__stridi_start_node_tl \l__stridi_end_node_tl
            { \msg_error:nnV { string-diagrams } { bare-self-no-direction } \l__stridi_start
            {
              \msg_error:nnVV { string-diagrams } { bare-no-direction }
                \l__stridi_start_node_tl \l__stridi_end_node_tl
            }
          \bool_set_false:N \l__stridi_start_ok_bool
        }
        {
          \dim_compare:nNnTF
            { \dim_abs:n { \l__stridi_end_x_dim - \l__stridi_start_x_dim } }
            >
            { \dim_abs:n { \l__stridi_end_y_dim - \l__stridi_start_y_dim } }
            {
              \dim_compare:nNnTF { \l__stridi_end_x_dim } > { \l__stridi_start_x_dim }
                { \int_set:Nn \l__stridi_dir_int { 0 } }
                { \int_set:Nn \l__stridi_dir_int { 180 } }
            }
            {
              \dim_compare:nNnTF { \l__stridi_end_y_dim } > { \l__stridi_start_y_dim }
                { \int_set:Nn \l__stridi_dir_int { 90 } }
                { \int_set:Nn \l__stridi_dir_int { 270 } }
            }
        }
    }
    {
      \int_set:Nn \l__stridi_dir_int
        { \str_case:Vn \l__stridi_flow_tl { { east } { 0 } { north } { 90 } { west } { 180 }
    }
}
\cs_new_protected:Nn \__stridi_infer_anchors: {
  \tl_if_empty:NTF \l__stridi_start_anchor_tl
    {
      \tl_if_empty:NTF \l__stridi_end_anchor_tl
        { \__stridi_both_bare_direction: }
        {
          \int_set:Nn \l__stridi_dir_int
            { \l__stridi_iangle_tl \bool_if:NF \l__stridi_end_in_bool { + 180 } }
        }
    }
    {
      \int_set:Nn \l__stridi_dir_int
        { \l__stridi_oangle_tl \bool_if:NT \l__stridi_start_in_bool { + 180 } }
    }
  \bool_if:NT \l__stridi_start_ok_bool
    {
      \__stridi_snap:
      \tl_if_empty:NT \l__stridi_start_anchor_tl
        {
```

```
          \tl_set:Ne \l__stridi_side_tl { \__stridi_side_of:n { \l__stridi_dir_int } }
          \bool_if:NT \l__stridi_start_in_bool
            { \tl_set:Ne \l__stridi_side_tl { \__stridi_opposite:V \l__stridi_side_tl } }
          \__stridi_side_anchor:NNVV \l__stridi_start_anchor_tl \l__stridi_start_ok_bool
            \l__stridi_start_node_tl \l__stridi_side_tl
        }
      \tl_if_empty:NT \l__stridi_end_anchor_tl
        {
          \tl_set:Ne \l__stridi_side_tl { \__stridi_side_of:n { \l__stridi_dir_int } }
          \bool_if:NF \l__stridi_end_in_bool
            { \tl_set:Ne \l__stridi_side_tl { \__stridi_opposite:V \l__stridi_side_tl } }
          \__stridi_side_anchor:NNVV \l__stridi_end_anchor_tl \l__stridi_end_ok_bool
            \l__stridi_end_node_tl \l__stridi_side_tl
        }
    }
}
\cs_generate_variant:Nn \__stridi_opposite:n { V }
\cs_generate_variant:Nn \__stridi_side_anchor:NNnn { NNVV }
\cs_generate_variant:Nn \str_case:nn { V }
```

(*End of definition for* `\__stridi_snap:`, `\__stridi_both_bare_direction:`, *and* `\__stridi_infer_anchors:`.)

`\__stridi_end_coordinate:NnnnnN` Calculates where a wire ends at an anchor: the anchor itself, unless the end faces inwards and the anchor is a side without a port, which the wire reaches straight across from the other end, as `\__stridi_calctarget:nnnn` finds. The result is stored in the given token list variable, as a point or as ⟨*node*⟩.⟨*anchor*⟩.

`#1` : token list variable
`#2` : node of the end
`#3` : anchor of the end
`#4` : node of the other end
`#5` : anchor of the other end
`#6` : boolean, true if the end faces inwards

```
\cs_new_protected:Nn \__stridi_end_coordinate:NnnnnN {
  \bool_lazy_and:nnTF { #6 } { \__stridi_if_side_name_p:n { #3 } }
    {
      \__stridi_calctarget:nnnn { #4 } { #5 } { #2 } { #3 }
      \tl_set:Ne #1 { \dim_use:N \pgf@x , \dim_use:N \pgf@y }
    }
    { \tl_set:Nn #1 { #2.#3 } }
}
\cs_generate_variant:Nn \__stridi_end_coordinate:NnnnnN { NVVVVN }
\prg_new_conditional:Nnn \__stridi_if_side_name:n { p }
  {
    \str_case:nnF { #1 }
      {
        { north } { \prg_return_true: } { east } { \prg_return_true: }
        { south } { \prg_return_true: } { west } { \prg_return_true: }
      }
      { \prg_return_false: }
  }
```

(*End of definition for* `\__stridi_end_coordinate:NnnnnN`.)

`\l__stridi_wire_bool` `\l__stridi_wire_drawn_bool` `\l__stridi_start_moved_bool` `\l__stridi_given_target_tl` `\s__stridi_auto`

Scratch variables for the `wire` key: whether the to path at hand is a wire, whether it is drawn, whether its start moves, and its target as Ti*k*Z gave it; and the scan mark that stands for the angles the key computes.

```
\bool_new:N \l__stridi_wire_bool
\bool_new:N \l__stridi_wire_drawn_bool
\bool_new:N \l__stridi_start_moved_bool
\tl_new:N \l__stridi_given_target_tl
\scan_new:N \s__stridi_auto
```

(*End of definition for* `\l__stridi_wire_bool` *and others.*)

`\__stridi_wire:` `\__stridi_wire_ends:` `\__stridi_wire_path:`

Turn a to path into a wire, just before Ti*k*Z computes its curve, after all the keys of the path, so that `wire flow` counts wherever it is set. Ti*k*Z computes a curve bent by `bend left` or `bend right` apart, from angles relative to the line between its ends, so both computations are hooked: the ends are resolved alike, and a bent wire keeps its angles. `\__stridi_wire:` finds the ends, infers the anchors of bare ends, and sets the angles that Ti*k*Z's own `out` and `in` draw the curve with, unless keys after `wire` have set them. An end faces inwards when its node holds the other, as `\__stridi_faces_inwards:` tells: then its angle turns round, and a side without a port is reached straight across, as on the boundary. An end that becomes a point, or a bare start, moves the start, which `\__stridi_wire_path:` does after the computation; it also turns a wire that isn't drawn into a move to its target, or into nothing if its target is on a node that doesn't exist, which Ti*k*Z would report again, and drops its labels, the nodes Ti*k*Z would place along it. Each wrong end raises an error.

```
\cs_new_protected:Nn \__stridi_wire: {
  \bool_set_false:N \l__stridi_end_missing_bool
  \bool_set_false:N \l__stridi_start_moved_bool
  \tl_set_eq:NN \l__stridi_given_target_tl \tikztotarget
  \__stridi_wire_end:VNNNN \tikztostart
    \l__stridi_start_node_tl \l__stridi_start_anchor_tl \l__stridi_start_ok_bool \c_true_boo
  \__stridi_wire_end:VNNNN \tikztotarget
    \l__stridi_end_node_tl \l__stridi_end_anchor_tl \l__stridi_end_ok_bool \c_false_bool
  \bool_lazy_and:nnT { \l__stridi_start_ok_bool } { \l__stridi_end_ok_bool } { \__stridi_wir
  \bool_lazy_and:nnTF { \l__stridi_start_ok_bool } { \l__stridi_end_ok_bool }
    {
      \bool_set_true:N \l__stridi_wire_drawn_bool
      \tl_set_eq:NN \tikztotarget \l__stridi_target_tl
      \str_if_eq:eeT { \exp_not:o { \tikz@to@out } } { \exp_not:N \s__stridi_auto }
        { \cs_set:Npe \tikz@to@out { \tl_use:N \l__stridi_oangle_tl } }
      \str_if_eq:eeT { \exp_not:o { \tikz@to@in } } { \exp_not:N \s__stridi_auto }
        { \cs_set:Npe \tikz@to@in { \tl_use:N \l__stridi_iangle_tl } }
      \bool_if:NT \l__stridi_start_moved_bool
        { \tl_set_eq:NN \tikztostart \l__stridi_source_tl }
    }
    {
      \bool_set_false:N \l__stridi_wire_drawn_bool
      \cs_set:Npn \tikz@to@out { 0 }
      \cs_set:Npn \tikz@to@in { 180 }
      \tl_set:Nn \tikztostart { 0pt , 0pt }
```

```
      \tl_set:Nn \tikztotarget { 0pt , 0pt }
    }
}
\cs_new_protected:Nn \__stridi_wire_ends: {
  \__stridi_point_of_end:VVNN \l__stridi_start_node_tl \l__stridi_start_anchor_tl
    \l__stridi_start_x_dim \l__stridi_start_y_dim
  \__stridi_point_of_end:VVNN \l__stridi_end_node_tl \l__stridi_end_anchor_tl
    \l__stridi_end_x_dim \l__stridi_end_y_dim
  \__stridi_faces_inwards:
  \bool_set:Nn \l__stridi_start_moved_bool { \tl_if_empty_p:N \l__stridi_start_anchor_tl }
  \__stridi_anchor_to_angle:NV \l__stridi_oangle_tl \l__stridi_start_anchor_tl
  \__stridi_anchor_to_angle:NV \l__stridi_iangle_tl \l__stridi_end_anchor_tl
  \bool_lazy_and:nnT
    { \l__stridi_start_ok_bool }
    { \tl_if_empty_p:N \l__stridi_start_anchor_tl || \tl_if_empty_p:N \l__stridi_end_anchor_
    {
      \__stridi_infer_anchors:
      \__stridi_anchor_to_angle:NV \l__stridi_oangle_tl \l__stridi_start_anchor_tl
      \__stridi_anchor_to_angle:NV \l__stridi_iangle_tl \l__stridi_end_anchor_tl
    }
  \bool_lazy_and:nnT { \l__stridi_start_ok_bool } { \l__stridi_end_ok_bool }
    {
      \bool_if:NT \l__stridi_start_in_bool
        { \tl_set:Ne \l__stridi_oangle_tl { \int_eval:n { \l__stridi_oangle_tl - 180 } } }
      \bool_if:NT \l__stridi_end_in_bool
        { \tl_set:Ne \l__stridi_iangle_tl { \int_eval:n { \l__stridi_iangle_tl - 180 } } }
      \__stridi_end_coordinate:NVVVVN \l__stridi_source_tl
        \l__stridi_start_node_tl \l__stridi_start_anchor_tl
        \l__stridi_end_node_tl \l__stridi_end_anchor_tl \l__stridi_start_in_bool
      \__stridi_end_coordinate:NVVVVN \l__stridi_target_tl
        \l__stridi_end_node_tl \l__stridi_end_anchor_tl
        \l__stridi_start_node_tl \l__stridi_start_anchor_tl \l__stridi_end_in_bool
      \bool_lazy_and:nnT { \l__stridi_start_in_bool } { \__stridi_if_side_name_p:V \l__strid
        { \bool_set_true:N \l__stridi_start_moved_bool }
    }
}
\cs_new_protected:Nn \__stridi_wire_path: {
  \bool_if:NTF \l__stridi_wire_drawn_bool
    {
      \bool_if:NT \l__stridi_start_moved_bool
        { \tl_put_left:Ne \tikz@computed@path { ( \exp_not:V \tikztostart ) ~ } }
    }
    {
      \cs_set:Npn \tikztonodes { }
      \bool_if:NTF \l__stridi_end_missing_bool
        { \tl_clear:N \tikz@computed@path }
        { \tl_set:Ne \tikz@computed@path { ( \exp_not:V \l__stridi_given_target_tl ) } }
    }
}
\prg_generate_conditional_variant:Nnn \__stridi_if_side_name:n { V } { p }
\AddToHook { cmd/tikz@to@compute/before }
  { \bool_if:NT \l__stridi_wire_bool { \__stridi_wire: } }
\AddToHook { cmd/tikz@to@compute/after }
  { \bool_if:NT \l__stridi_wire_bool { \__stridi_wire_path: } }
```

```
\AddToHook { cmd/tikz@to@compute@relative/before }
  { \bool_if:NT \l__stridi_wire_bool { \__stridi_wire: } }
\AddToHook { cmd/tikz@to@compute@relative/after }
  { \bool_if:NT \l__stridi_wire_bool { \__stridi_wire_path: } }
```

(*End of definition for* `\__stridi_wire:`, `\__stridi_wire_ends:`, *and* `\__stridi_wire_path:`.)

`/tikz/wire`
`/tikz/every␣wire`
`/tikz/wire␣flow`

Defines the key that draws a to path or an edge as a wire, the style it applies after it, as `every edge` or `every to` would, and the key that sets the flow of the wires. The `wire` key only marks the path as a wire, draws it as a curve, and leaves its angles to be computed, unless keys after it set them; it undoes a `bend left` or `bend right` before it, as it overrides `out` and `in`.

```
\cs_new_protected:Nn \__stridi_wire_key: {
  \bool_set_true:N \l__stridi_wire_bool
  \cs_set:Npn \tikz@to@out { \s__stridi_auto }
  \cs_set:Npn \tikz@to@in { \s__stridi_auto }
  \tikz@to@relativefalse
  \tikz@to@switch@on
}
\tikzset{
  every~wire/.style = { },
  wire/.code = { \__stridi_wire_key: \pgfkeysalso { every~wire } },
  wire~flow/.code = { \__stridi_set_flow:n { #1 } },
}
```

(*End of definition for* `/tikz/wire`, `/tikz/every wire`, *and* `/tikz/wire flow`. *These functions are documented on page* 6.)

`\__stridi_compass:n`
`\__stridi_side_turn:nn`

`\__stridi_compass:n` expands to the compass anchor of a direction, in degrees, a multiple of 45. `\__stridi_side_turn:nn` expands to its first argument for a label to the left of a path, and to its second for one to the right, as `swap` sets.

```
\cs_new:Nn \__stridi_compass:n
  {
    \int_case:nn { \int_mod:nn { #1 + 720 } { 360 } }
      {
        { 0 } { east } { 45 } { north~east } { 90 } { north } { 135 } { north~west }
        { 180 } { west } { 225 } { south~west } { 270 } { south } { 315 } { south~east }
      }
  }
\cs_new:Nn \__stridi_side_turn:nn
  { \str_if_eq:VnTF \tikz@auto@anchor@direction { left } { #1 } { #2 } }
```

(*End of definition for* `\__stridi_compass:n` *and* `\__stridi_side_turn:nn`.)

`\l__stridi_label_fp`
`\l__stridi_label_angle_tl`

Scratch variables for the placement of labels: a direction of travel, and the angle of the anchor a label belongs to.

```
\fp_new:N \l__stridi_label_fp
\tl_new:N \l__stridi_label_angle_tl
```

(*End of definition for* `\l__stridi_label_fp` *and* `\l__stridi_label_angle_tl`.)

__stridi_label_key:nnnnnn
__stridi_label_on_path:nnnn

Choose the anchor of a label, as Ti*k*Z's `auto` does, but turned from the direction of travel by the given angles, the first of each pair to the left and the second to the right. For a label on a path, the direction is the tangent of the path at the label, which Ti*k*Z reads when it places the label, as for `auto`, and the turn depends on which end of the path the label is nearer. For a label of a node, given with `label`, the direction is that of a wire leaving the anchor the label belongs to, from the anchor's name; an anchor without an angle, such as `center`, leaves the placement to Ti*k*Z. The label is then placed at the anchor itself, and its own anchor is a macro, expanded when the node is made. In both cases the side is read once all the keys of the label are set, so a `swap` after the key counts.

`#1` : turn near the start of a path, to the left
`#2` : turn near the start of a path, to the right
`#3` : turn near the end of a path, to the left
`#4` : turn near the end of a path, to the right
`#5` : turn for the label of a node, to the left
`#6` : turn for the label of a node, to the right

```
\cs_new_protected:Nn \__stridi_label_key:nnnnnn {
  \cs_set:Npn \tikz@do@auto@anchor
    { \__stridi_label_on_path:nnnn { #1 } { #2 } { #3 } { #4 } }
  \str_if_eq:VnT \tikz@anchor { @auto }
    {
      \__stridi_anchor_to_angle:NV \l__stridi_label_angle_tl \tikz@label@angle
      \tl_if_empty:NF \l__stridi_label_angle_tl
        {
          \pgfkeysalso { label~distance = 0pt }
          \cs_set:Npe \tikz@anchor
            {
              \exp_not:N \__stridi_compass:n
                { \l__stridi_label_angle_tl + \exp_not:N \__stridi_side_turn:nn { #5 } { #6
            }
        }
    }
}
\cs_new_protected:Nn \__stridi_label_on_path:nnnn {
  \tikz@auto@pre
    \fp_set:Nn \l__stridi_label_fp
      { round ( atand ( \dim_to_fp:n { \pgf@y } , \dim_to_fp:n { \pgf@x } ) / 45 ) * 45 }
    \cs_set:Npe \tikz@anchor
      {
        \__stridi_compass:n
          {
            \fp_to_int:N \l__stridi_label_fp
            + \fp_compare:nNnTF { \tikz@time } < { 0.5 }
              { \__stridi_side_turn:nn { #1 } { #2 } }
              { \__stridi_side_turn:nn { #3 } { #4 } }
          }
      }
  \tikz@auto@post
}
```

(*End of definition for* `\__stridi_label_key:nnnnnn` *and* `\__stridi_label_on_path:nnnn`.)

/tikz/beyond
/tikz/beside
/tikz/beyond␣end
/tikz/beyond␣start
/tikz/beside␣end
/tikz/beside␣start

Define the keys that place labels of wires and ports, as Ti*k*Z's `auto` and `swap` do, but past the end of a wire or beside it, clear of the box. On a path, `beyond` faces the label back along the path, past its end, and `beside` turns it 45 degrees further than `auto` does: away from the box the wire enters, near its end, or leaves, near its start. In `label`, they place the label of a port as at the start of a wire leaving it: straight out, or beside the wire. The others add the position, as `at end` and `at start`.

```
\tikzset{
  beyond/.code = { \__stridi_label_key:nnnnnn { 0 } { 0 } { 180 } { 180 } { 180 } { 180 } },
  beside/.code = { \__stridi_label_key:nnnnnn { -135 } { 135 } { -45 } { 45 } { -135 } { 135
  beyond~end/.style = { at~end, beyond },
  beyond~start/.style = { at~start, beyond },
  beside~end/.style = { at~end, beside },
  beside~start/.style = { at~start, beside },
}
```

(*End of definition for* `/tikz/beyond` *and others. These functions are documented on page* 7.)

`\exp_args:Nnnnoo`

This helper is required by `\__stridi_adj_wire:nNnnn`.

```
\exp_args_generate:n { nnnoo }
```

(*End of definition for* `\exp_args:Nnnnoo`.)

`\l__stridi_adj_node_tl`
`\l__stridi_adj_anchor_tl`
`\l__stridi_target_node_tl`
`\l__stridi_target_anchor_tl`

Scratch variables for the parser of `\wires`: the two parts of an end, and of a target.

```
\tl_new:N \l__stridi_adj_node_tl
\tl_new:N \l__stridi_adj_anchor_tl
\tl_new:N \l__stridi_target_node_tl
\tl_new:N \l__stridi_target_anchor_tl
```

(*End of definition for* `\l__stridi_adj_node_tl` *and others.*)

`\s__stridi_keys`
`\s__stridi_stop`

Scan marks for the parser of `\wires`: the first marks the keys of a bracketed group, and the second ends a loop.

```
\scan_new:N \s__stridi_keys
\scan_new:N \s__stridi_stop
```

(*End of definition for* `\s__stridi_keys` *and* `\s__stridi_stop`.)

`\__stridi_protect_brackets:Nn`
`\__stridi_protect_brackets_out:Nw`
`\__stridi_protect_brackets_in:Nw`
`\__stridi_protect_brackets_skip:w`
`\__stridi_before_closing:w`

Replaces each bracketed group that isn't in braces by `\s__stridi_keys` followed by its keys in braces, so that their commas survive the split of a list into items, and they can't be mistaken for a group in braces. As in Ti*k*Z, a group ends at the first closing bracket outside braces. Each piece is read after a quark, so that a piece in braces keeps them. A bracket outside braces without its match ends the list with an error.

`#1` : token list variable to store the result
`#2` : token list

```
\msg_new:nnn { string-diagrams } { unclosed-bracket }
  { A~`[`~in~the~wires~has~no~matching~`]`,~so~the~rest~of~the~list~is~ignored. }
\msg_new:nnn { string-diagrams } { unopened-bracket }
```

```
  { A~']'~in~the~wires~has~no~matching~'[',~so~the~rest~of~the~list~is~ignored. }
\cs_new_protected:Nn \__stridi_protect_brackets:Nn {
  \tl_clear:N #1
  \__stridi_protect_brackets_out:Nw #1 \q_mark #2 [ \q_stop ]
}
\cs_generate_variant:Nn \__stridi_protect_brackets:Nn { NV }
\cs_new_protected:Npn \__stridi_protect_brackets_out:Nw #1#2 [ {
  \tl_if_in:onTF { \use_none:n #2 } { ] }
    {
      \msg_error:nn { string-diagrams } { unopened-bracket }
      \tl_put_right:Ne #1 { \__stridi_before_closing:w #2 \q_stop }
      \__stridi_protect_brackets_skip:w
    }
    {
      \tl_put_right:No #1 { \use_none:n #2 }
      \__stridi_protect_brackets_in:Nw #1 \q_mark
    }
}
\cs_new_protected:Npn \__stridi_protect_brackets_skip:w #1 \q_stop ] { }
\cs_new:Npn \__stridi_before_closing:w #1 ] #2 \q_stop { \exp_not:o { \use_none:n #1 } }
\cs_new_protected:Npn \__stridi_protect_brackets_in:Nw #1#2 ] {
  \tl_if_eq:onF { \use_none:n #2 } { \q_stop }
    {
      \tl_if_in:onTF { \use_none:n #2 } { \q_stop }
        { \msg_error:nn { string-diagrams } { unclosed-bracket } }
        {
          \tl_put_right:Ne #1
            { \exp_not:N \s__stridi_keys { \exp_not:o { \use_none:n #2 } } }
          \__stridi_protect_brackets_out:Nw #1 \q_mark
        }
    }
}
\prg_generate_conditional_variant:Nnn \tl_if_eq:nn { on } { F }
\prg_generate_conditional_variant:Nnn \tl_if_in:nn { on } { TF }
```

(*End of definition for* `\__stridi_protect_brackets:Nn` *and others.*)

`\l__stridi_adj_keys_tl` `\l__stridi_adj_piece_keys_tl` `\l__stridi_adj_rest_tl` `\l__stridi_adj_list_tl`

Scratch variables for the parser of `\wires`: the keys of a piece with those inherited, the keys of the piece alone, the rest of the piece, and a list whose brackets are protected.

```
\tl_new:N \l__stridi_adj_keys_tl
\tl_new:N \l__stridi_adj_piece_keys_tl
\tl_new:N \l__stridi_adj_rest_tl
\tl_new:N \l__stridi_adj_list_tl
```

(*End of definition for* `\l__stridi_adj_keys_tl` *and others.*)

`\__stridi_adj_piece:nn` `\__stridi_adj_piece_loop:` `\__stridi_drop_keys:w`

Splits a piece of the adjacencies, such as an item or a value, into the keys in brackets that begin it and the rest. The keys are added to those inherited, in `\l__stridi_adj_keys_tl`, and the rest goes in `\l__stridi_adj_rest_tl`, without spaces around it, and without the braces of a group that it would be alone. Keys in brackets anywhere else in the piece are reported and dropped, and so are keys followed by nothing, since they begin no wire.

#1 : inherited Ti*k*Z keys
#2 : piece, with its brackets protected

```
\msg_new:nnn { string-diagrams } { misplaced-keys }
  {
    Keys~in~brackets~after~the~beginning~of~‘#1’~are~ignored:~
    keys~begin~an~item~or~a~value,~as~in~a.north~=~[red]~b.south.
  }
\msg_new:nnn { string-diagrams } { keys-without-wire }
  { Keys~‘[#1]’~begin~no~wire,~so~they~are~ignored. }
\cs_new_protected:Nn \__stridi_adj_piece:nn {
  \tl_clear:N \l__stridi_adj_piece_keys_tl
  \tl_set:Nn \l__stridi_adj_rest_tl { #2 }
  \__stridi_adj_piece_loop:
  \tl_if_in:NnT \l__stridi_adj_rest_tl { \s__stridi_keys }
    {
      \tl_set:Ne \l__stridi_adj_rest_tl
        {
          \exp_after:wN \__stridi_drop_keys:w \exp_after:wN \q_mark
            \l__stridi_adj_rest_tl \s__stridi_keys \s__stridi_stop
        }
      \tl_trim_spaces:N \l__stridi_adj_rest_tl
      \msg_error:nnV { string-diagrams } { misplaced-keys } \l__stridi_adj_rest_tl
    }
  \bool_lazy_and:nnT
    { \tl_if_head_is_group_p:V \l__stridi_adj_rest_tl }
    { \int_compare_p:nNn { \tl_count:N \l__stridi_adj_rest_tl } = { 1 } }
    { \tl_set:Ne \l__stridi_adj_rest_tl { \tl_head:N \l__stridi_adj_rest_tl } }
  \tl_set:Nn \l__stridi_adj_keys_tl { #1 }
  \tl_if_blank:VF \l__stridi_adj_piece_keys_tl
    {
      \tl_if_blank:VTF \l__stridi_adj_rest_tl
        {
          \msg_error:nnV { string-diagrams } { keys-without-wire }
            \l__stridi_adj_piece_keys_tl
        }
        {
          \tl_if_blank:nF { #1 } { \tl_put_right:Nn \l__stridi_adj_keys_tl { , } }
          \tl_put_right:NV \l__stridi_adj_keys_tl \l__stridi_adj_piece_keys_tl
        }
    }
}
\cs_new_protected:Nn \__stridi_adj_piece_loop: {
  \tl_trim_spaces:N \l__stridi_adj_rest_tl
  \tl_if_head_eq_meaning:VNT \l__stridi_adj_rest_tl \s__stridi_keys
    {
      \tl_if_blank:VF \l__stridi_adj_piece_keys_tl
        { \tl_put_right:Nn \l__stridi_adj_piece_keys_tl { , } }
      \tl_put_right:Ne \l__stridi_adj_piece_keys_tl
        { \tl_item:Nn \l__stridi_adj_rest_tl { 2 } }
      \tl_set:Ne \l__stridi_adj_rest_tl { \tl_tail:N \l__stridi_adj_rest_tl }
      \tl_set:Ne \l__stridi_adj_rest_tl { \tl_tail:N \l__stridi_adj_rest_tl }
      \__stridi_adj_piece_loop:
    }
```

```
}
\cs_new:Npn \__stridi_drop_keys:w #1 \s__stridi_keys #2
  {
    \exp_not:o { \use_none:n #1 }
    \str_if_eq:nnF { #2 } { \s__stridi_stop } { \__stridi_drop_keys:w \q_mark }
  }
\prg_generate_conditional_variant:Nnn \tl_if_head_is_group:n { V } { p }
\prg_generate_conditional_variant:Nnn \tl_if_head_eq_meaning:nN { V } { T }
```

(*End of definition for* `\__stridi_adj_piece:nn`, `\__stridi_adj_piece_loop:`, *and* `\__stridi_drop_keys:w`.)

`\__stridi_adj_if_name:nTF` Tests whether a piece of the adjacencies, once its keys are peeled, can be a name: a node, an anchor, or both. A piece with a comma, an equals sign or braces in it is a group in braces where a name belongs, such as `{b.south, c.south}` in a list of targets, which has no beginning to share: it is reported. A blank piece is not a name either, silently, like an empty item of a list.

```
\msg_new:nnn { string-diagrams } { group-without-beginning }
  {
    '#1'~is~a~group~without~a~beginning,~so~it~is~ignored:~
    a~group~follows~a~node~or~an~anchor~and~=,~as~in~b~=~\{~south.1,~south.2~\}.
  }
\prg_new_protected_conditional:Nnn \__stridi_adj_if_name:n { T, F, TF }
  {
    \regex_match:nnTF { [,=] | \cB. | \cE. } { #1 }
      {
        \msg_error:nnn { string-diagrams } { group-without-beginning } { #1 }
        \prg_return_false:
      }
      { \tl_if_blank:nTF { #1 } { \prg_return_false: } { \prg_return_true: } }
  }
\prg_generate_conditional_variant:Nnn \__stridi_adj_if_name:n { V } { T, F, TF }
```

(*End of definition for* `\__stridi_adj_if_name:nTF`.)

`\__stridi_adj_map_function:nnN` `\__stridi_adj_top_alone:n` Unrolls the adjacencies given to `\wires` into wires and maps a function over them, in order. The given list of Ti*k*Z keys is passed through to the function, with the two ends of each wire, each as a node and an anchor. A wire is written ⟨*node*⟩`.`⟨*anchor*⟩ `=` ⟨*node*⟩`.`⟨*anchor*⟩, and wires that begin alike can share their beginning: it is written once, followed by `=` and a group of the rest of each wire. A beginning ends after a node name or at the `=` of the wire, never inside an anchor, so a group holds anchors after the node a wire starts from, targets after the `=` of a wire, and anchors after the node of a target. So the entries at the top are read as wires, or as groups after the node they start from; a node listed twice keeps the wires of both entries, and an anchor listed twice fans out. Keys in brackets can begin any item or value: they are added to the keys of the call, and of the groups around them, for every wire they reach, so that the keys closest to a wire come last and win.

`#1` : Ti*k*Z keys to pass through
`#2` : adjacencies
`#3` : function to be mapped over wires

```
\msg_new:nnn { string-diagrams } { node-without-wires }
  { Node~‘#1’~has~no~wires:~list~them~as~in~#1~=~\{~north~=~b.south~\}. }
\msg_new:nnn { string-diagrams } { end-without-target }
  { Wire~end~‘#1’~has~no~target:~give~it~one~as~in~#2~=~b.south. }
\cs_new_protected:Nn \__stridi_adj_map_function:nnN {
  \__stridi_protect_brackets:Nn \l__stridi_adj_list_tl { #2 }
  \keyval_parse:nnV
    { \__stridi_adj_top_alone:n }
    { \__stridi_adj_source:nNnn { #1 } #3 }
    \l__stridi_adj_list_tl
}
\cs_new_protected:Nn \__stridi_adj_top_alone:n {
  \__stridi_adj_piece:nn { } { #1 }
  \__stridi_adj_if_name:VT \l__stridi_adj_rest_tl
    {
      \tl_if_in:VnTF \l__stridi_adj_rest_tl { . }
        {
          \msg_error:nnVV { string-diagrams } { end-without-target }
            \l__stridi_adj_rest_tl \l__stridi_adj_rest_tl
        }
        { \msg_error:nnV { string-diagrams } { node-without-wires } \l__stridi_adj_rest_tl }
    }
}
\cs_generate_variant:Nn \keyval_parse:nnn { nnV }
```

(*End of definition for* `\__stridi_adj_map_function:nnN` *and* `\__stridi_adj_top_alone:n`.)

`\__stridi_adj_source:nNnn`
`\__stridi_adj_source_name:Nn`
`\__stridi_adj_node_list:nNnn`
`\__stridi_adj_node_item:NNnn`

Reads an entry at the top: a wire, or a group, from the beginning of the wires to the rest of each. A beginning with an anchor ends at the = of its wires, and one without ends after the name of their node.

`#1` : Ti*k*Z keys to pass through
`#2` : function to be mapped over wires
`#3` : beginning: a node, or a node and an anchor
`#4` : rest of each wire

```
\cs_new_protected:Nn \__stridi_adj_source:nNnn {
  \__stridi_adj_piece:nn { #1 } { #3 }
  \__stridi_adj_if_name:VT \l__stridi_adj_rest_tl { \__stridi_adj_source_name:Nn #2 { #4 } }
}
\cs_new_protected:Nn \__stridi_adj_source_name:Nn {
  \tl_if_in:VnTF \l__stridi_adj_rest_tl { . }
    {
      \__stridi_split_node_anchor:NNV \l__stridi_adj_node_tl \l__stridi_adj_anchor_tl \l__st
      \__stridi_adj_targets:VNVVn \l__stridi_adj_keys_tl #1
        \l__stridi_adj_node_tl \l__stridi_adj_anchor_tl { #2 }
    }
    {
      \tl_set_eq:NN \l__stridi_adj_node_tl \l__stridi_adj_rest_tl
      \__stridi_adj_piece:Vn \l__stridi_adj_keys_tl { #2 }
      \__stridi_protect_brackets:NV \l__stridi_adj_list_tl \l__stridi_adj_rest_tl
      \__stridi_adj_node_list:VNVV \l__stridi_adj_keys_tl #1 \l__stridi_adj_node_tl \l__stri
    }
}
```

```
\cs_new_protected:Nn \__stridi_adj_node_list:nNnn {
  \keyval_parse:nnn
    { \__stridi_adj_node_alone:nNnn { #1 } #2 { #3 } }
    { \__stridi_adj_node_item:nNnnn { #1 } #2 { #3 } }
    { #4 }
}
\cs_new_protected:Nn \__stridi_adj_node_item:nNnnn {
  \__stridi_adj_piece:nn { #1 } { #4 }
  \__stridi_adj_if_name:VT \l__stridi_adj_rest_tl
    { \__stridi_adj_targets:VNnVn \l__stridi_adj_keys_tl #2 { #3 } \l__stridi_adj_rest_tl {
}
\cs_generate_variant:Nn \__stridi_split_node_anchor:NNn { NNV }
\cs_generate_variant:Nn \__stridi_adj_piece:nn { V }
\cs_generate_variant:Nn \__stridi_adj_node_list:nNnn { VNVV }
```

(*End of definition for* `\__stridi_adj_source:nNnn` *and others.*)

`\__stridi_adj_node_alone:nNnn`
`\__stridi_adj_node_alone_name:Nn`

Reads an item without `=` in a group after the node a wire starts from. It is the target of a wire from that node without an anchor, as in `f = { g }`, if it names a node that exists. Otherwise it is an error: an anchor left without a target, if it is an anchor of the node the wires start from, as the shape of that node tells, or an angle, which pgf reads as a point of the border; or else a node that doesn't exist; if the node the wires start from doesn't exist either, the error names it, since the item can't be told apart.

`#1` : Ti*k*Z keys to pass through
`#2` : function to be mapped over wires
`#3` : node the wires start from
`#4` : item

```
\cs_new_protected:Nn \__stridi_adj_node_alone:nNnn {
  \__stridi_adj_piece:nn { #1 } { #4 }
  \__stridi_adj_if_name:VT \l__stridi_adj_rest_tl { \__stridi_adj_node_alone_name:Nn #2 { #3
}
\cs_new_protected:Nn \__stridi_adj_node_alone_name:Nn {
  \__stridi_split_node_anchor:NNV \l__stridi_adj_node_tl \l__stridi_adj_anchor_tl \l__stridi
  \cs_if_exist:cTF { pgf@sh@ns@ \l__stridi_adj_node_tl }
    { \__stridi_adj_wire:VNnnV \l__stridi_adj_keys_tl #1 { #2 } { } \l__stridi_adj_rest_tl }
    {
      \cs_if_exist:cTF { pgf@sh@ns@ #2 }
        {
          \__stridi_if_anchor:nVTF { #2 } \l__stridi_adj_rest_tl
            {
              \msg_error:nnee { string-diagrams } { end-without-target }
                { \exp_not:n { #2 } . \exp_not:V \l__stridi_adj_rest_tl }
                { \exp_not:V \l__stridi_adj_rest_tl }
            }
            { \msg_error:nnV { string-diagrams } { end-no-node } \l__stridi_adj_rest_tl }
        }
        { \msg_error:nnn { string-diagrams } { end-no-node } { #2 } }
    }
}
```

(*End of definition for* `\__stridi_adj_node_alone:nNnn` *and* `\__stridi_adj_node_alone_name:Nn`.)

__stridi_adj_targets:nNnnn
__stridi_adj_target_list:nNnnn
__stridi_adj_target_alone:nNnnn
__stridi_adj_target_node:nNnnnn
__stridi_adj_target_node_name:Nnnn
__stridi_adj_target_anchors:nNnnnn
__stridi_adj_target_anchor:nNnnnn

Reads the rest of the wires from an end: their targets, or groups after the node of a target, which hold its anchors. A group can't begin inside an anchor, as in `b.east = { 1, 2 }`, and the anchors of a target take no value.

#1 : Ti*k*Z keys to pass through
#2 : function to be mapped over wires
#3 : node the wires start from
#4 : anchor the wires start from
#5 : targets

```
\msg_new:nnn { string-diagrams } { target-group-in-anchor }
  {
    Wire~end~‘#1’~can’t~begin~a~group:~
    targets~share~their~node,~as~in~b~=~\{~east.1,~east.2~\},~not~part~of~an~anchor.
  }
\msg_new:nnn { string-diagrams } { target-anchor-with-value }
  {
    Wire~end~‘#1.#2’~has~a~value:~
    list~the~anchors~of~‘#1’~alone,~as~in~#1~=~\{~east.1,~east.2~\}.
  }
\cs_new_protected:Nn \__stridi_adj_targets:nNnnn {
  \__stridi_adj_piece:nn { #1 } { #5 }
  \__stridi_protect_brackets:NV \l__stridi_adj_list_tl \l__stridi_adj_rest_tl
  \__stridi_adj_target_list:VNnnV \l__stridi_adj_keys_tl #2 { #3 } { #4 } \l__stridi_adj_lis
}
\cs_new_protected:Nn \__stridi_adj_target_list:nNnnn {
  \keyval_parse:nnn
    { \__stridi_adj_target_alone:nNnnn { #1 } #2 { #3 } { #4 } }
    { \__stridi_adj_target_node:nNnnnn { #1 } #2 { #3 } { #4 } }
    { #5 }
}
\cs_new_protected:Nn \__stridi_adj_target_alone:nNnnn {
  \__stridi_adj_piece:nn { #1 } { #5 }
  \__stridi_adj_if_name:VT \l__stridi_adj_rest_tl
    { \__stridi_adj_wire:VNnnV \l__stridi_adj_keys_tl #2 { #3 } { #4 } \l__stridi_adj_rest_t
}
\cs_new_protected:Nn \__stridi_adj_target_node:nNnnnn {
  \__stridi_adj_piece:nn { #1 } { #5 }
  \__stridi_adj_if_name:VT \l__stridi_adj_rest_tl
    { \__stridi_adj_target_node_name:Nnnn #2 { #3 } { #4 } { #6 } }
}
\cs_new_protected:Nn \__stridi_adj_target_node_name:Nnnn {
  \tl_if_in:VnTF \l__stridi_adj_rest_tl { . }
    { \msg_error:nnV { string-diagrams } { target-group-in-anchor } \l__stridi_adj_rest_tl }
    {
      \tl_set_eq:NN \l__stridi_adj_node_tl \l__stridi_adj_rest_tl
      \__stridi_adj_piece:Vn \l__stridi_adj_keys_tl { #4 }
      \__stridi_protect_brackets:NV \l__stridi_adj_list_tl \l__stridi_adj_rest_tl
      \__stridi_adj_target_anchors:VNnnVV \l__stridi_adj_keys_tl #1 { #2 } { #3 }
        \l__stridi_adj_node_tl \l__stridi_adj_list_tl
    }
}
\cs_new_protected:Nn \__stridi_adj_target_anchors:nNnnnn {
  \keyval_parse:nnn
```

```
    { \__stridi_adj_target_anchor:nNnnnn { #1 } #2 { #3 } { #4 } { #5 } }
    { \msg_error:nnnn { string-diagrams } { target-anchor-with-value } { #5 } }
    { #6 }
}
\cs_new_protected:Nn \__stridi_adj_target_anchor:nNnnnn {
  \__stridi_adj_piece:nn { #1 } { #6 }
  \__stridi_adj_if_name:VT \l__stridi_adj_rest_tl
    {
      \use:e
        {
          \exp_not:N #2 { \exp_not:V \l__stridi_adj_keys_tl }
            { \exp_not:n { #3 } } { \exp_not:n { #4 } }
            { \exp_not:n { #5 } } { \exp_not:V \l__stridi_adj_rest_tl }
        }
    }
}
\cs_generate_variant:Nn \__stridi_adj_targets:nNnnn { VNVV , VNnV }
\cs_generate_variant:Nn \__stridi_adj_target_list:nNnnn { VNnnV }
\cs_generate_variant:Nn \__stridi_adj_target_anchors:nNnnnn { VNnnVV }
```

(*End of definition for* `\__stridi_adj_targets:nNnnn` *and others.*)

`\__stridi_adj_wire:nNnnn` Applies a function to one wire, splitting its target into node and anchor.

`#1` : TikZ keys to pass through
`#2` : function to be applied
`#3` : source node
`#4` : source anchor
`#5` : target

```
\cs_new_protected:Nn \__stridi_adj_wire:nNnnn {
  \__stridi_split_node_anchor:NNn \l__stridi_target_node_tl \l__stridi_target_anchor_tl { #5
  \exp_args:Nnnnoo #2
    { #1 } { #3 } { #4 } { \l__stridi_target_node_tl } { \l__stridi_target_anchor_tl }
}
\cs_generate_variant:Nn \__stridi_adj_wire:nNnnn { VNnnV }
```

(*End of definition for* `\__stridi_adj_wire:nNnnn`.)

`\wires` Define our superstar.

```
\NewDocumentCommand{\wires}{ O{} m }
{
  \__stridi_draw_boundary:
  \__stridi_adj_map_function:nnN
    { #1 } { #2 } \__stridi_draw_wire:nnnnn
}
```

(*End of definition for* `\wires`. *This function is documented on page* 3.)

Close the DocStrip guards and call it a day.

```
⟨/package⟩
```

Change History

0.1.0
General: initial version 1
0.2.0
General: make box ports configurable . 1
/pgf/box: acts as a shortcut for setting port counts 12
box: make ports configurable through Ti*k*Z keys 11
0.2.1
\wires: now correctly handles optional style parameter 40
0.3.0
General: replace loose ends with wires to the boundary node @ 1
/pgf/box: without counts, keeps the ones in force 12
/pgf/box␣ports: counts start at zero, and the shortcut checks that it gets four 9
box: ports a box doesn't have are an error 11
\wires: every listed wire is drawn, so nodes can be listed twice and anchors can fan out 40
loose ends are wires to the boundary node @, replacing the third argument 40
wire ends off the sides or without an anchor are errors 40
wires leave corners diagonally 40
0.4.0
General: write wires whole, leave out anchors, draw wires on Ti*k*Z paths with the wire key, and label their ends 1

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols

\- 320
\. 273, 354
/pgf/box *1*, 123
/pgf/box␣ports *2*, 13
/pgf/box␣ports␣east *2*, 13
/pgf/box␣ports␣north *2*, 13
/pgf/box␣ports␣south *2*, 13
/pgf/box␣ports␣west *2*, 13
/pgf/dot *8*, 138
/tikz/beside *7*, 868
/tikz/beside␣end *7*, 868
/tikz/beside␣start *7*, 868
/tikz/beyond *7*, 868
/tikz/beyond␣end *7*, 868
/tikz/beyond␣start *7*, 868
/tikz/every␣wire *6*, 811
/tikz/wire *6*, 811
/tikz/wire␣flow *5*, 811
\\ 387
\{ 986, 999, 1079, 1084
\} 986, 999, 1079, 1084
\␣ 166, 167, 168, 169, 170, 171, 172, 173, 197, 198, 199, 200

A

\A 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 197, 198, 199, 200, 201, 202, 203, 204, 320, 354, 394
\AddToHook 291, 803, 805, 807, 809
\AddToHookWithArguments 289

B

\b 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 197, 198, 199, 200, 201, 202, 203, 204
bool commands:
\bool_if:NTF 416, 426, 674, 679, 681, 687, 695, 745, 775, 777, 790, 792, 797, 804, 806, 808, 810

\bool_if:nTF 533, 536, 613, 631
\bool_lazy_all:nTF 300, 488
\bool_lazy_and:nnTF 244, 405, 706, 736, 737, 765, 773, 785, 945
\bool_lazy_any:nTF 500
\bool_lazy_or:nnTF 539
\bool_new:N 389, 390, 451, 452, 457, 458, 475, 476, 723, 724, 725
\bool_set:Nn 531, 532, 762
\bool_set_eq:NN 393
\bool_set_false:N 392, 516, 517, 545, 595, 643, 729, 730, 749
\bool_set_true:N 396, 411, 419, 432, 443, 522, 523, 534, 537, 739, 786, 812
\c_false_bool 735
\c_true_bool 733
box 84

C

\cB 990
\cE 990
clist commands:
\clist_map_inline:nn 88
\coordinate 4
cs commands:
\cs_generate_variant:Nn . 369, 370, 371, 445, 446, 466, 485, 598, 599, 702, 703, 704, 713, 891, 1021, 1051, 1052, 1053, 1136, 1137, 1138, 1144
\cs_if_exist:NTF 51, 62, 215, 280, 292, 317, 403, 1060, 1063
\cs_if_exist_p:N 245
\cs_new:Nn 239, 574, 579, 823, 831
\cs_new:Npn 905, 976
\cs_new_nopar:Npn 55
\cs_new_protected:Nn 42, 49, 150, 161, 194, 214, 230, 250, 262, 272, 279, 299, 336, 350, 358, 391, 439, 459, 477, 515, 557, 586, 610, 628, 667, 705, 728, 756, 789, 811, 835, 852, 887, 931, 963, 1002, 1009, 1022, 1026, 1040, 1046, 1054, 1058, 1086, 1091, 1097, 1102, 1107, 1118, 1124, 1139
\cs_new_protected:Npn 293, 892, 904, 906
\cs_set:Npe 742, 744, 844, 856
\cs_set:Npn 750, 751, 796, 813, 814, 836
\csname 111

D

dim commands:
\dim_abs:n 303, 304, 618, 619, 633, 634, 647, 649
\dim_compare:nNnTF . 253, 646, 651, 656
\dim_compare_p:n 303, 304, 490, 491, 492, 493, 502, 503, 504, 505, 616, 633, 634
\dim_max:nn 157, 158
\dim_min:nn 154, 155
\dim_new:N 148, 149, 453, 454, 455, 456, 467, 468, 469, 470, 471, 472, 473, 474
\dim_set:Nn 151, 152, 154, 155, 157, 158, 463, 464, 479, 480, 482, 483
\dim_to_fp:n 855
\dim_use:N 285, 709
\draw 6, 8

E

\endcsname 111
exp commands:
\exp_after:wN 939
\exp_args:Ne 30
\exp_args:Nnnnoo 876, 1141
\exp_args_generate:n 876
\exp_not:N 233, 741, 743, 846, 847, 913, 1130
\exp_not:n 235, 240, 741, 743, 793, 799, 905, 913, 978, 1068, 1069, 1130, 1131, 1132
\expandafter 111
\ExplSyntaxOff 123, 138
\ExplSyntaxOn 137, 147

F

\foreach 17
fp commands:
\fp_compare:nNnTF 861
\fp_eval:n 624, 625
\fp_new:N 833
\fp_set:Nn 854
\fp_to_int:N 860

G

group commands:
\group_begin: 340
\group_end: 343

I

\inheritanchor 96
\inheritanchorborder 97
\inheritbackgroundpath 98
\inheritsavedanchors 87
int commands:
\int_case:nn 576, 825
\int_case:nnTF 588
\int_compare:nNnTF 28, 58, 611
\int_compare:nTF 327

\int_compare_p:nNn 947
\int_eval:n 776, 778
\int_gset:Nn 342
\int_mod:nn 576, 611, 825
\int_new:N 193, 334, 335, 572
\int_set:Nn 197, 198, 199, 200, 201, 202, 203, 204, 624, 625, 652, 653, 657, 658, 663, 673, 678
\int_set_eq:NN 344
\int_step_inline:nnnn 50
\int_use:N 207
\int_zero:N 337
iow commands:
\iow_char:N 387

K

keyval commands:
\keyval_parse:nnn 1004, 1021, 1041, 1092, 1119

M

msg commands:
\msg_error:nn 895, 910
\msg_error:nnn 38, 226, 401, 408, 429, 431, 570, 638, 943, 954, 992, 1018, 1071, 1073, 1109
\msg_error:nnnn 361, 442, 543, 640, 1015, 1067, 1121
\msg_error:nnnnn 594, 599
\msg_error:nnnnnn 60, 364, 370
\msg_new:nnn 13, 47, 347, 372, 377, 379, 384, 510, 552, 581, 600, 605, 883, 885, 924, 929, 983, 998, 1000, 1076, 1081

N

\NeedsTeXFormat 3
\NewDocumentCommand 1145
\node 263
\northeast 113, 115, 117, 119

P

\path *6*, *8*, 233
\pgfdeclareshape 84
\pgfextractx 43
\pgfextracty 44
\pgfkeys 19
\pgfkeysalso 30, 820, 843
\pgfkeysvalueof 102, 104, 106, 108
\pgfmathdivide 73
\pgfmathresult 77
\pgfmathtruncatemacro . 102, 104, 106, 108
\pgfpictureid . *17*, 246, 280, 284, 292, 302
\pgfpoint 45, 159, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189
\pgfpointanchor *17*, 153, 156, 162, 163, 461, 462, 478, 481
\pgfpointlineattime 67, 76
\pgfreferencednodename 62, 63
\portseast 103, 104, 114
\portsnorth 101, 102, 112
\portssouth 105, 106, 116
\portswest 107, 108, 118
prg commands:
\prg_break: 53
\prg_do_nothing: 54
\prg_generate_conditional_variant:Nnn 333, 802, 918, 919, 981, 982, 997
\prg_new_conditional:Nnn 241, 486, 498, 714
\prg_new_protected_conditional:Nnn 312, 988
\prg_return_false: 248, 322, 330, 496, 508, 721, 993, 995
\prg_return_true: 247, 318, 321, 329, 495, 507, 718, 719, 995
\ProvidesExplPackage 8

Q

quark commands:
\q_mark 889, 901, 914, 939, 979
\q_stop *18*, 293, 397, 889, 896, 904, 905, 907, 909

R

regex commands:
\regex_extract_once:nnN 273, 354
\regex_match:nnTF . . 320, 394, 428, 990
\regex_match_case:nn 164
\regex_match_case:nnTF 195
\RequirePackage 4, 5

S

\savedmacro 101, 103, 105, 107
scan commands:
\scan_new:N 727, 881, 882
scan internal commands:
\s__stridi_auto 723, 741, 743, 813, 814
\s__stridi_keys *33*, 881, 913, 935, 940, 965, 976
\s__stridi_stop 881, 940, 979
seq commands:
\seq_clear:N 351
\seq_count:N 28
\seq_gclear:N 292
\seq_gput_left:Nn 282

\seq_if_empty:NTF 315, 360
\seq_item:Nn 32, 33, 34, 35, 326, 328, 363, 365
\seq_map_break: 309
\seq_map_inline:Nn 296
\seq_new:N 18, 271, 278, 349
\seq_pop:NN 274, 275, 276
\seq_set_split:Nnn 27
\southwest 113, 115, 117, 119
str commands:
\str_case:nn 580, 664, 704
\str_case:nnTF 558, 716
\str_if_eq:nnTF 338, 352, 518, 520, 637, 741, 743, 832, 838, 979
\str_if_eq_p:nn 246, 302, 406, 621, 622
stridi internal commands:
\l__stridi_adj_anchor_tl 877, 1029, 1031, 1059
__stridi_adj_if_name:n 988, 997
__stridi_adj_if_name:nTF 983, 1011, 1024, 1048, 1056, 1099, 1104, 1126
\l__stridi_adj_keys_tl *34*, 920, 949, 958, 959, 1030, 1035, 1037, 1049, 1061, 1089, 1100, 1112, 1114, 1130
\l__stridi_adj_list_tl 920, 1003, 1007, 1036, 1037, 1088, 1089, 1113, 1115
__stridi_adj_map_function:nnN 998, 1002, 1148
__stridi_adj_node_alone:nNnn 1042, 1054, 1054
__stridi_adj_node_alone_name:Nn 1054, 1056, 1058
__stridi_adj_node_item:nNnnn 1022, 1043, 1046
__stridi_adj_node_list:nNnn 1022, 1037, 1040, 1053
\l__stridi_adj_node_tl 877, 1029, 1031, 1034, 1037, 1059, 1060, 1111, 1115
__stridi_adj_piece:nn 924, 931, 1010, 1023, 1035, 1047, 1052, 1055, 1087, 1098, 1103, 1112, 1125
\l__stridi_adj_piece_keys_tl . . . 920, 932, 950, 955, 959, 967, 968, 969
__stridi_adj_piece_loop: 924, 934, 963, 973
\l__stridi_adj_rest_tl *34*, 920, 933, 935, 937, 940, 942, 943, 946, 947, 948, 952, 964, 965, 970, 971, 972, 1011, 1013, 1016, 1018, 1024, 1027, 1029, 1034, 1036, 1048, 1049, 1056, 1059, 1061, 1065, 1068, 1069, 1071, 1088, 1099, 1100, 1104, 1108, 1109, 1111, 1113, 1126, 1132
__stridi_adj_source:nNnn 1006, 1022, 1022
__stridi_adj_source_name:Nn 1022, 1024, 1026
__stridi_adj_target_alone:nNnnn 1076, 1093, 1097
__stridi_adj_target_anchor:nNnnnn 1076, 1120, 1124
__stridi_adj_target_anchors:nNnnnn 1076, 1114, 1118, 1138
__stridi_adj_target_list:nNnnn 1076, 1089, 1091, 1137
__stridi_adj_target_node:nNnnnn 1076, 1094, 1102
__stridi_adj_target_node_name:Nnnn 1076, 1105, 1107
__stridi_adj_targets:nNnnn 1030, 1049, 1076, 1086, 1136
__stridi_adj_top_alone:n 998, 1005, 1009
__stridi_adj_wire:nNnnn *33*, 1061, 1100, 1139, 1139, 1144
__stridi_anchor_to_angle:Nn *14*, 194, 194, 440, 446, 763, 764, 770, 771, 840
\g__stridi_anchors_seq 278, 282, 292, 296
\l__stridi_angle_int . . . 193, 197, 198, 199, 200, 201, 202, 203, 204, 207
__stridi_before_closing:w 883, 896, 905
__stridi_both_bare_direction: 600, 628, 671
__stridi_calctarget:nnnn *28*, 161, 161, 708
__stridi_clamp_to_node:n 148, 150, 191
__stridi_compass:n *31*, 823, 823, 846, 858
\l__stridi_dir_int 572, 611, 624, 625, 652, 653, 657, 658, 663, 673, 678, 686, 694
__stridi_draw_boundary: 250, 250, 1147
__stridi_draw_boundary_around:nn 250, 254, 256, 262
__stridi_draw_wire:nnnnn 214, 214, 1149
__stridi_draw_wire_edge:nnnnn 214, 218, 221, 230
__stridi_drop_keys:w 924, 939, 976, 979

\l__stridi_end_anchor_tl 447, 541, 670, 692, 697, 735, 759, 764, 767, 771, 781, 783
\l__stridi_end_angle_tl 438, 440, 441
__stridi_end_coordinate:NnnnnN 13, 705, 705, 713, 779, 782
\l__stridi_end_east_dim 467, 503, 530
\l__stridi_end_holds_bool 467, 532, 533, 536
\l__stridi_end_in_bool 447, 517, 523, 537, 674, 695, 777, 784
\l__stridi_end_missing_bool 20, 389, 419, 432, 729, 797
\l__stridi_end_node_tl 447, 518, 529, 544, 637, 641, 698, 735, 759, 781, 783
\l__stridi_end_north_dim 467, 505, 530
\l__stridi_end_ok_bool 447, 697, 735, 736, 737, 773
\l__stridi_end_resolved_bool 390, 393, 396, 416, 426
\l__stridi_end_south_dim 467, 504, 530
\l__stridi_end_west_dim 467, 502, 530
\l__stridi_end_x_dim 447, 618, 633, 647, 651, 760
\l__stridi_end_y_dim 447, 619, 634, 649, 656, 760
__stridi_faces_inwards: 22, 29, 510, 515, 761
__stridi_find_anchor:NNw 293, 293, 397
__stridi_find_anchor_check:NNnnnnnnn 293, 297, 299
\l__stridi_flow_tl 551, 560, 561, 562, 563, 564, 565, 566, 567, 568, 615, 621, 622, 629, 664
\l__stridi_given_target_tl 723, 731, 799
\l__stridi_iangle_tl 210, 674, 744, 764, 771, 778
__stridi_if_anchor:nn 312, 333
__stridi_if_anchor:nnTF 220, 312, 413, 1065
__stridi_if_boundary_in_picture: 241
__stridi_if_boundary_in_picture:TF 241, 251
__stridi_if_boundary_in_picture_p: 241, 407
__stridi_if_box_within:nn 486
__stridi_if_box_within_p:nn 23, 486, 531, 532
__stridi_if_boxes_apart: 498
__stridi_if_boxes_apart_p: 23, 486, 540
__stridi_if_side_name:n . . . 714, 802
__stridi_if_side_name_p:n . 706, 785
__stridi_infer_anchors: 600, 667, 769
__stridi_intersect_hv_lines_through:NN . . 42, 42, 69, 70, 78, 79
\l__stridi_label_angle_tl 833, 840, 841, 847
\l__stridi_label_fp . . . 833, 854, 860
__stridi_label_key:nnnnnn 835, 835, 869, 870
__stridi_label_on_path:nnnn 835, 837, 852
__stridi_no_anchor:nn 19, 222, 334, 358, 371, 418
__stridi_node_box:nNNNN 477, 477, 485, 527, 529
\l__stridi_oangle_tl 210, 679, 742, 763, 770, 776
__stridi_opposite:n 25, 574, 579, 688, 696, 702
__stridi_point_of_end:nnNN 459, 459, 466, 757, 759
\l__stridi_port_counts_seq 18, 27, 28, 32, 33, 34, 35
__stridi_port_of_box:nn 19, 314, 334, 350, 359
\l__stridi_port_seq 19, 315, 326, 328, 349, 351, 355, 360, 363, 365
\g__stridi_ports_int . . 335, 342, 344
\l__stridi_ports_int 326, 328, 334, 363, 366, 587, 588, 594
__stridi_protect_brackets:Nn . . . 883, 887, 891, 1003, 1036, 1088, 1113
__stridi_protect_brackets_in:Nw 883, 901, 906
__stridi_protect_brackets_out:Nw 883, 889, 892, 914
__stridi_protect_brackets_skip:w 883, 897, 904
__stridi_record_anchor:nn 279, 279, 290
__stridi_set_flow:n . . 552, 557, 821
__stridi_side_anchor:NNnn 25, 574, 586, 598, 689, 697, 703
__stridi_side_of:n 25, 574, 574, 686, 694
__stridi_side_ports:nnN 19, 326, 334, 336, 363, 369, 587
\l__stridi_side_tl 572, 686, 688, 690, 694, 696, 698
__stridi_side_turn:nn 31, 823, 831, 847, 862, 863

__stridi_snap: 600, 610, 683
\l__stridi_source_tl . . 210, 746, 779
__stridi_split_node_anchor:NNn *17*, 272, 272, 399, 1029, 1051, 1059, 1140
\l__stridi_split_seq 271, 273, 274, 275, 276
\l__stridi_start_anchor_tl 447, 541, 668, 684, 689, 733, 757, 762, 763, 767, 770, 780, 784, 785
\l__stridi_start_east_dim 467, 502, 528
\l__stridi_start_holds_bool 467, 531, 533, 536
\l__stridi_start_in_bool . . . 447, 516, 522, 534, 679, 687, 775, 781, 785
\l__stridi_start_moved_bool 723, 730, 745, 762, 786, 792
\l__stridi_start_node_tl 447, 518, 520, 527, 544, 637, 638, 641, 690, 733, 757, 780, 784
\l__stridi_start_north_dim 467, 504, 528
\l__stridi_start_ok_bool 447, 545, 643, 681, 689, 733, 736, 737, 766, 773
\l__stridi_start_south_dim 467, 505, 528
\l__stridi_start_west_dim 467, 503, 528
\l__stridi_start_x_dim 447, 618, 633, 647, 651, 758
\l__stridi_start_y_dim 447, 619, 634, 649, 656, 758
__stridi_subdivide_segment:nNNNNN 47, 49, 112, 114, 116, 118
\l__stridi_target_anchor_tl 877, 1140, 1142
\l__stridi_target_node_tl 877, 1140, 1142
\l__stridi_target_tl . . 210, 740, 782
__stridi_wire: *14*, *15*, *21*, *29*, 728, 728, 804, 808
\l__stridi_wire_bool 723, 804, 806, 808, 810, 812
__stridi_wire_coordinate:nn 214, 227, 234, 236, 239
\l__stridi_wire_drawn_bool 723, 739, 749, 790
__stridi_wire_end:nNNNN 372, 391, 445, 732, 734
__stridi_wire_end_side:NNN 372, 414, 439
__stridi_wire_ends: . . 728, 736, 756
__stridi_wire_key: 811, 820
__stridi_wire_path: *29*, 728, 789, 806, 810
\l__stridi_x_dim 148, 151, 154, 157, 159
\l__stridi_y_dim 149, 152, 155, 158, 159

T

TEX and LATEX 2ε commands:
\pgf@picmaxx 253
\pgf@x 151, 154, 157, 162, 163, 285, 463, 479, 482, 709, 855
\pgf@xa 43, 45, 162, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 186, 187, 188, 189
\pgf@xb . . . 163, 170, 171, 172, 173, 182, 183, 184, 185, 186, 187, 188, 189
\pgf@y 152, 155, 158, 162, 163, 285, 464, 480, 483, 709, 855
\pgf@ya 44, 45, 162, 166, 167, 168, 169, 170, 171, 172, 173, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189
\pgf@yb . . . 163, 166, 167, 168, 169, 174, 175, 176, 177, 178, 179, 180, 181
\pgfutil@g@addto@macro 111
\tikz@anchor 838, 844, 856
\tikz@auto@anchor@direction . . . 832
\tikz@auto@post 866
\tikz@auto@pre 853
\tikz@computed@path . . . 793, 798, 799
\tikz@do@auto@anchor 836
\tikz@label@angle 840
\tikz@time 861
\tikz@to@in 743, 744, 751, 814
\tikz@to@out 741, 742, 750, 813
\tikz@to@relativefalse 815
\tikz@to@switch@on 816
\tikzset 124, 139, 818, 868
\tikztonodes 796
\tikztostart 732, 746, 752, 793
\tikztotarget 731, 734, 740, 753
tl commands:
\tl_clear:N 208, 294, 295, 568, 798, 888, 932
\tl_count:N 947
\tl_head:N 948
\tl_if_blank:nTF 25, 240, 950, 952, 958, 967, 995
\tl_if_empty:NTF 400, 410, 441, 629, 668, 670, 684, 692, 841
\tl_if_empty:nTF 217, 460
\tl_if_empty_p:N . . . 541, 615, 762, 767
\tl_if_eq:nn 918
\tl_if_eq:nnTF 907

\tl_if_head_eq_meaning:nN 982
\tl_if_head_eq_meaning:nNTF . . . 965
\tl_if_head_is_group:n 981
\tl_if_head_is_group_p:n 946
\tl_if_in:nn 919
\tl_if_in:NnTF 935
\tl_if_in:nnTF 893, 909, 1013, 1027, 1108
\tl_item:Nn 970
\tl_new:N 210, 211, 212, 213, 438, 447, 448, 449, 450, 551, 573, 726, 834, 877, 878, 879, 880, 920, 921, 922, 923
\tl_put_left:Nn 793
\tl_put_right:Nn 896, 900, 912, 958, 959, 968, 969
\tl_set:Nn 207, 307, 308, 560, 561, 562, 563, 564, 565, 566, 567, 590, 591, 686, 688, 694, 696, 709, 711, 752, 753, 776, 778, 799, 933, 937, 948, 949, 971, 972
\tl_set_eq:NN 731, 740, 746, 1034, 1111
\tl_tail:N 971, 972
\tl_trim_spaces:N 942, 964
\tl_use:N 742, 744

U

use commands:
\use:N 246, 317, 338, 341, 342, 352, 490, 491, 492, 493
\use:n 231, 1128
\use_none:n 893, 900, 905, 907, 909, 913, 978
\usepackage *1*
\usetikzlibrary *1*, 6, 7

W

\wires . . *3–6*, *8*, *9*, *15*, *16*, *33*, *34*, *36*, 1145

Z

\Z 320, 354, 394