Release Summary

v0.6

```
Import["https://qtechtheory.org/questlink.m"];
CreateDownloadedQuESTEnv[];
```

This release introduces device specifications for precise simulation of real hardware devices, utilities for analytically simplifying expressions of Pauli operators, new plotting capabilities, phase gates and any-qubit general unitary gates. Be sure to check out **InsertCircuitNoise**, which is the most substantial of the device specification related functions.

Table of Contents (clickable):

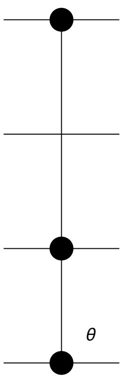
New gates

DrawCircuit[], **CalcCircuitMatrix[]** and **ApplyCircuit[]** now support the multi-qubit phase gates **Ph**

```
? Ph
```

Ph is the phase shift gate, which introduces phase factor exp(i∗theta)
    upon state |1...1> of the target and control qubits. The gate is the same under
    different orderings of qubits, and division between control and target qubits.

```
DrawCircuit @ Circuit @ Ph_{0,3,1}[θ]
```



```
CalcCircuitMatrix @ Circuit @ Ph_0[θ]
```

```
{{1, 0}, {0, e^(i θ)}}
```

```
CalcCircuitMatrix @ Circuit @ Ph_{0,2,1}[θ] // MatrixForm
```

```
( 1  0  0  0  0  0  0    0     )
( 0  1  0  0  0  0  0    0     )
( 0  0  1  0  0  0  0    0     )
( 0  0  0  1  0  0  0    0     )
( 0  0  0  0  1  0  0    0     )
( 0  0  0  0  0  1  0    0     )
( 0  0  0  0  0  0  1    0     )
( 0  0  0  0  0  0  0  e^(i θ) )
```

```
CalcCircuitMatrix @ Circuit @ C_{1,0}[Ph_2[θ]] // MatrixForm
```

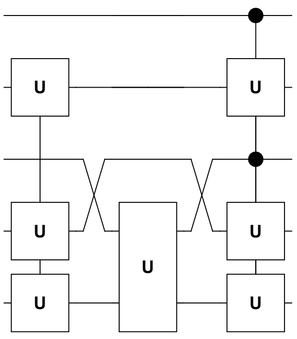
```
( 1  0  0  0  0  0  0    0     )
( 0  1  0  0  0  0  0    0     )
( 0  0  1  0  0  0  0    0     )
( 0  0  0  1  0  0  0    0     )
( 0  0  0  0  1  0  0    0     )
( 0  0  0  0  0  1  0    0     )
( 0  0  0  0  0  0  1    0     )
( 0  0  0  0  0  0  0  e^(i θ) )
```

```
ψ = InitPlusState @ CreateQureg[3];
ApplyCircuit[Circuit[ Ph_{0,1,2}[π/3] ], ψ];
GetQuregMatrix[ψ] // MatrixForm
DestroyQureg[ψ]
```

```
(     0.353553 + 0. i     )
(     0.353553 + 0. i     )
(     0.353553 + 0. i     )
(     0.353553 + 0. i     )
(     0.353553 + 0. i     )
(     0.353553 + 0. i     )
(     0.353553 + 0. i     )
( 0.176777 + 0.306186 i   )
```

These functions now also support general unitaries **U** with *any* number of target and control qubits!

```
DrawCircuit @ Circuit[ U_{0,1,3}[m] U_{0,2}[m] C_{4,2}[U_{0,1,3}[m]]]
```



```
m = RandomVariate @ CircularUnitaryMatrixDistribution[8];
CalcCircuitMatrix @ Circuit[ C_2[U_{0,3,1}[m]] ] // Chop // MatrixForm
```

```
( 1.  0   0   0          0                      0                      0
  0   1.  0   0          0                      0                      0
  0   0   1.  0          0                      0                      0
  0   0   0   1.         0                      0                      0
  0   0   0   0   0.216339 + 0.0574256 i   0.198238 + 0.0437794 i   0.0584521 - 0.338695
  0   0   0   0   -0.125926 - 0.405489 i   0.374411 + 0.307865 i   -0.0911487 - 0.182594
  0   0   0   0  -0.0164171 - 0.0272309 i  0.223708 - 0.147173 i   0.171821 + 0.325859
  0   0   0   0   0.425961 + 0.266029 i    0.250329 - 0.125352 i    0.439816 - 0.148638
  0   0   0   0          0                      0                      0
  0   0   0   0          0                      0                      0
  0   0   0   0          0                      0                      0
  0   0   0   0          0                      0                      0
  0   0   0   0   -0.339317 + 0.0134158 i  -0.413904 + 0.211306 i  -0.0142742 - 0.246384
  0   0   0   0  -0.0604362 + 0.0919235 i   0.434339 - 0.10463 i    -0.189323 - 0.450505
  0   0   0   0   0.102416 + 0.330571 i    -0.372911 - 0.127445 i  -0.0888425 - 0.281985
  0   0   0   0   0.220909 + 0.469479 i    0.0388954 + 0.0368131 i -0.317796 - 0.0295451
```

```
ψ = InitPlusState @ CreateQureg[4];
ApplyCircuit[Circuit[ C_2[U_{0,3,1}[m]] ], ψ];
GetQuregMatrix[ψ] // MatrixForm
DestroyQureg[ψ]
```

```
(       0.25 + 0. i
        0.25 + 0. i
        0.25 + 0. i
        0.25 + 0. i
   0.049818 - 0.184916 i
  0.276642 + 0.0253675 i
  0.0937001 + 0.142666 i
   0.300421 + 0.203788 i
        0.25 + 0. i
        0.25 + 0. i
        0.25 + 0. i
        0.25 + 0. i
  -0.269764 + 0.193368 i
 -0.242995 - 0.0772997 i
-0.00763598 - 0.168026 i
   0.100809 + 0.10768 i     )
```

SimplifyPaulis[]

```
? SimplifyPaulis
```

SimplifyPaulis[expr] freezes commutation and analytically simplifies the given expression of Pauli operators, and expands it in the Pauli basis. The input expression can include sums, products, powers and non–commuting products of (subscripted) X, Y and Z operators and other Mathematica symbols (including variables defined as Pauli expressions).
For example, try SimplifyPaulis[Subscript[Y,0] (a Subscript[X,0] + b Subscript[Z,0] Subscript[X,1])^3].
Be careful of performing algebra with Pauli operators outside of SimplifyPaulis[], since Mathematica may erroneously automatically commute them.

SimplifyPaulis[] simplifies and expands an expression into the basis of Pauli products.

SimplifyPaulis$[\,Y_0\,(a\,X_0 + b\,Z_0\,X_1)^3\,]$

$i\,b\,(a^2+b^2)\,X_0\,X_1 - i\,a\,(a^2-b^2)\,Z_0$

Commutation is disabled within the argument to **SimplifyPaulis[]**, so it is safe to use multiplication (in lieu of non-commuting-multiply **):

SimplifyPaulis$[\,Y_0\,Z_5\,Z_0\,Y_5]$

$X_0\,X_5$

SimplifyPaulis[] can accept expressions which contain variables too (which is more impressive than it seems)!

$\sigma1 = a\,X_0\,Y_{10}\,Z_{50} + b\,Y_0\,Z_{10};$
$\sigma2 = c\,Z_0\,X_{50} + d\,X_{10};$
SimplifyPaulis$[\sigma1\,\sigma2 + \sigma1^3]$

$-i\,b\,d\,Y_0\,Y_{10} + a\,c\,Y_0\,Y_{10}\,Y_{50} + i\,b\,c\,X_0\,X_{50}\,Z_{10} +$
$b\,(-a^2+b^2)\,Y_0\,Z_{10} + a\,(a^2+b^2)\,X_0\,Y_{10}\,Z_{50} - i\,a\,d\,X_0\,Z_{10}\,Z_{50}$

Note if some evaluation *must* happen outside the **SimplifyPaulis[]** environment, then be sure to use non-commuting multiply where it matters (when Pauli operators in a product target the same qubit):

$Z_0 ** Y_0 ** X_0 + b\,X_0\,X_1$
SimplifyPaulis[%]

$Z_0 ** Y_0 ** X_0 + b\,X_0\,X_1$

$-i + b\,X_0\,X_1$

otherwise Mathematica will incorrectly commute the operators:

```
Z₀ Y₀ X₀
SimplifyPaulis[%]
```

$X_0 Y_0 Z_0$

i

DrawCircuit[]

DrawCircuit[] now automatically compactifies circuits, and has been extended to additionally support drawing sub-circuits, schedules and noisy schedules (outputs of **GetCircuitSchedule[]** and **InsertCircuitNoise[]**, which are introduced later)

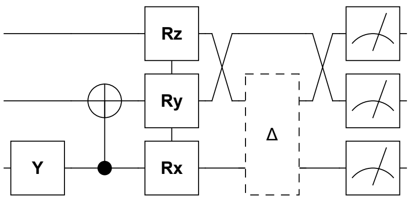
```
? DrawCircuit
```

DrawCircuit[circuit] generates a circuit diagram. The circuit can contain symbolic parameters.
DrawCircuit[circuit, numQubits] generates a circuit diagram with
numQubits, which can be more or less than that inferred from the circuit.
DrawCircuit[{circ1, circ2, ...}] draws the total circuit, divided into the given
subcircuits. This is the output format of GetCircuitColumns[].
DrawCircuit[{{t1, circ1}, {t2, circ2}, ...}] draws the total circuit, divided into the given subcircuits,
labeled by their scheduled times {t1, t2, ...}. This is the output format of GetCircuitSchedule[].
DrawCircuit[{{t1, A1,A2}, {t2, B1,B2}, ...}] draws the total circuit, divided into subcircuits {A1 A2, B1 B2,
...}, labeled by their scheduled times {t1, t2, ...}. This is the output format of InsertCircuitNoise[].
DrawCircuit accepts optional arguments Compactify, DividerStyle, SubcircuitSpacing,
SubcircuitLabels, LabelDrawer and any Graphics option. For example,
the fonts can be changed with 'BaseStyle -> {FontFamily -> "Arial"}'.

Existing functionality

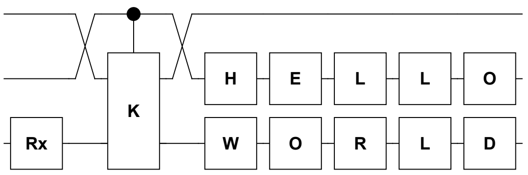
DrawCircuit[] can draw the same circuits that **ApplyCircuit[]** can.

```
u = Circuit[ Y₀ C₀[X₁] R[π, X₀ Y₁ Z₂] Depol₀,₂[.1] M₀,₁,₂];
DrawCircuit[u]
```



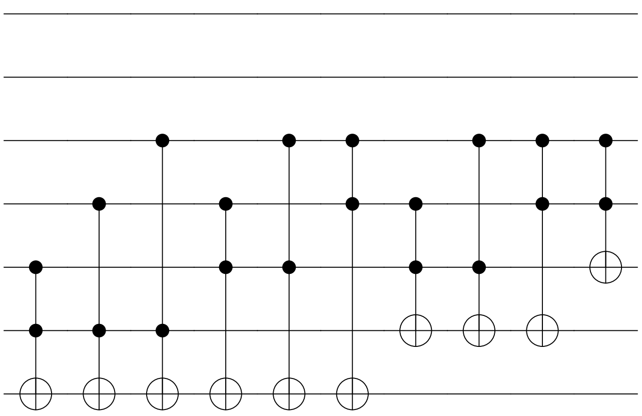
It can additionally handle arbitrary gate symbols, and symbolic parameters:

`u = Circuit[` $Rx_0[\theta]\ C_1[K_{0,2}]\ H_1\ E_1\ L_1\ L_1\ O_1\ W_0\ O_0\ R_0\ L_0\ D_0$`];`
`DrawCircuit[u]`



and overriding qubit number

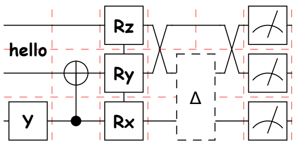
`u = ` $C_{\text{Rest[\#]}}[X_{\text{First[\#]}}]$ ` & /@ Subsets[Range[0, 4], {3}];`
`DrawCircuit[u, 7]`



DrawCircuit[] can accept any optional argument that **Graphics[]** can, to customise the diagram.

`u = Circuit[` $Y_0\ C_0[X_1]\ R[\pi, X_0\ Y_1\ Z_2]\ Depol_{0,2}[.1]\ M_{0,1,2}$`];`

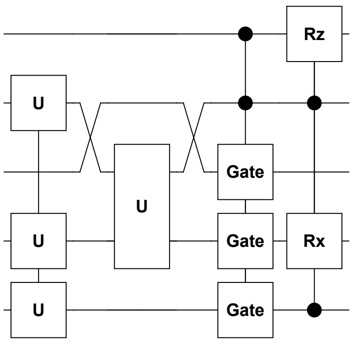
```
DrawCircuit[u,
      BaseStyle → {FontFamily → "Comic Sans MS"},
      ImageSize → 150,
      GridLines → {Range@10, Range@10},
      GridLinesStyle → Directive[Thin, Red, Dashed],
      Epilog → {Text["hello", {.5, 2}]}
]
```



New functionality

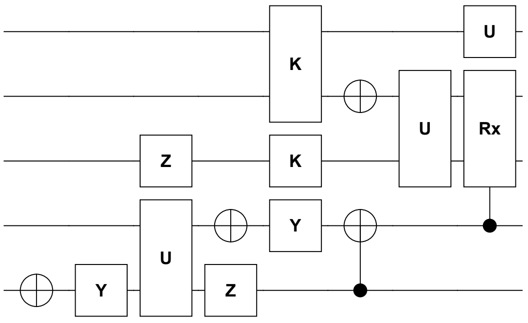
DrawCircuit[] can now draw gates with >2 target qubits, and controlled Pauli gadgets.

DrawCircuit @ Circuit[$U_{0,1,3}$[m] $U_{1,3}$[m] $C_{3,4}$[$Gate_{0,1,2}$] $C_{0,3}$[R[θ, X_1 Z_4]]]

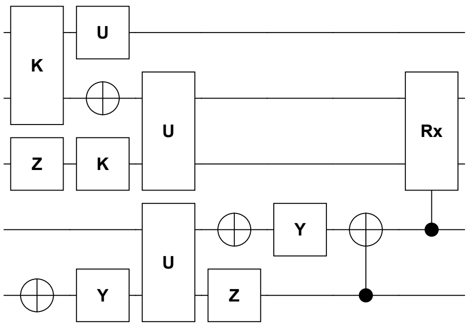


DrawCircuit[] will now attempt to compact the circuit, unless disabled via **Compactify**

u = Circuit[X_0 Y_0 $U_{0,1}$ Z_2 Z_0 X_1 Y_1 K_2 $K_{3,4}$ X_3 C_0[X_1] $U_{2,3}$ C_1[$Rx_{3,2}$[θ]] U_4];
DrawCircuit[u, Compactify → False]

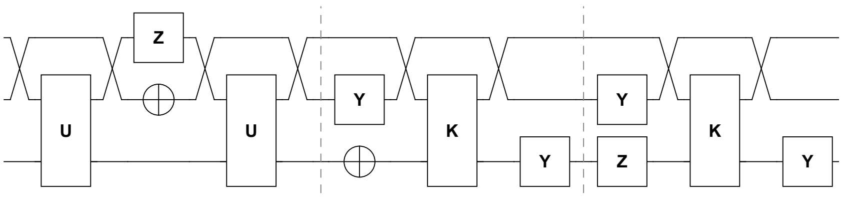


DrawCircuit[u]



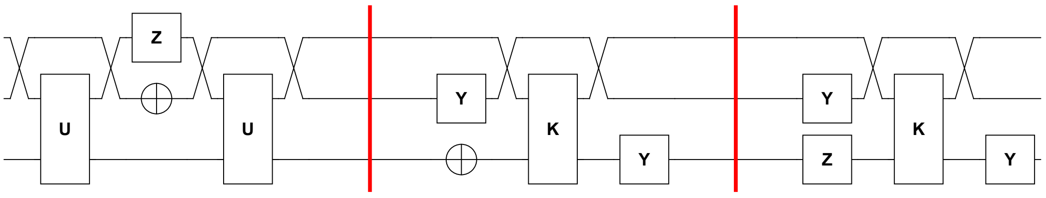
DrawCircuit[] can additionally handle sub-circuits, which will not be compactified together.

circs = {Circuit[$U_{0,2}$ Z_2 X_1 $U_{0,2}$], Circuit[X_0 $K_{0,2}$ Y_0 Y_1], Circuit[Z_0 $K_{0,2}$ Y_0 Y_1]};
DrawCircuit[circs]



The dividing between subcircuits can be customised:

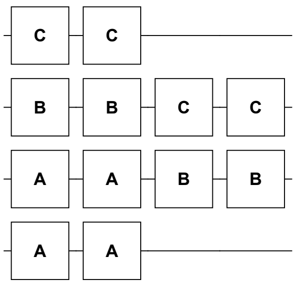
```
DrawCircuit[circs,
     SubcircuitSpacing → 2,
     DividerStyle → Directive[Red, Thick]]
```



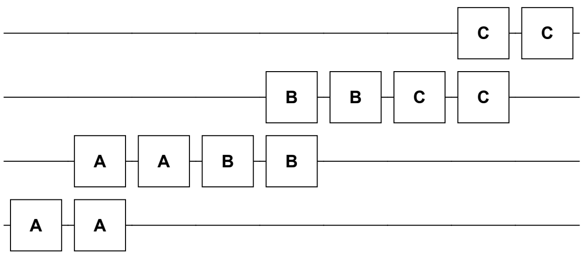
Subcircuits can be used to force gate layouts:

```
u = Circuit[A₀ A₀ A₁ A₁ B₁ B₁ B₂ B₂ C₂ C₂ C₃ C₃];
```

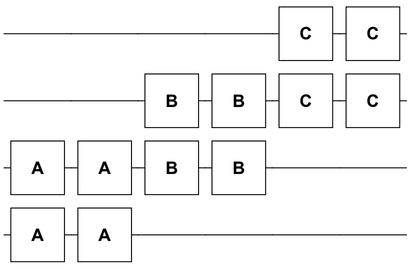
```
DrawCircuit[u]
```



```
DrawCircuit[u, Compactify → False]
```

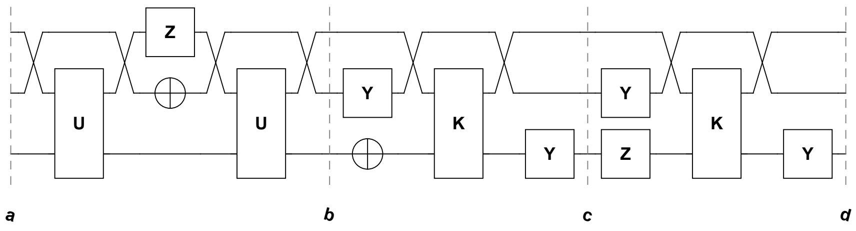


```
DrawCircuit[
     {u⟦1 ;; 4⟧, u⟦5 ;; 8⟧, u⟦9 ;; 12⟧},
     DividerStyle → None, SubcircuitSpacing → 0]
```



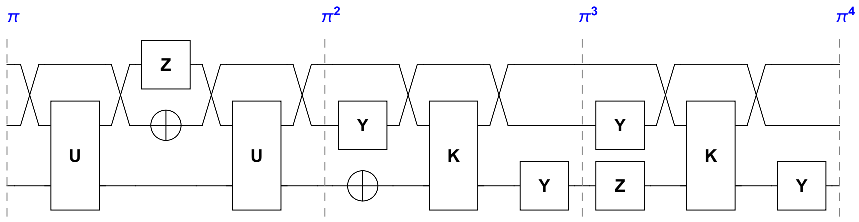
DrawCircuit[] can now even label these sub-circuits:

```
DrawCircuit[circs, SubcircuitLabels → {"a", "b", "c", "d"}]
```



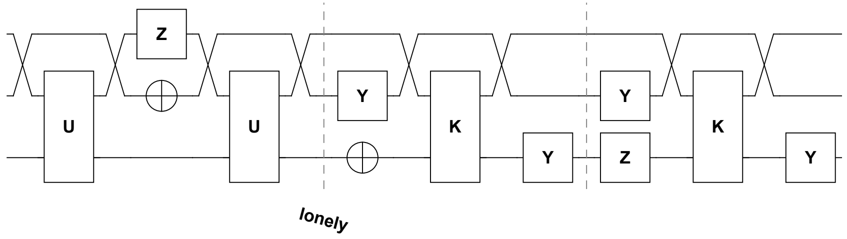
and supports lots of customisation

```
DrawCircuit[circs,
      SubcircuitLabels → Range[4],
      LabelDrawer → Function[{label, x},
        Style[Text[π^label, {x + .1, 3.3}], Blue]]
]
```



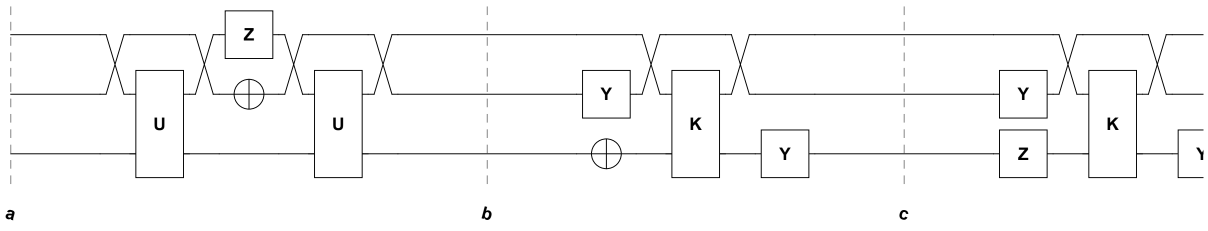
Labels can be skipped using **None**, or by a shorter list. Notice how the left-most and right-most separator lines are not drawn if unlabeled!

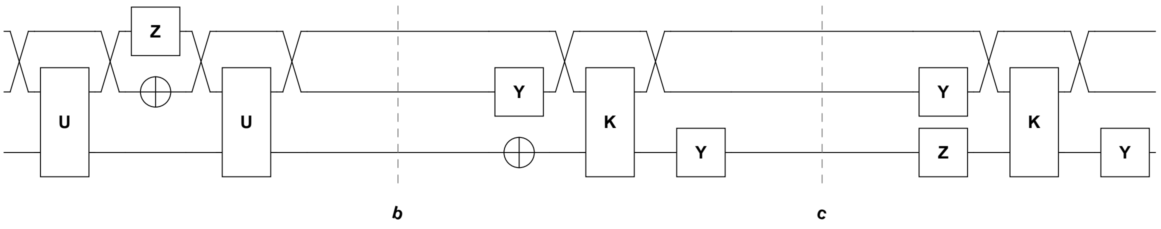
```
DrawCircuit[circs, SubcircuitLabels → {None, "lonely"}]
```



Notice too how carefully **DrawCircuit** extends the qubit lines to meet the left-most and right-most dividers, if displayed

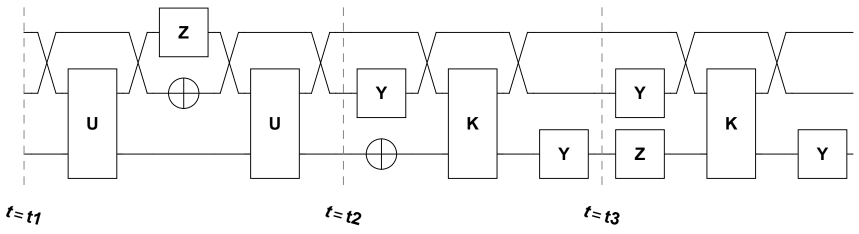
```
DrawCircuit[circs,
     SubcircuitLabels → {"a", "b", "c", "d"},
     SubcircuitSpacing → 3]
DrawCircuit[circs,
     SubcircuitLabels → {None, "b", "c", None},
     SubcircuitSpacing → 3]
```





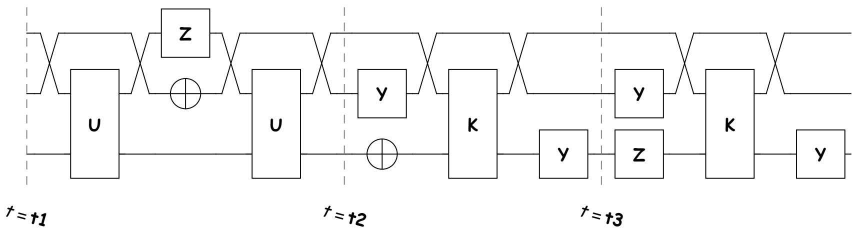
DrawCircuit[] can also accept a *schedule* and will automatically display the labels as timestamps (unless overridden). This is the output format of **GetCircuitSchedule[]**

```
DrawCircuit[{
      {t1, Circuit[ U₀,₂ Z₂ X₁ U₀,₂]},
      {t2, Circuit[X₀ K₀,₂ Y₀ Y₁]},
      {t3, Circuit[Z₀ K₀,₂ Y₀ Y₁]}
 }]
```



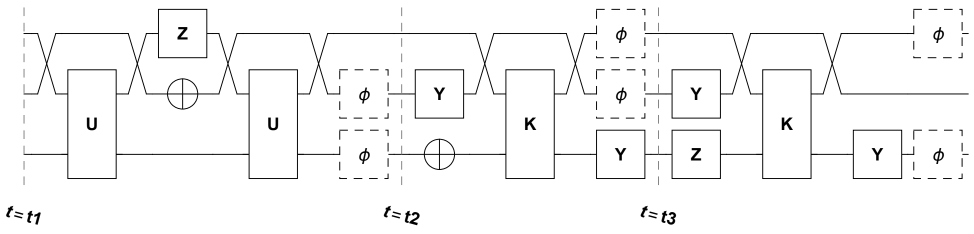
Note that these labels are also affected by the **Graphics[]** style.

```
DrawCircuit[{
      {t1, Circuit[ U₀,₂ Z₂ X₁ U₀,₂]},
      {t2, Circuit[X₀ K₀,₂ Y₀ Y₁]},
      {t3, Circuit[Z₀ K₀,₂ Y₀ Y₁]}},
     BaseStyle → {FontFamily → "Comic Sans MS"}
]
```



DrawCircuit[] can also accept a *noisy schedule* in a similar manner. This is the output format of **InsertCircuitNoise[]**

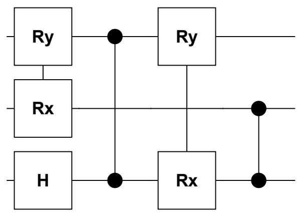
```
DrawCircuit[{
      {t1, Circuit[ U₀,₂ Z₂ X₁ U₀,₂], Circuit[Deph₀[.1] Deph₁[.1]]},
      {t2, Circuit[X₀ K₀,₂ Y₀ Y₁], Circuit[Deph₁[.1] Deph₂[.1]]},
      {t3, Circuit[Z₀ K₀,₂ Y₀ Y₁], Circuit[Deph₀[.1] Deph₂[.1]]}}
]
```

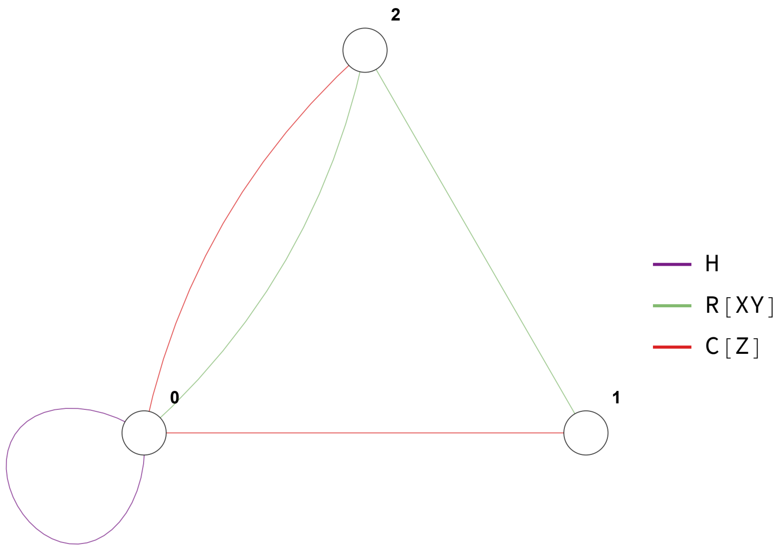


DrawCircuitTopology[]

DrawCircuitTopology[] draws the qubit connectivity of a circuit.

```
u = Circuit[ H₀ R[θ, X₁ Y₂] C₂[Z₀] R[ϕ, X₀ Y₂] C₀[Z₁]];
DrawCircuit[u]
DrawCircuitTopology[u]
```



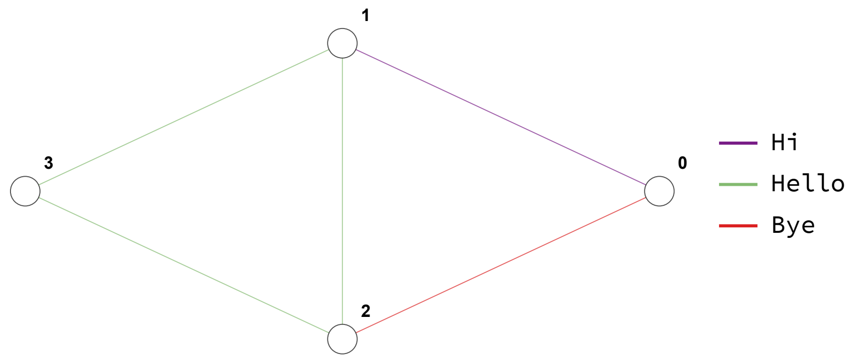


```
? DrawCircuitTopology
```

DrawCircuitTopology[circuit] generates a graph plot of the qubit connectivity implied by the given circuit. The precise nature of the information plotted depends on the following options.
DrawCircuitTopology accepts optional arguments DistinguishBy, ShowLocalGates, ShowRepetitions to modify the presented graph.
DrawCircuitTopology additionally accepts DistinguishedStyles and all options of Graph[], Show[] and LineLegend[] for customising the plot aesthetic.

DrawCircuitTopology makes no distinction between canonical and custom gates:

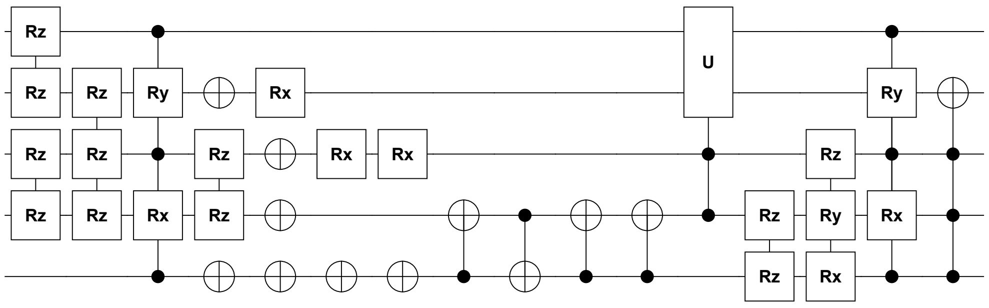
```
DrawCircuitTopology @ Circuit[ Hi$_{0,1}$ Hello$_{1,2,3}$[θ] Bye$_{0,2}$ ]
```



How gates in the circuit are distinguished (as edges) is controlled primarily by the optional argument **DistinguishBy**. Consider example:

$u = \text{Circuit}[R[\pi, Z_1 Z_2]\ R[\pi, Z_3 Z_4]\ R[\pi, Z_1 Z_3 Z_2]\ C_{0,2,4}[R[\pi, X_1 Y_3]]\ R[.1, Z_1 Z_2]$
$\quad X_0 X_0 X_0 X_1 X_2 X_3 X_0\ C_0[X_1]\ C_1[X_0]\ Rx_2[.1]\ Rx_2[\phi]\ Rx_3[\phi]\ C_0[X_1]\ C_0[X_1]$
$\quad C_{1,2}[\ U_{3,4}[\omega]\ ]\ Rz_{0,1}[1]\ R[\phi,\ X_0 Y_1 Z_2]\ C_{0,2,4}[R[\pi, X_1 Y_3]]\ C_{0,1,2}[X_3]];$

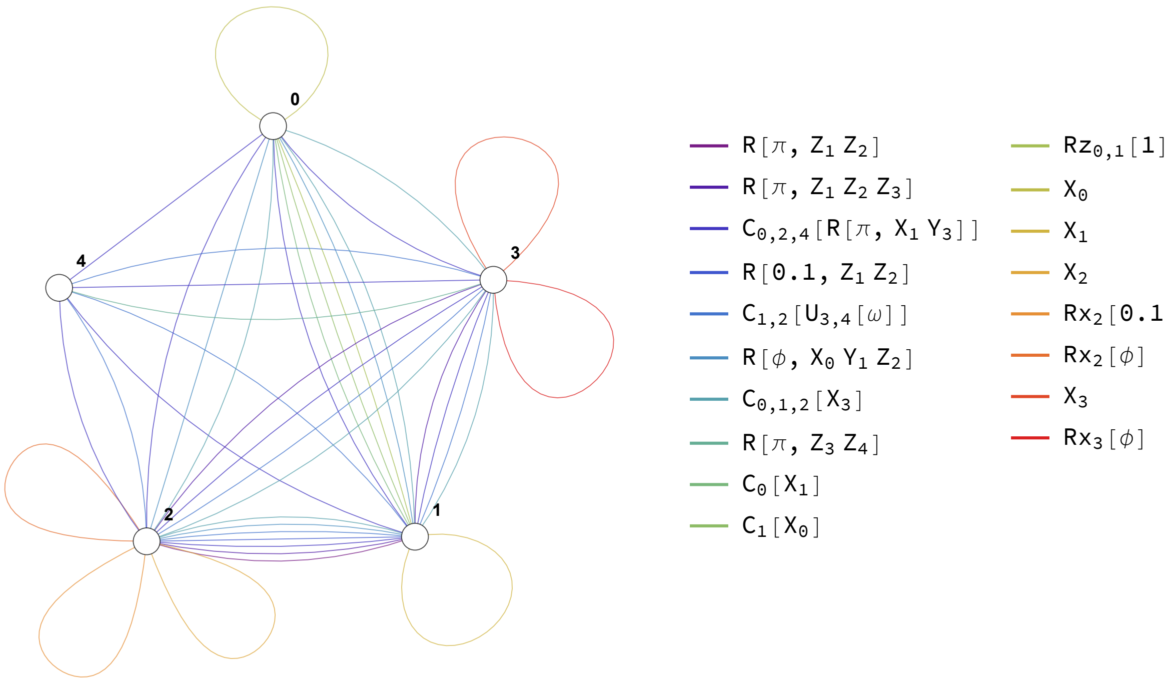
```
DrawCircuit[u]
```



We demonstrate the choices of **DistinguishBy** in decreasing specificity / granularity.

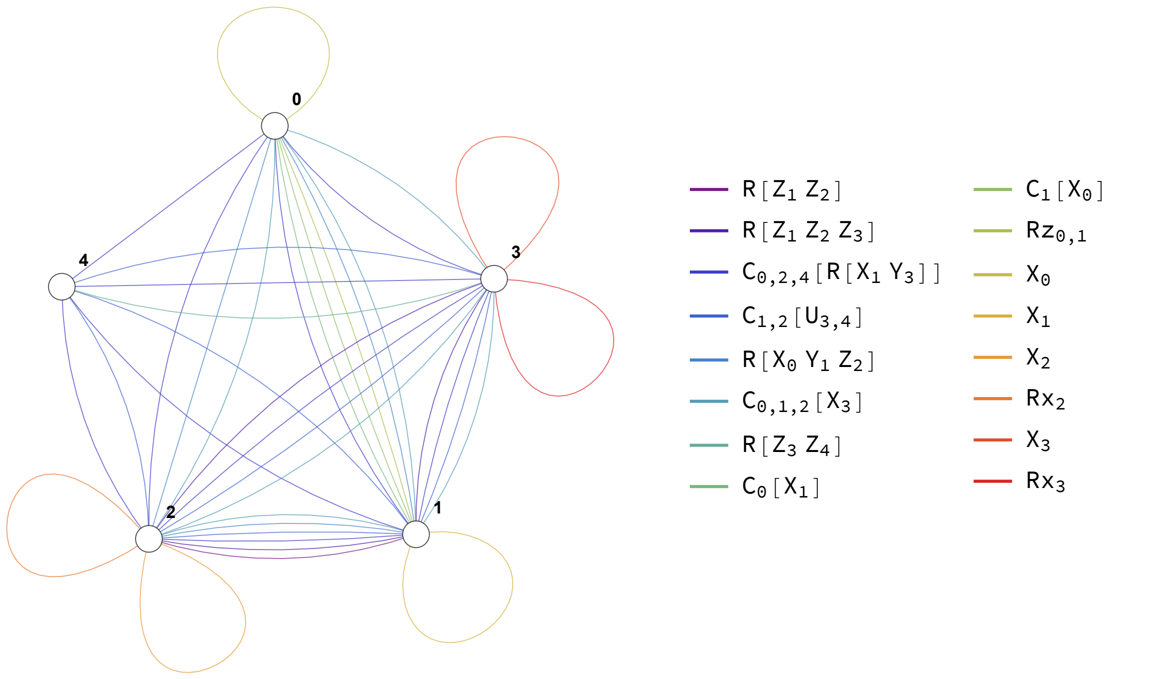
DistinguishBy → “Parameters” distinguishes every unique gate (considering gate family, qubits and parameters) in the circuit into its own graph edge. This is the most granular option. Below, notice that **Rx_2[0.1]** and **$Rx_2[\phi]$** are assigned separate edge colours, because they differ in parameter.

```
DrawCircuitTopology[u, DistinguishBy → "Parameters"]
```



DistinguishBy → "Qubits" distinguishes gates by only their family/symbol and targeted qubits. Notice that the **Rx_2[0.1]** and **$Rx_2[\phi]$** gates are merged into one edge label, though the **$Rx_3[\phi]$** gate remains separate.

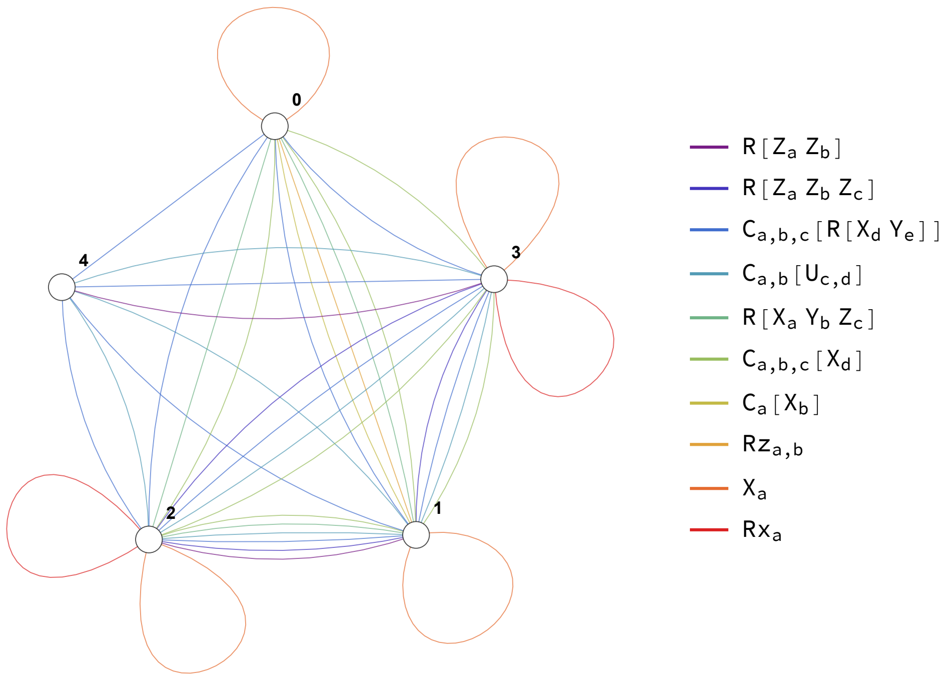
```
DrawCircuitTopology[u, DistinguishBy → "Qubits"]
```



Notice too that $R[\pi,\ Z_1 Z_2]$ and $R[\pi,\ Z_3 Z_4]$ are given separate edge labels, since they operate upon different qubits. The label excludes the parameter π, since the same gate with a different parameter would be merged with this setting.

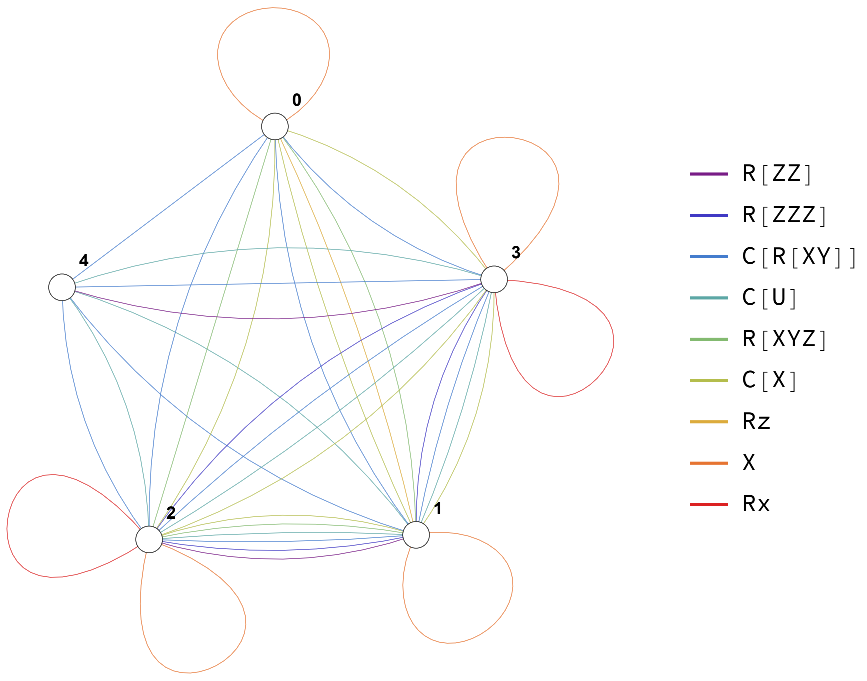
DistinguishBy → "NumberOfQubits" distinguishes gates by only their family/symbol and the *number* of qubits they target (or are controlled by). In this example, $R[\pi,\ Z_1 Z_2]$ and $R[\pi,\ Z_3 Z_4]$ become the same edge, labeled $R[Z_a Z_b]$.

```
DrawCircuitTopology[u, DistinguishBy → "NumberOfQubits"]
```



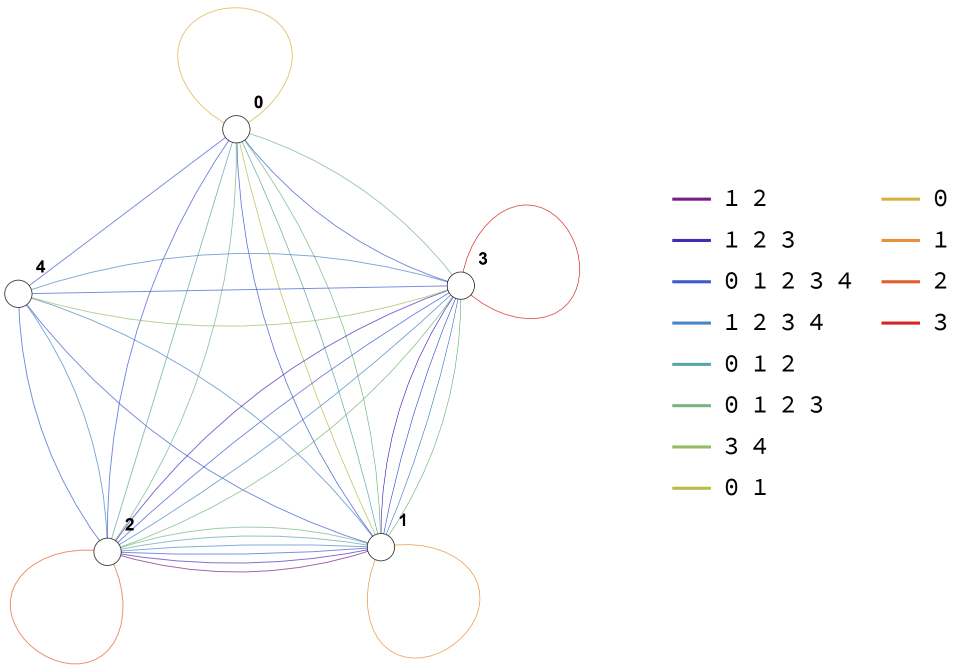
DistinguishBy → "Gates" distinguishes gates entirely by their family / operator form, regardless of the number of controlled or targeted qubits. For example, while a controlled and uncontrolled gate remain distinct, otherwise equivalent gates with 1 and 2 controls are labeled together. Notice that $C_{a,b,c}[X_d]$ and $C_a[X_b]$ above have been merged into the single label $C[X]$. Notice too that Pauli gadgets with differing numbers of Pauli operators remain distinct.

```
DrawCircuitTopology[u, DistinguishBy → "Gates"]
```



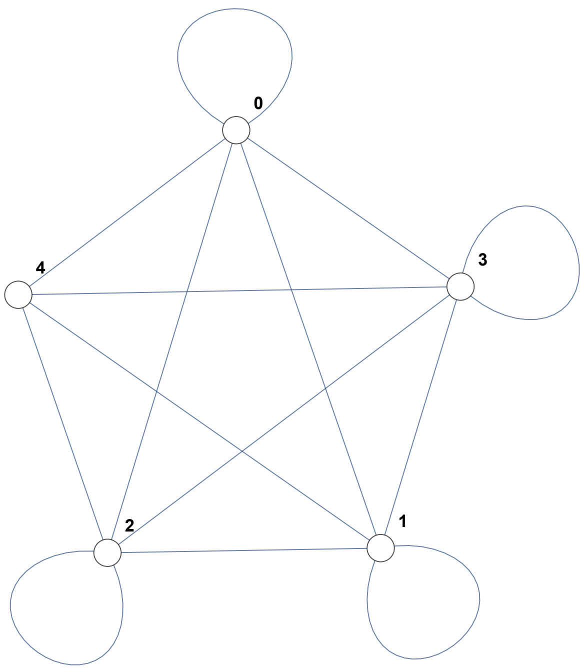
DistinguishBy → "Connectivity" disregards all information about a gate *except* the set of qubits it operates upon, through its control and target qubits.

```
DrawCircuitTopology[u, DistinguishBy → "Connectivity"]
```



DistinguishBy → "None" does not distinguish gates in any way. An edge will exist between a pair of qubits if they are involved together in any gate (which may include additional qubits) in the given circuit. The example below informs us that qubit **4** was not targeted by any single-qubit gates.

```
DrawCircuitTopology[u, DistinguishBy → "None"]
```



```
? DistinguishBy
```

Optional argument to DrawCircuitTopology to specify how gates are aggregated into graph edges and legend labels. The possible values (in order of decreasing specificity) are "Parameters", "Qubits", "NumberOfQubits", "Gates", "None", and a distinct "Connectivity" mode.
DistinguishBy –> "Parameters" assigns every unique gate (even distinguishing similar operators with different parameters) its own label.
DistinguishBy –> "Qubits" discards gate parameters, but respects target qubits, so will assign similar gates (acting on the same qubits) but with different parameters to the same label.
DistinguishBy –> "NumberOfQubits" discards gate qubit indices, but respects the number of qubits in a gate. Hence, for example, similar gates controlled on different pairs of qubits will be merged together, but not with the same gate controlled on three qubits.
DistinguishBy –> "Gates" respects only the gate type (and whether it is controlled or not), and discards all qubit and parameter information. Hence similar gates acting on different numbers of qubits will be merged to one label. This does not apply to pauli–gadget gates R, which remain distinguished for unique pauli sequences (though discarding qubit indices).
DistinguishBy –> "None" performs no labelling or distinguishing of edges.
DistinguishBy –> "Connectivity" merges all gates, regardless of type, acting upon the same set of qubits (orderless).

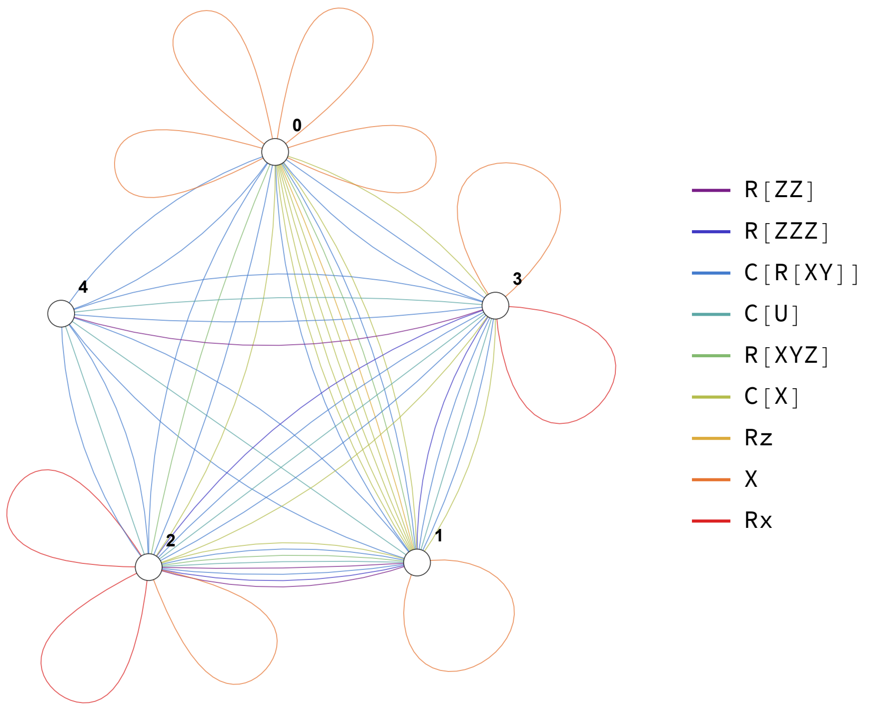
In the above examples, between any given pair of qubits, there was at most one edge of a certain kind/colour. This was regardless of how many times a gate of that kind was present in the circuit.

We can instead opt to display *multiple* edges of a kind between qubits, to indicate repetition in the circuit, by **ShowRepetitions**.

```
? ShowRepetitions
```

Optional argument to DrawCircuitTopology, to specify (True or False) whether repeated instances of gates (or other groups as set by DistinguishBy) in the circuit should each yield a distinct edge.
For example, if ShowRepetitions –> True and DistinguishBy –> "Qubits", then a circuit containing three C[Rz] gates between qubits 0 and 1 will produce a graph with three edges between vertices 0 and 1.

```
DrawCircuitTopology[u, ShowRepetitions → True]
```



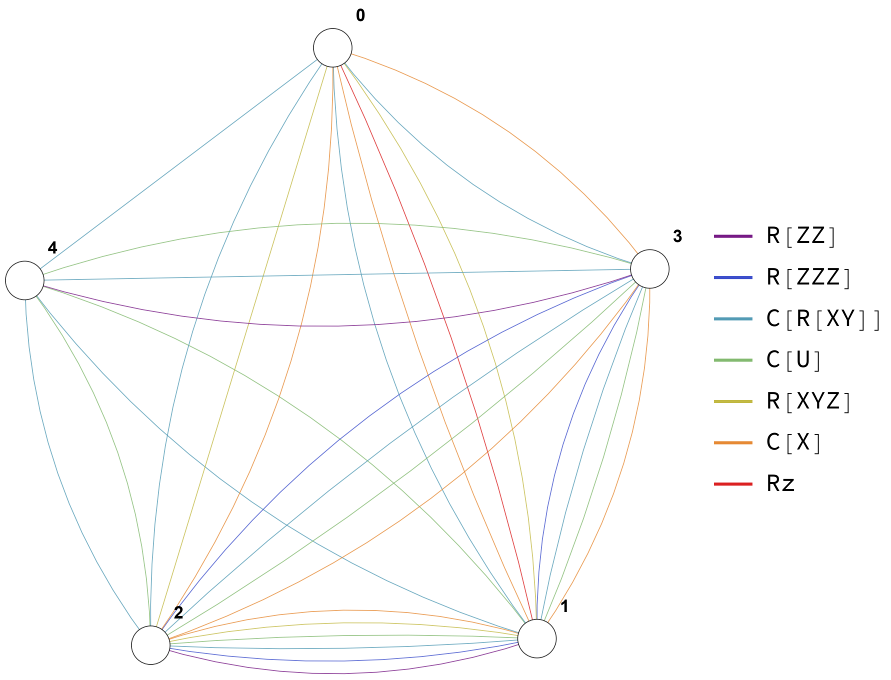
This allows us to see, for example, that circuit **u** contained *four* **Rz** gates acting on qubit **0**.

We can furthermore exclude single-qubit gates from the graph using **ShowLocalGates -> False**

```
? ShowLocalGates
```

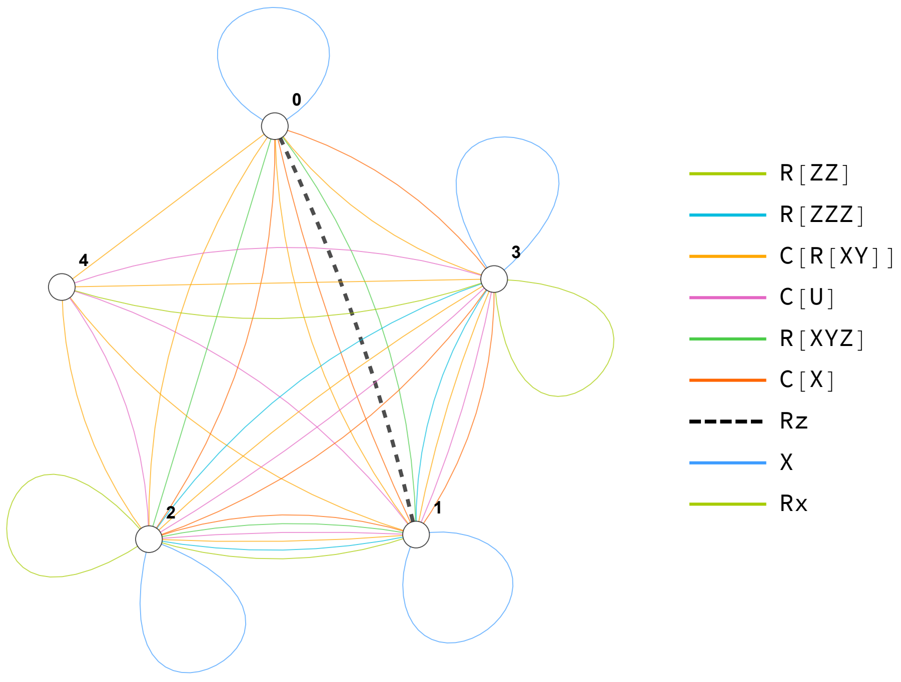
Optional argument to DrawCircuitTopology, to specify (True or False) whether single–qubit gates should be included in the plot (as single–vertex loops).

```
DrawCircuitTopology[u, ShowLocalGates → False]
```



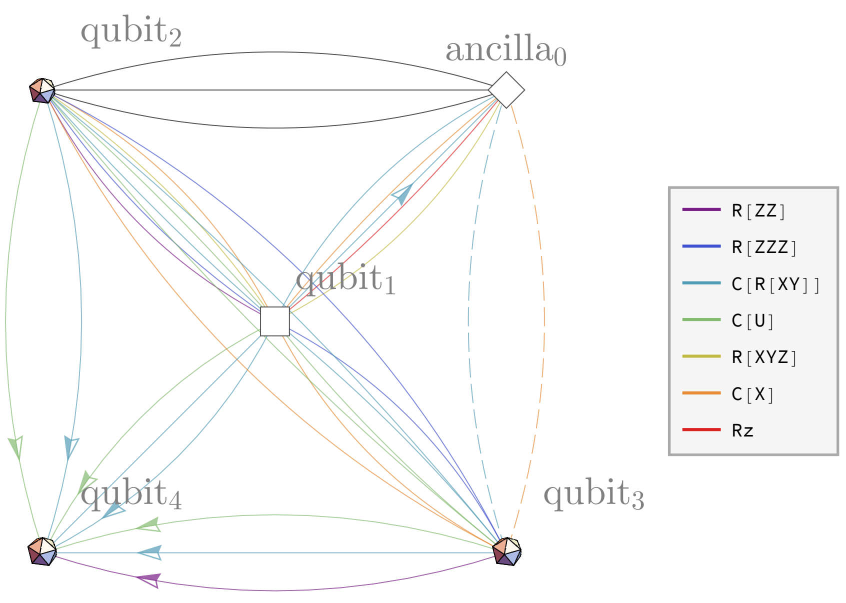
We can customise the colour and style of the distinguished edges, using **DistinguishedStyles**. If we provide too few styles, it will simply repeat from the first.

```
DrawCircuitTopology[u, DistinguishedStyles → {
        🟩, 🟦, 🟧, 🟪, 🟩, 🟥, Directive[Black, Thick, Dashed], 🟦}]
```



We can also customise and override the plot aesthetic directly, using all optional arguments for **Graph[]**, **Show[]** and **LineLegend**.

```
DrawCircuitTopology[u, ShowLocalGates → False,

     (* spread out the vertices *)
     GraphLayout → "BalloonEmbedding",

     (* override the edges between vertices 0 and 2 to be black *)
     EdgeStyle → {(0 ⟷ 2) → Black},

     (* override the shapes of edges between 0 and 3,
 and all edges to/from 4 *)
     EdgeShapeFunction → {(0 ⟷ 3) → "DashedLine", (_ ⟷ 4) → "CarvedArrow"},

     (* give vertices 0 and 1 unique shapes *)
     VertexShapeFunction → {0 → "Diamond", 1 → "Square"},
     (* and make all other vertices a polyhedron *)
     VertexShape → ,

     (* change the style of all vertex labels *)
     VertexLabelStyle → Directive[Gray, 20, FontFamily → "CMU Serif"],

     (* and the vertex labels themselves; give vertex 0 a unique label *)
     VertexLabels → {i_ → "qubit"_i, 0 → Placed["ancilla_0", Above]},

     (* change the style of the legend *)
     LegendFunction → "Panel"
]
```



Notice that edges are identified using **UndirectedEdge** (shortcut: ESC u e ESC) within parenthesis, and many edges can be represented at once using concise patterns

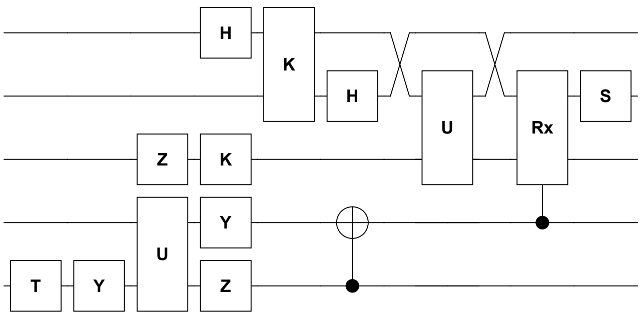
GetCircuitColumns[]

```
? GetCircuitColumns
```

GetCircuitColumns[circuit] divides circuit into sub–circuits of gates on unique qubits (i.e. columns), filled from the left. Flatten the result to restore an equivalent but potentially compacted Circuit.

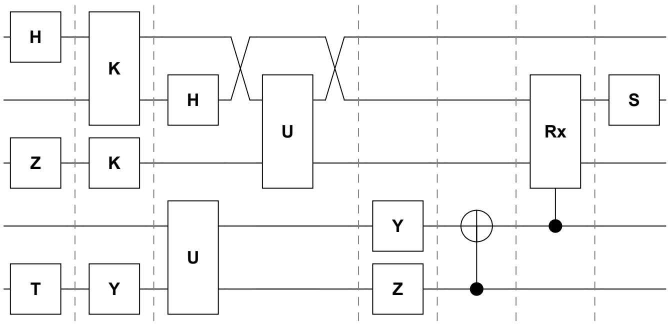
```
u = Circuit[ T0 Y0 U0,1 Z2 Z0 H4 Y1 K2 K3,4 H3 C0[X1] U2,4 C1[Rx3,2[θ]] S3]
DrawCircuit[u, Compactify → False]
```

$\{T_0, Y_0, U_{0,1}, Z_2, Z_0, H_4, Y_1, K_2, K_{3,4}, H_3, C_0[X_1], U_{2,4}, C_1[Rx_{3,2}[\theta]], S_3\}$



```
GetCircuitColumns[u]
DrawCircuit[%]
```

$\{\{T_0, Z_2, H_4\}, \{Y_0, K_2, K_{3,4}\}, \{U_{0,1}, H_3, U_{2,4}\}, \{Z_0, Y_1\}, \{C_0[X_1]\}, \{C_1[Rx_{3,2}[\theta]]\}, \{S_3\}\}$



ViewCircuitSchedule[]

```
? ViewCircuitSchedule
```

ViewCircuitSchedule[schedule] displays a table form of the given circuit schedule, as output by InsertCircuitNoise[] or GetCircuitSchedule[].
ViewCircuitSchedule accepts all optional arguments of Grid[], for example 'FrameStyle', and 'BaseStyle -> {FontFamily -> "CMU Serif"}'.

ViewCircuitSchedule is a convenient way to view the times and parameters in a circuit schedule. This is most useful for viewing the results of **GetCircuitSchedule[]** and **InsertCircuitNoise**[] as introduced later.

```
ViewCircuitSchedule[{
   {0, Circuit[ Rx_2[8.5 π] ]},
   {4, Circuit[Ry_0[.1] C_0[Rx_1[.2]]]},
   {15, Circuit[ Rx_1[.3]]},
   {16, Circuit[ Ry_0[.1] C_0[Rx_2[.4]]]}
 }]
```

time	gates
0	$Rx_2[26.7035]$
4	$Ry_0[0.1]\ C_0[Rx_1[0.2]]$
15	$Rx_1[0.3]$
16	$Ry_0[0.1]\ C_0[Rx_2[0.4]]$

It can handle symbolic expressions, and explicit specification of {{time, {active noise}, {passive noise}} ...}.

```
ViewCircuitSchedule[{
   {t0,
        {Rx_2[θ], Ry_0[ϕ]},
        {Deph_2[0.1], Depol_{0,1}[2 δ]}},
   {t0 + Δt,
        {C_0[Rx_1[0.2]]},
        {Depol_0[δ], Deph_1[δ], Deph_2[δ], Deph_3[δ], Deph_4[δ]}},
   {Exp[π^4],
        {Rx_1[0.3], Ry_0[π/2]},
        {Deph_1[π/10], Depol_{0,1}[α]}},
   {t3,
        {C_0[Rx_2[θ^2]]},
        {Depol_{0,2}[α]}}
 }]
```

time	active noise	passive noise
t0	$Rx_2[\theta]\ Ry_0[\phi]$	$Deph_2[0.1]\ Depol_{0,1}[2\,\delta]$
t0 + Δt	$C_0[Rx_1[0.2]]$	$Depol_0[\delta]\ Deph_1[\delta]\ Deph_2[\delta]\ Deph_3[\delta]\ Deph_4[\delta]$
e^{π^4}	$Rx_1[0.3]\ Ry_0\left[\frac{\pi}{2}\right]$	$Deph_1\left[\frac{\pi}{10}\right]\ Depol_{0,1}[\alpha]$
t3	$C_0\left[Rx_2\left[\theta^2\right]\right]$	$Depol_{0,2}[\alpha]$

ViewCircuitSchedule can be customised with all options accepted by **Grid[]**

```
ViewCircuitSchedule[{
  {t₀, Circuit[ Rx₂[8.5 π] ]},
  {t₁, Circuit[Ry₀[.1] C₀[Rx₁[.2]]]}},

 BaseStyle → {FontFamily → "Comic Sans MS"},
 FrameStyle → Blue
]
```

time	gates
t_0	$Rx_2[26.7035]$
t_1	$Ry_0[0.1]\ C_0[Rx_1[0.2]]$

Device specifications

A device specification is an **Association** with specific keys, which represents a realistic hardware device. It describes the gate and qubit constraints of the device, how long gates take to apply, the noise channels induced by imperfect attempts to perform each gate, and the passive noise channels on inactive qubits. A device specification can be used to transform an ideal circuit description (one compatible with perfect state-vector simulation) into a schedule of gates, and/or a realistic noisy channel description (compatible with density matrix simulation).

Device specifications can capture nuances about a hardware device like:
- qubit connectivity constraints
- supported gates, with parameter and qubit constraints
- bespoke operations, like qubit initialisation or non-canonical unitaries
- global time-dependent active and passive noise processes
- advanced noise processes like cross-talk, or channels dependent on variables updated through a circuit evaluation
- the duration to effect gates, which may be qubit or parameter dependent

Device specifications can range from very simple to very complicated, depending on the nuance of the represented hardware device. For a thorough guide on *creating* device specifications (and to understand their syntax), see **guide_creating_device_spec.nb.** Below is an example of a device spec

```
myDevSpec = Module[{Δt},
<|
    DeviceDescription → "Five funky qubits.",
    NumAccessibleQubits → 5,
    NumTotalQubits → 5,

    Aliases → {
        (* Supported gates with a general matrix description must be aliased *)
```

$$A_{q\_}[\theta\_] :\rightarrow \text{Circuit}\left[\ U_q\left[\frac{1}{\sqrt{2}}\begin{pmatrix} e^{-i\ \theta}\ (\text{Cos}[\theta]-\text{Sin}[\theta]) & \text{Cos}[\theta]+\text{Sin}[\theta] \\ \text{Cos}[\theta]+\text{Sin}[\theta] & e^{i\ \theta}\ (-\text{Cos}[\theta]+\text{Sin}[\theta]) \end{pmatrix}\right]\ \right],$$

```
        (* Aliases can resolve to a sequence of gates, including other aliases *)
        B_{q1_,q2_} :→ Circuit[ A_{q1}[π/3] A_{q2}[π/3] SWAP_{q1,q2} A_{q1}[-π/3] A_{q2}[-π/3] ],

        (* Aliases are useful for declaring qubit preparations (here, set a qubit to |
        Init_{q_} :→ Circuit[ Damp_{q}[1] H_{q} ]
    },

    Gates → {

        (* Allow Hadamards on every qubit *)
        H_{q_} :→ <|
            (* which cause a small dephasing *)
            NoisyForm → Circuit[ H_{q} Deph_{q}[.01] ],
            (* and requires a duration of 1 unit of time *)
            GateDuration → 1
        |>,

        (* Allow Rx, Ry, Rz gates (when angle in (0,π)) on every qubit *)
        (r:Rx|Ry|Rz)_{q_}[θ_] /; (0 < θ < π) :→ <|
            (* which causes depolarising then a small over-rotation *)
            NoisyForm → Circuit[ Depol_{q}[.01] r_{q}[θ + .01] ],
            (* and requires a duration dependent on the parameter *)
            GateDuration → θ
        |>,

        (* Allow controlling Rx gates by qubits to the right of the target *)
        C_{c_}[Rx_{q_}[θ_]] /; (q > c) :→ <|
            NoisyForm → Circuit[ C_{c}[Rx_{q}[θ]] Deph_{c,q}[.01] ],
            GateDuration → 2 θ
        |>,

        (* Allow A (alias) gates only on even-index qubits, with parameter ≤ π/3 *)
        A_{q_}[θ_ /; 0 < θ ≤ π/3] /; ( EvenQ[q] ) :→ <|
            NoisyForm → Circuit[ A_{q}[θ] Damp_{q}[θ/100] ],
            (* with a duration dependent on the target qubit *)
            GateDuration → 1 + q
        |>,

        (* Allow B gates only on pairs of odd and even qubit indices *)
        B_{q1_,q2_} /; ( OddQ @ Abs[q1-q2] ) :→ <|
            NoisyForm → Circuit[ A_{q1}[π/10] Depol_{q1,q2}[.1] B_{q1,q2}  ],
            GateDuration → 5
        |>,

        (* Allow perfect qubit initialisation *)
        Init_{q_} :→ <|
            NoisyForm → Circuit[ Init_{q} ],
            GateDuration → 1
            |>
    },

    DurationSymbol → Δt,
    Qubits → {
        q_ :→ <|
            (* idle qubits dephase and "get A'd" depending on duration *)
            PassiveNoise → Circuit[ Deph_q[Δt/1000] A_q[Δt] ]
        |>
    }
|>];
```

A device specification is accepted by a variety of new QuESTlink facilities, which include converting a circuit into a schedule via the device’s gate durations (**GetCircuitSchedule[]**), mapping a noise-free circuit to a realistic noisy circuit for density matrix simulation (**InsertCircuitNoise[]**), determining the conditions under which a candidate schedule is compatible with a device (**CheckCircuitSchedule[]**, **GetUnsupportedGates[]**) and visualising devices (**ViewDeviceSpec[]**).

ViewDeviceSpec[]

```
? ViewDeviceSpec
```

ViewDeviceSpec[spec] displays all information about the given device specification in table form.
ViewDeviceSpec accepts all optional arguments of Grid[]
    (to customise all tables), and Column[] (to customise their placement).

```
ViewDeviceSpec[myDevSpec]
```

Fields	
Number of accessible qubits	5
Number of hidden qubits	0
Number of qubits (total)	5
Duration symbol	Δt
Description	Five funky qubits.

Aliases	
Operator	Definition
$A_{q\_}[\theta\_]$	$U_q\left[\begin{pmatrix} \frac{e^{-i\theta}(\text{Cos}[\theta]-\text{Sin}[\theta])}{\sqrt{2}} & \frac{\text{Cos}[\theta]+\text{Sin}[\theta]}{\sqrt{2}} \\ \frac{\text{Cos}[\theta]+\text{Sin}[\theta]}{\sqrt{2}} & \frac{e^{i\theta}(-\text{Cos}[\theta]+\text{Sin}[\theta])}{\sqrt{2}} \end{pmatrix}\right]$
$B_{q1\_,q2\_}$	$A_{q1}\left[\frac{\pi}{3}\right]\ A_{q2}\left[\frac{\pi}{3}\right]\ \text{SWAP}_{q1,q2}\ A_{q1}\left[-\frac{\pi}{3}\right]\ A_{q2}\left[-\frac{\pi}{3}\right]$
$\text{Init}_{q\_}$	$\text{Damp}_q[1]\ H_q$

Gates			
Gate	Conditions	Noisy form	Duration (Δt)
$H_{q\_}$		H_q $\text{Deph}_q[0.01]$	1
$(r : \text{Rx} \mid \text{Ry} \mid \text{Rz}_{q\_})[\theta\_]$	$0 < \theta < \pi$	$\text{Depol}_q[0.01]$ $r_q[0.01 + \theta]$	θ
$C_{c\_}[\text{Rx}_{q\_}[\theta\_]]$	$q > c$	$C_c[\text{Rx}_q[\theta]]$ $\text{Deph}_{c,q}[0.01]$	$2\,\theta$
$A_{q\_}[\theta\_]$	$0 < \theta \leq \frac{\pi}{3}$ EvenQ[q]	$A_q[\theta]$ $\text{Damp}_q\left[\frac{\theta}{100}\right]$	$1 + q$
$B_{q1\_,q2\_}$	OddQ[Abs[q1 - q2]]	$A_{q1}\left[\frac{\pi}{10}\right]$ $\text{Depol}_{q1,q2}[0.1]$ $B_{q1,q2}$	5
$\text{Init}_{q\_}$		Init_q	1

Qubits	
Qubit	Passive noise
0	$\text{Deph}_0\left[\frac{\Delta t}{1000}\right]$ $A_0[\Delta t]$
1	$\text{Deph}_1\left[\frac{\Delta t}{1000}\right]$ $A_1[\Delta t]$
2	$\text{Deph}_2\left[\frac{\Delta t}{1000}\right]$ $A_2[\Delta t]$
3	$\text{Deph}_3\left[\frac{\Delta t}{1000}\right]$ $A_3[\Delta t]$
4	$\text{Deph}_4\left[\frac{\Delta t}{1000}\right]$ $A_4[\Delta t]$

The appearence can be controlled by all options to Grid[] and Column[]

```
ViewDeviceSpec[myDevSpec,
    BaseStyle → {FontFamily → "Comic Sans MS"},
    Background → LightBlue,
    FrameStyle → White]
```

Fields	
Number of accessible qubits	5
Number of hidden qubits	0
Number of qubits (total)	5
Duration symbol	Δt
Description	Five funky qubits.

Aliases	
Operator	Definition
$A_{q\_}[\theta\_]$	$U_q\left[\begin{pmatrix} \frac{e^{-i\theta}(\text{Cos}[\theta]-\text{Sin}[\theta])}{\sqrt{2}} & \frac{\text{Cos}[\theta]+\text{Sin}[\theta]}{\sqrt{2}} \\ \frac{\text{Cos}[\theta]+\text{Sin}[\theta]}{\sqrt{2}} & \frac{e^{i\theta}(-\text{Cos}[\theta]+\text{Sin}[\theta])}{\sqrt{2}} \end{pmatrix}\right]$
$B_{q1\_,q2\_}$	$A_{q1}\left[\frac{\pi}{3}\right] A_{q2}\left[\frac{\pi}{3}\right] \text{SWAP}_{q1,q2} A_{q1}\left[-\frac{\pi}{3}\right] A_{q2}\left[-\frac{\pi}{3}\right]$
$\text{Init}_{q\_}$	$\text{Damp}_q[1]\ H_q$

Gates			
Gate	Conditions	Noisy form	Duration (Δt)
$H_{q\_}$		H_q $\text{Deph}_q[0.01]$	1
$(r : \text{Rx} \mid \text{Ry} \mid \text{Rz}_{q\_})[\theta\_]$	$0 < \theta < \pi$	$\text{Depol}_q[0.01]$ $r_q[0.01 + \theta]$	θ
$C_{c\_}[\text{Rx}_{q\_}[\theta\_]]$	$q > c$	$C_c[\text{Rx}_q[\theta]]$ $\text{Deph}_{c,q}[0.01]$	$2\,\theta$
$A_{q\_}[\theta\_]$	$0 < \theta \leq \frac{\pi}{3}$ EvenQ[q]	$A_q[\theta]$ $\text{Damp}_q\left[\frac{\theta}{100}\right]$	$1 + q$
$B_{q1\_,q2\_}$	OddQ[Abs[q1 − q2]]	$A_{q1}\left[\frac{\pi}{10}\right]$ $\text{Depol}_{q1,q2}[0.1]$ $B_{q1,q2}$	5
$\text{Init}_{q\_}$		Init_q	1

Qubits	
Qubit	Passive noise
0	$\text{Deph}_0\left[\frac{\Delta t}{1000}\right]$ $A_0[\Delta t]$
1	$\text{Deph}_1\left[\frac{\Delta t}{1000}\right]$ $A_1[\Delta t]$
2	$\text{Deph}_2\left[\frac{\Delta t}{1000}\right]$ $A_2[\Delta t]$
3	$\text{Deph}_3\left[\frac{\Delta t}{1000}\right]$ $A_3[\Delta t]$
4	$\text{Deph}_4\left[\frac{\Delta t}{1000}\right]$ $A_4[\Delta t]$

Individual **Grid**s can also be directly accessed

```
ViewDeviceSpec[myDevSpec]⟦1, 1⟧
```

Fields	
Number of accessible qubits	5
Number of hidden qubits	0
Number of qubits (total)	5
Duration symbol	Δt
Description	Five funky qubits.

CheckDeviceSpec[]

```
? CheckDeviceSpec
```

CheckDeviceSpec[spec] checks that the given device specification satisfies a set of validity requirements, returning True if so, otherwise reporting a specific error. This is a useful debugging tool when creating a device specification, though a result of True does not gaurantee the spec is valid.

```
CheckDeviceSpec[myDevSpec]
True

CheckDeviceSpec @ <||>
... CheckDeviceSpec: Specification is missing the required key: DeviceDescription.
False
```

GetCircuitSchedule[]

```
? GetCircuitSchedule
```

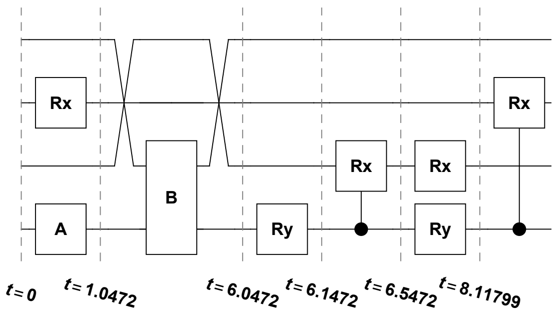
GetCircuitSchedule[circuit, spec] divides circuit into sub–circuits of simultaneously–applied gates (filled from the left), and assigns each a start–time based on the duration of the slowest gate according to the given device specification. The returned structure is {{t1, sub–circuit1}, {t2, sub–circuit2}, ...}, which can be given directly to DrawCircuit[] or ViewCircuitSchedule[].
GetCircuitSchedule[subcircuits, spec] uses the given division (lists of circuits), assumes the gates in each can be performed simultaneously, and performs the same scheduling.
GetCircuitSchedule accepts optional argument ReplaceAliases.
GetCircuitSchedule will take into consideration gates with durations dependent on their scheduled start time.

GetCircuitSchedule can infer what gates can be done simultaneously (by acting on different qubits), and consult their duration in the hardware configuration, to create a schedule of gate columns. The schedule waits for the slowest gate within each sub-circuit to be completed.

```
u =
  Circuit[ Rx2[π/3.] A0[.1] B0,3 Ry0[.1] C0[Rx1[.2]] Rx1[.3] Ry0[π/2] C0[Rx2[.4]]];
GetCircuitSchedule[u, myDevSpec]

ViewCircuitSchedule[%]
DrawCircuit[%%]
```

```
{{0, {Rx2[1.0472], A0[0.1]}}, {1.0472, {B0,3}},
 {6.0472, {Ry0[0.1]}}, {6.1472, {C0[Rx1[0.2]]}},
 {6.5472, {Rx1[0.3], Ry0[π/2]}}, {8.11799, {C0[Rx2[0.4]]}}}
```

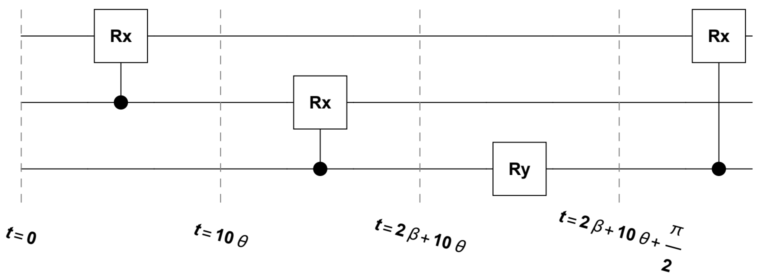
time	gates
0	$Rx_2[1.0472]$ $A_0[0.1]$
1.0472	$B_{0,3}$
6.0472	$Ry_0[0.1]$
6.1472	$C_0[Rx_1[0.2]]$
6.5472	$Rx_1[0.3]$ $Ry_0[\frac{\pi}{2}]$
8.11799	$C_0[Rx_2[0.4]]$



The parameters, and hence resulting durations (depending on the hardware), can be symbolic!

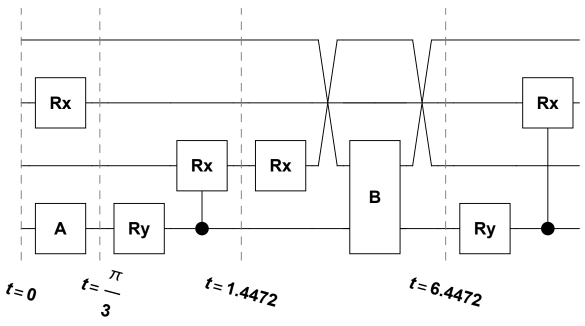
```
GetCircuitSchedule[
      Circuit[ C_1[Rx_2[5 θ]]  C_0[Rx_1[β]] Ry_0[π/2] C_0[Rx_2[.4]]],
      myDevSpec
 ];
ViewCircuitSchedule[%]
DrawCircuit[%%, SubcircuitSpacing → 2]
```

time	gates
0	$C_1[Rx_2[5\,\theta]]$
$10\,\theta$	$C_0[Rx_1[\beta]]$
$2\,\beta + 10\,\theta$	$Ry_0\left[\frac{\pi}{2}\right]$
$\frac{\pi}{2} + 2\,\beta + 10\,\theta$	$C_0[Rx_2[0.4]]$



GetCircuitSchedule[] can also accept pre-divided sub-circuits, and assumes the gates in each can be done simultaneously.

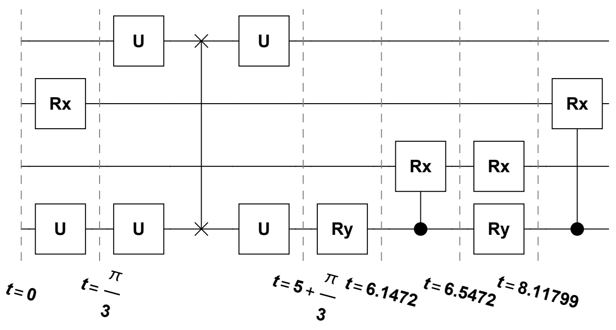
```
GetCircuitSchedule[{
   Circuit[ Rx_2[π/3] A_0[.1] ],
   Circuit[Ry_0[.1] C_0[Rx_1[.2]]],
   Circuit[ Rx_1[.3] B_{0,3}],
   Circuit[ Ry_0[π/2] C_0[Rx_2[.4]]]},
      myDevSpec
 ];
DrawCircuit[%]
```



Notice **GetCircuitSchedule[]** kept the device specification's custom alias gates, **A** and **B**, in their symbolic form. To substitute these gates with their form in the canonical gate basis, use **ReplaceAliases**.

u = Circuit[$Rx_2[\pi/3]\ A_0[.1]\ B_{0,3}\ Ry_0[.1]\ C_0[Rx_1[.2]]\ Rx_1[.3]\ Ry_0[\pi/2]\ C_0[Rx_2[.4]]$];
s = GetCircuitSchedule[u, myDevSpec, ReplaceAliases → True];
ViewCircuitSchedule[s]
DrawCircuit[s]

time	gates
0	$Rx_2\left[\frac{\pi}{3}\right]\ U_0\left[\begin{pmatrix} 0.629819 - 0.0631927\, i & 0.774167 \\ 0.774167 & -0.629819 - 0.0631927\, i \end{pmatrix}\right]$
$\frac{\pi}{3}$	$U_0\left[\begin{pmatrix} \frac{\left(\frac{1}{2}-\frac{\sqrt{3}}{2}\right) e^{-\frac{i\pi}{3}}}{\sqrt{2}} & \frac{\frac{1}{2}+\frac{\sqrt{3}}{2}}{\sqrt{2}} \\ \frac{\frac{1}{2}+\frac{\sqrt{3}}{2}}{\sqrt{2}} & \frac{\left(-\frac{1}{2}+\frac{\sqrt{3}}{2}\right) e^{\frac{i\pi}{3}}}{\sqrt{2}} \end{pmatrix}\right]\ U_3\left[\begin{pmatrix} \frac{\left(\frac{1}{2}-\frac{\sqrt{3}}{2}\right) e^{-\frac{i\pi}{3}}}{\sqrt{2}} & \frac{\frac{1}{2}+\frac{\sqrt{3}}{2}}{\sqrt{2}} \\ \frac{\frac{1}{2}+\frac{\sqrt{3}}{2}}{\sqrt{2}} & \frac{\left(-\frac{1}{2}+\frac{\sqrt{3}}{2}\right) e^{\frac{i\pi}{3}}}{\sqrt{2}} \end{pmatrix}\right]$ $SWAP_{0,3}\ U_0\left[\begin{pmatrix} \frac{\left(\frac{1}{2}+\frac{\sqrt{3}}{2}\right) e^{\frac{i\pi}{3}}}{\sqrt{2}} & \frac{\frac{1}{2}-\frac{\sqrt{3}}{2}}{\sqrt{2}} \\ \frac{\frac{1}{2}-\frac{\sqrt{3}}{2}}{\sqrt{2}} & \frac{\left(-\frac{1}{2}-\frac{\sqrt{3}}{2}\right) e^{-\frac{i\pi}{3}}}{\sqrt{2}} \end{pmatrix}\right]\ U_3\left[\begin{pmatrix} \frac{\left(\frac{1}{2}+\frac{\sqrt{3}}{2}\right) e^{\frac{i\pi}{3}}}{\sqrt{2}} & \frac{\frac{1}{2}-\frac{\sqrt{3}}{2}}{\sqrt{2}} \\ \frac{\frac{1}{2}-\frac{\sqrt{3}}{2}}{\sqrt{2}} & \frac{\left(-\frac{1}{2}-\frac{\sqrt{3}}{2}\right) e^{-\frac{i\pi}{3}}}{\sqrt{2}} \end{pmatrix}\right]$
$5+\frac{\pi}{3}$	$Ry_0[0.1]$
6.1472	$C_0[Rx_1[0.2]]$
6.5472	$Rx_1[0.3]\ Ry_0\left[\frac{\pi}{2}\right]$
8.11799	$C_0[Rx_2[0.4]]$



Naturally the input circuit must be compatible with the given hardware specification.

u = Circuit[$C_1[Ry_0[\phi]]\ C_0[Rx_1[\beta]]\ Rx_1[\phi]\ Ry_0[\pi/2]\ C_0[Rx_2[.4]]$];
GetCircuitSchedule[u, myDevSpec]

```
… GetCircuitSchedule: The circuit(s) contains gates unsupported by the given device specification. See
       ?GetUnsupportedGates.
```

```
$Failed
```

CheckCircuitSchedule

```
? CheckCircuitSchedule
```

CheckCircuitSchedule[{{t1, circ1}, {t2, circ2}, ...}, spec] checks whether the given schedule of sub−circuits is compatible with the device specification, else if it prescribes overlapping sub−circuit execution (regardless of targeted qubits). Times and gate parameters can be symbolic. All gates in a sub−circuit are assumed applicable simultaneously, even if they target overlapping qubits.
CheckCircuitSchedule returns False if the (possibly symbolic) times cannot possibly be monotonic, nor admit a sufficient duration for any sub−circuit.
CheckCircuitSchedule returns True if the schedule is valid for any assignment of the times and gate parameters.
CheckCircuitSchedule returns a list of symbolic conditions which must be simultaneously satisfied for the schedule to be valid, if it cannot determine so absolutely. These conditions include constraints of both motonicity and duration.
CheckCircuitSchedule will take into consideration gates with durations dependent on their scheduled start time.

CheckCircuitSchedule compares the time needed to execute each sub-circuit (per the slowest gate within) to the time between scheduled sub-circuits.

```
CheckCircuitSchedule[{
       {0, Circuit[Rx_1[.1] Ry_0[.1] A_0[.1]]},
       {20, Circuit[C_0[Rx_1[5]]]},
       {40, {Rx_1[0.3], Ry_0[0.1]}},
       {60, {C_0[Rx_2[0.4`]]}}},
 myDevSpec]
```

```
True
```

```
CheckCircuitSchedule[{
       {0, Circuit[Rx_1[.1] Ry_0[.1]]},
       {10, Circuit[C_0[Rx_1[5]]]},
       {12, {Rx_1[0.3], Ry_0[0.1]}},
   (* insufficient time for previous C_0[Rx_1[5]] *)
       {60, {C_0[Rx_2[0.4`]]}}},
 myDevSpec]
```

```
False
```

Naturally, **GetCircuitSchedule** returns a valid schedule.

```
u = Circuit[ Rx_2[π/10] Ry_0[.1] C_0[Rx_1[β]] Rx_1[π/20] Ry_0[π/2] C_0[Rx_2[.4]]];
s = GetCircuitSchedule[u, myDevSpec];
CheckCircuitSchedule[s, myDevSpec]
```

```
True
```

CheckCircuitSchedule can accept both symbolic sub-circuit times *and* parameters. In the event that this does not unambiguously decide the schedule validity, **CheckCircuitSchedule** returns a list of symbolic conditions which must be simultaneously satisfied for the schedule to be valid. This includes consideration that the input schedule is real valued and monotonically increasing. Furthermore, unless reported by an error, it is the *only* solution!

```
CheckCircuitSchedule[{
      {a, Circuit[Rx₁[π/3] Ry₀[.1]]},
      {b, Circuit[C₀[Rx₁[β]]]},
      {c, Circuit[Rx₁[0.3] Ry₀[.1]]},
      {d, Circuit[C₀[Rx₂[0.4]]]}},
 myDevSpec]
```

```
{3 a + π ≤ 3 b, b + 2 β ≤ c, 0.3 + c ≤ d}
```

There are many ways a nominated schedule can be invalid. **CheckCircuitSchedule**[] has comprehensive input validation...

Validation

A sub-circuit might include an unsupported gate..

```
badgate = C₁[Rx₀[θ]];
```

```
CheckCircuitSchedule[{
      {0, Circuit[badgate Ry₀[.1]]},
      {100, Circuit[C₀[Rx₁[β]]]},
      {400, {Rx₁[0.3], Ry₀[0.1]}},
      {1000, {C₀[Rx₂[0.4]]}}},
 myDevSpec]
```

```
CheckCircuitSchedule: The given schedule contains gates incompatible with the device specification. See
      ?GetUnsupportedGates
```

```
$Failed
```

or the schedule contain complex or negative numbers..

```
badtime = 3 + ⅈ;
CheckCircuitSchedule[{
      {0, Circuit[Rx₁[.1] Ry₀[.1]]},
      {badtime, Circuit[C₀[Rx₁[β]]]},
      {400, {Rx₁[0.3], Ry₀[0.1]}},
      {1000, {C₀[Rx₂[0.4]]}}},
 myDevSpec]
```

```
CheckCircuitSchedule: The given schedule times are not motonically increasing, nor can be for any assignment
      of symbols, or they are not real and positive.
```

```
$Failed
```

```
badtime = -1;
CheckCircuitSchedule[{
      {badtime, Circuit[Rx₁[.1] Ry₀[.1]]},
      {100, Circuit[C₀[Rx₁[β]]]},
      {400, {Rx₁[0.3], Ry₀[0.1]}},
      {1000, {C₀[Rx₂[0.4]]}}},
 myDevSpec]
```

```
... CheckCircuitSchedule: The given schedule times are not motonically increasing, nor can be for any assignment
        of symbols, or they are not real and positive.
```

```
$Failed
```

The schedule times might not be monotonically increasing...

```
CheckCircuitSchedule[{
      {0, Circuit[Rx₁[.1] Ry₀[.1]]},
      {100, Circuit[C₀[Rx₁[β]]]},
      {50, {Rx₁[0.3], Ry₀[0.1]}},
      {30, {C₀[Rx₂[0.4]]}}},
 myDevSpec]
```

```
... CheckCircuitSchedule: The given schedule times are not motonically increasing, nor can be for any assignment
        of symbols, or they are not real and positive.
```

```
$Failed
```

```
CheckCircuitSchedule[{
      {t1, Circuit[Rx₁[.1] Ry₀[.1]]},
      {50, Circuit[C₀[Rx₁[β]]]},
      {t2, {Rx₁[0.3`], Ry₀[0.1]}},
      {10, {C₀[Rx₂[0.4`]]}}},
 myDevSpec]
```

```
... CheckCircuitSchedule: The given schedule times are not motonically increasing, nor can be for any assignment
        of symbols, or they are not real and positive.
```

```
$Failed
```

or their may exist no real assignments of the symbolic times that admit them to be monotonically increasing.

```
CheckCircuitSchedule[{
      {2 t1, Circuit[Rx₁[.1] Ry₀[.1]]},
      {50, Circuit[C₀[Rx₁[β]]]},
      {t1/3, {Rx₁[0.3`], Ry₀[0.1`]}},
      {100, {C₀[Rx₂[0.4`]]}}},
 myDevSpec]
```

```
... CheckCircuitSchedule: The given schedule times are not motonically increasing, nor can be for any assignment
        of symbols, or they are not real and positive.
```

```
$Failed
```

GetUnsupportedGates[]

GetUnsupportedGates is a helpful debugging function to determine why a circuit(s) or schedule is incompatible with a given hardware specification.

```
? GetUnsupportedGates
```

GetUnsupportedGates[circuit, spec] returns a list of the gates in circuit which either on non–existent qubits or are not present in or satisfy the gate rules in the device specification. The circuit can contain symbolic parameters, though if it cannot be inferred that the parameter satisfies a gate condition, the gate is assumed unsupported.
GetUnsupportedGates[{circ1, circ2, ...}, spec] returns the unsupported gates in each subcircuit, as separate lists.
GetUnsupportedGates[{{t1, circ1}, {t2, circ2}, ...}, spec] ignores the times in the schedule and returns the unsupported gates in each subcircuit, as separate lists.

```
ViewDeviceSpec[myDevSpec]⟦1, 3⟧
```

Gates			
Gate	Conditions	Noisy form	Duration (Δt)
$H_{q\_}$		H_q $\text{Deph}_q[0.01]$	1
$(r : Rx \mid Ry \mid Rz_{q\_})[\theta\_]$	$0 < \theta < \pi$	$\text{Depol}_q[0.01]$ $r_q[0.01 + \theta]$	θ
$C_{c\_}[Rx_{q\_}[\theta\_]]$	$q > c$	$C_c[Rx_q[\theta]]$ $\text{Deph}_{c,q}[0.01]$	$2\,\theta$
$A_{q\_}[\theta\_]$	$0 < \theta \leq \frac{\pi}{3}$ EvenQ[q]	$A_q[\theta]$ $\text{Damp}_q\left[\frac{\theta}{100}\right]$	$1 + q$
$B_{q1\_,q2\_}$	OddQ[Abs[q1 - q2]]	$A_{q1}\left[\frac{\pi}{10}\right]$ $\text{Depol}_{q1,q2}[0.1]$ $B_{q1,q2}$	5
$\text{Init}_{q\_}$		Init_q	1

u = Circuit[
$X_3\ A_0[\theta]\ C_0[A_2[\pi]]\ Rx_2[8.5\,\pi]\ Ry_0[.1]\ C_1[Ry_0[.1]] \times$
$C_0[Rx_1[.2]]\ Rx_1[.3]\ Ry_0[\pi/2]\ C_0[Rx_2[.4]]\ Ry_0[\alpha]$];
GetUnsupportedGates[u, myDevSpec]

$\{X_3, A_0[\theta], C_0[A_2[\pi]], Rx_2[26.7035], C_1[Ry_0[0.1]], Ry_0[\alpha]\}$

Notice that $\boldsymbol{A_0[\theta]}$ is flagged as unsupported, since it cannot yet be known that $\boldsymbol{\theta < \pi/3}$

GetUnsupportedGates can accept sub-circuits, returning the unsupported gates in each:

```
GetUnsupportedGates[{
   Circuit[ Rx₂[8.5 π] Rz₀[ϕ] ],
   Circuit[Ry₁[.1] C₀[Rx₁[.2]]],
   Circuit[ Rx₀[.3]],
   Circuit[ Ry₀[3 π / 2] C₀[Rx₂[.4]]]},
      myDevSpec
]
```

```
{{Rx₂[26.7035], Rz₀[ϕ]}, {}, {}, {Ry₀[3π/2]}}
```

It can also directly accept a schedule, though will *not* consider whether the scheduled times are compatible with the gate durations (for that, use **CheckCircuitSchedule[]**)

```
GetUnsupportedGates[{
   {1, Circuit[ Rx₂[8.5 π] ]},
   {4, Circuit[Ry₁[.1] C₀[Rx₁[.2]]]},
   {15, Circuit[ Rx₀[.3]]},
   {16, Circuit[ Ry₀[3 π / 2] C₀[Rx₂[.4]]]}},
      myDevSpec
]
```

```
{{Rx₂[26.7035]}, {}, {}, {Ry₀[3π/2]}}
```

InsertCircuitNoise[]

InsertCircuitNoise[] consults the device specification to insert noise into a circuit. The output includes the schedule assumed by **GetCircuitSchedule[]**, based on the gate durations, and active and passive noise sources. The output format...

{ {t1, subcirc1, active1, passive1}, ... }

keeps the stages of gates, active and passive noises separate, so that they can be easily distinguished.

```
? InsertCircuitNoise
```

InsertCircuitNoise[circuit, spec] divides the circuit into scheduled subcircuits, then replaces them with rounds of active and passive noise, according to the given device specification. Scheduling is performed by GetCircuitSchedule[]. The output format is {{t1, active, passive}, ...}, which can be given directly to DrawCircuit[], ViewCircuitSchedule[] or ExtractCircuit[].

InsertCircuitNoise[{circ1, circ2, ...}, spec] uses the given list of sub−circuits (output format of GetCircuitColumns[]), assuming each contain gates which can be simultaneously performed.

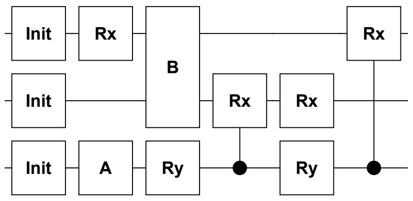
InsertCircuitNoise[{{t1, circ1}, {t2, circ2}, ...} assumes the given schedule (output format of GetCircuitSchedule[]) of {t1,t2,...} for the rounds of gates and noise. These times can be symbolic.

InsertCircuitNoise accepts optional argument ReplaceAliases.

InsertCircuitNoise can handle gates with time−dependent noise operators and durations.

Consider the following simple circuit, which includes aliases **Init**, **A** and **B**.

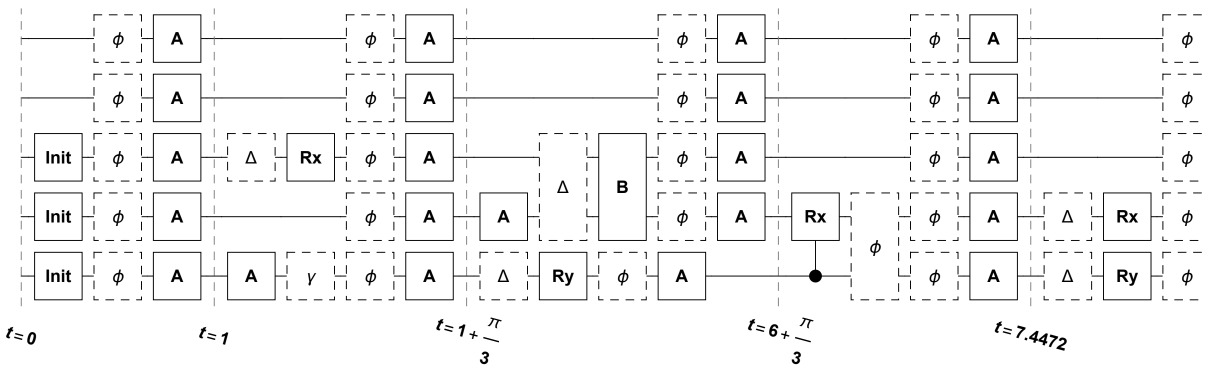
```
u = Circuit[ Init₀ Init₁ Init₂ Rx₂[π/3]
     A₀[.1] B₁,₂ Ry₀[.1] C₀[Rx₁[.2]] Rx₁[.3] Ry₀[π/2] C₀[Rx₂[.4]]];
DrawCircuit[
 u]
```



Attempting to perform this ideal circuit on the represented hardware device will ultimately effect the noisy channel:

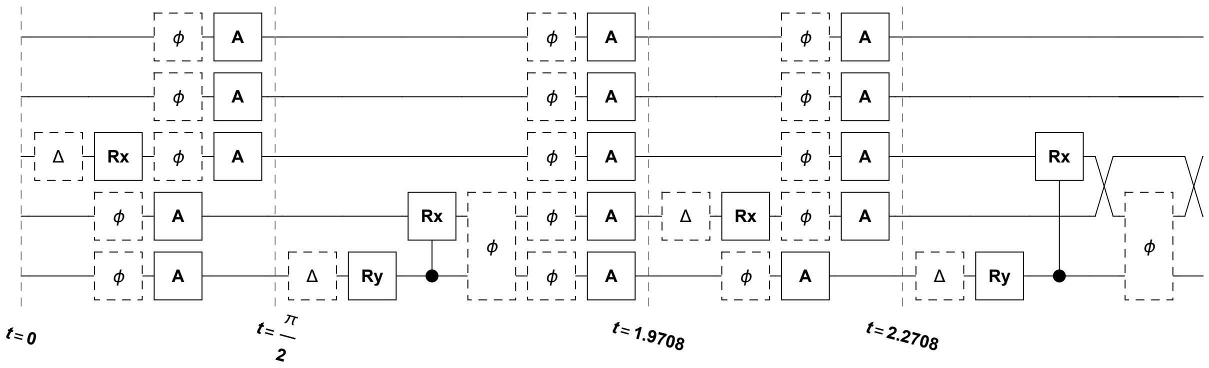
```
InsertCircuitNoise[u, myDevSpec];
ViewCircuitSchedule[%]
DrawCircuit[%%]
```

time	active noise	passive noise
0	Init$_0$ Init$_1$ Init$_2$	Deph$_0$[0] A$_0$[0] Deph$_1$[0] A$_1$[0] Deph$_2$[0] A$_2$[0] Deph$_3$[$\frac{1}{1000}$] A$_3$[1] Deph$_4$[$\frac{1}{1000}$] A$_4$[1]
1	Depol$_2$[0.01] Rx$_2$[1.0572] A$_0$[0.1] Damp$_0$[0.001]	Deph$_0$[$\frac{-1+\frac{\pi}{3}}{1000}$] A$_0$[$-1+\frac{\pi}{3}$] Deph$_1$[$\frac{\pi}{3000}$] A$_1$[$\frac{\pi}{3}$] Deph$_2$[0] A$_2$[0] Deph$_3$[$\frac{\pi}{3000}$] A$_3$[$\frac{\pi}{3}$] Deph$_4$[$\frac{\pi}{3000}$] A$_4$[$\frac{\pi}{3}$]
$1+\frac{\pi}{3}$	A$_1$[$\frac{\pi}{10}$] Depol$_{1,2}$[0.1] B$_{1,2}$ Depol$_0$[0.01] Ry$_0$[0.11]	Deph$_0$[0.0049] A$_0$[4.9] Deph$_1$[0] A$_1$[0] Deph$_2$[0] A$_2$[0] Deph$_3$[$\frac{1}{200}$] A$_3$[5] Deph$_4$[$\frac{1}{200}$] A$_4$[5]
$6+\frac{\pi}{3}$	C$_0$[Rx$_1$[0.2]] Deph$_{0,1}$[0.01]	Deph$_0$[0.] A$_0$[0.] Deph$_1$[0.] A$_1$[0.] Deph$_2$[0.0004] A$_2$[0.4] Deph$_3$[0.0004] A$_3$[0.4] Deph$_4$[0.0004] A$_4$[0.4]
7.4472	Depol$_1$[0.01] Rx$_1$[0.31] Depol$_0$[0.01] Ry$_0$[1.5808]	Deph$_0$[0] A$_0$[0] Deph$_1$[0.0012708] A$_1$[1.2708] Deph$_2$[$\frac{\pi}{2000}$] A$_2$[$\frac{\pi}{2}$] Deph$_3$[$\frac{\pi}{2000}$] A$_3$[$\frac{\pi}{2}$] Deph$_4$[$\frac{\pi}{2000}$] A$_4$[$\frac{\pi}{2}$]
9.01799	C$_0$[Rx$_2$[0.4]] Deph$_{0,2}$[0.01]	Deph$_0$[0.] A$_0$[0.] Deph$_1$[0.0008] A$_1$[0.8] Deph$_2$[0.] A$_2$[0.] Deph$_3$[0.0008] A$_3$[0.8] Deph$_4$[0.0008] A$_4$[0.8]



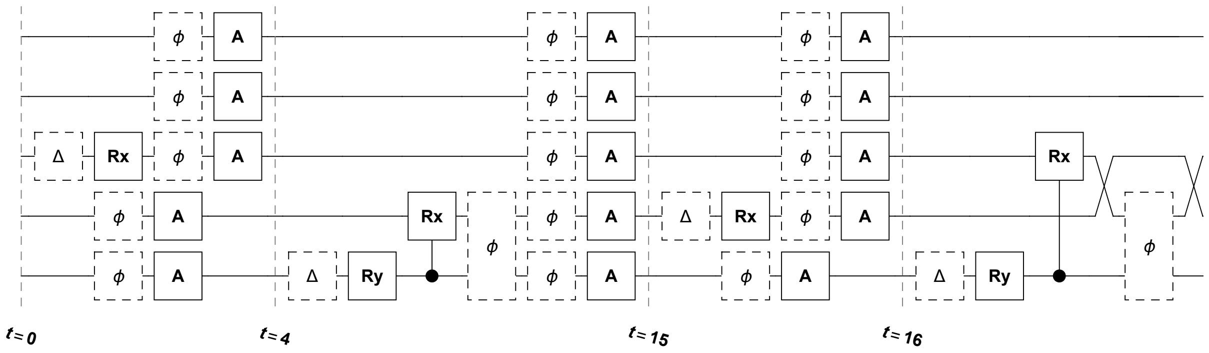
InsertCircuitNoise[] can accept a list of sub-circuits, each of which are assumed to contain gates which can be applied simultaneously.

```
InsertCircuitNoise[{
   Circuit[ Rx₂[π/2] ],
   Circuit[Ry₀[.1] C₀[Rx₁[.2]]],
   Circuit[ Rx₁[.3]],
   Circuit[ Ry₀[π/2] C₀[Rx₂[.4]]]},
      myDevSpec
 ];
DrawCircuit @ %
```



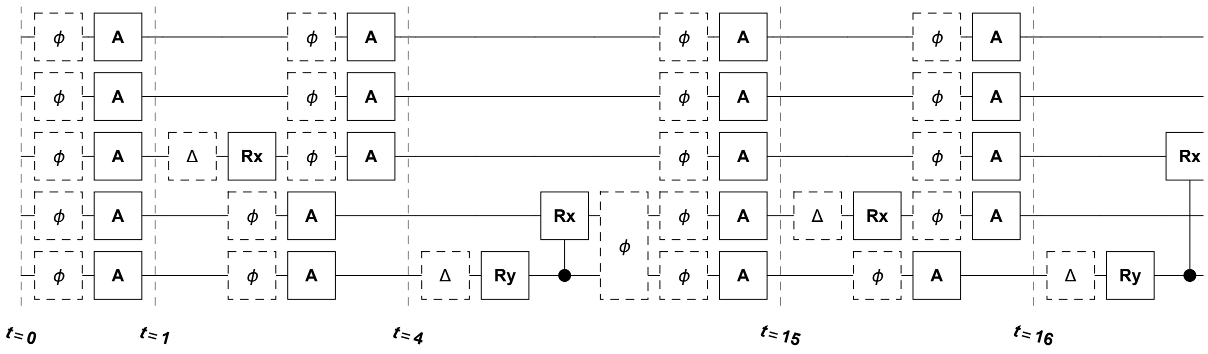
InsertCircuitNoise[] can also accept a completed schedule of gates, e.g. as output by **GetCircuitSchedule[]**. This is a complete specification of the applying of the circuit, and allows delays between sub-circuits which invoke more passive noise.

```
InsertCircuitNoise[{
   {0, Circuit[ Rx₂[π/2] ]},
   {4, Circuit[Ry₀[.1] C₀[Rx₁[.2]]]},
   {15, Circuit[ Rx₁[.3]]},
   {16, Circuit[ Ry₀[π/2] C₀[Rx₂[.4]]]}},
      myDevSpec
 ];
DrawCircuit @ %
```



Notice that an initial sub-circuit time of **t>0** implies an initial stage of passive noise:

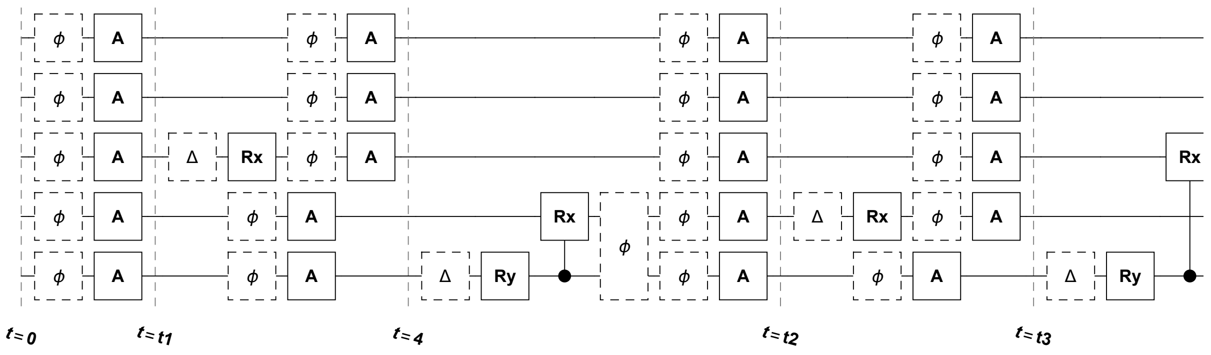
```
InsertCircuitNoise[{
   {1, Circuit[ Rx₂[π/3] ]},
   {4, Circuit[Ry₀[.1] C₀[Rx₁[.2]]]},
   {15, Circuit[ Rx₁[.3]]},
   {16, Circuit[ Ry₀[π/2] C₀[Rx₂[.4]]]}},
      myDevSpec
 ];
DrawCircuit @ %
```



The schedule, and the circuit parameters, can be symbolic! This assumes the sequence of symbolic and numerical times are strictly increasing, which is used to simplify the passive noise parameters.

```
InsertCircuitNoise[{
    {t1, Circuit[ Rx_2[π/3] ]},
    {4, Circuit[Ry_0[.1] C_0[Rx_1[μ / π]]]},
    {t2, Circuit[ Rx_1[1]]},
    {t3, Circuit[ Ry_0[π/2] C_0[Rx_2[.4]]]}},
      myDevSpec
  ];
ViewCircuitSchedule[%]
DrawCircuit[%%]
```

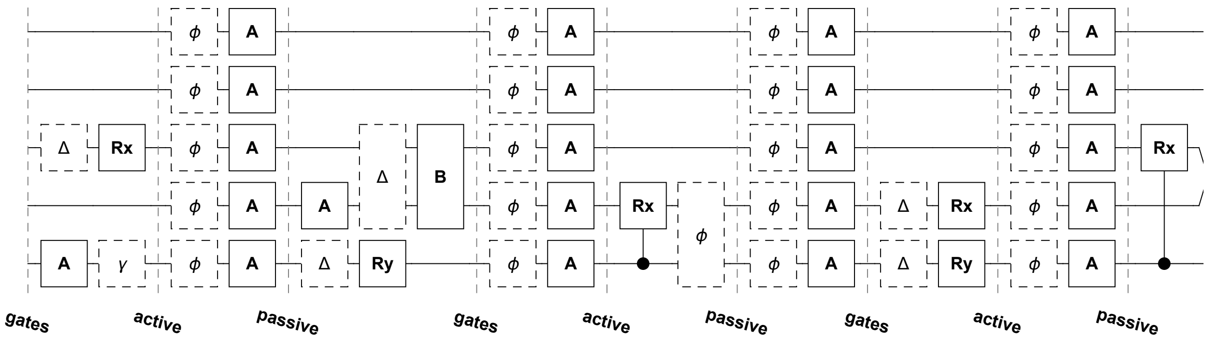
time	active noise	passive noise
0		$\text{Deph}_0[\frac{t1}{1000}]$ $A_0[t1]$ $\text{Deph}_1[\frac{t1}{1000}]$ $A_1[t1]$ $\text{Deph}_2[\frac{t1}{1000}]$ $A_2[t1]$ $\text{Deph}_3[\frac{t1}{1000}]$ $A_3[t1]$ $\text{Deph}_4[\frac{t1}{1000}]$ $A_4[t1]$
t1	$\text{Depol}_2[0.01]$ $Rx_2[1.0572]$	$\text{Deph}_0[\frac{4-t1}{1000}]$ $A_0[4-t1]$ $\text{Deph}_1[\frac{4-t1}{1000}]$ $A_1[4-t1]$ $\text{Deph}_2[\frac{4-\frac{\pi}{3}-t1}{1000}]$ $A_2[4-\frac{\pi}{3}-t1]$ $\text{Deph}_3[\frac{4-t1}{1000}]$ $A_3[4-t1]$ $\text{Deph}_4[\frac{4-t1}{1000}]$ $A_4[4-t1]$
4	$\text{Depol}_0[0.01]$ $Ry_0[0.11]$ $C_0[Rx_1[\frac{\mu}{\pi}]]$ $\text{Deph}_{0,1}[0.01]$	$\text{Deph}_0[\frac{-4+t2-\frac{2\mu}{\pi}}{1000}]$ $A_0[-4+t2-\frac{2\mu}{\pi}]$ $\text{Deph}_1[\frac{-4+t2-\frac{2\mu}{\pi}}{1000}]$ $A_1[-4+t2-\frac{2\mu}{\pi}]$ $\text{Deph}_2[\frac{-4+t2}{1000}]$ $A_2[-4+t2]$ $\text{Deph}_3[\frac{-4+t2}{1000}]$ $A_3[-4+t2]$ $\text{Deph}_4[\frac{-4+t2}{1000}]$ $A_4[-4+t2]$
t2	$\text{Depol}_1[0.01]$ $Rx_1[1.01]$	$\text{Deph}_0[\frac{-t2+t3}{1000}]$ $A_0[-t2+t3]$ $\text{Deph}_1[\frac{-1-t2+t3}{1000}]$ $A_1[-1-t2+t3]$ $\text{Deph}_2[\frac{-t2+t3}{1000}]$ $A_2[-t2+t3]$ $\text{Deph}_3[\frac{-t2+t3}{1000}]$ $A_3[-t2+t3]$ $\text{Deph}_4[\frac{-t2+t3}{1000}]$ $A_4[-t2+t3]$
t3	$\text{Depol}_0[0.01]$ $Ry_0[1.5808]$ $C_0[Rx_2[0.4]]$ $\text{Deph}_{0,2}[0.01]$	$\text{Deph}_0[0.000770796]$ $A_0[0.770796]$ $\text{Deph}_1[\frac{\pi}{2000}]$ $A_1[\frac{\pi}{2}]$ $\text{Deph}_2[0.000770796]$ $A_2[0.770796]$ $\text{Deph}_3[\frac{\pi}{2000}]$ $A_3[\frac{\pi}{2}]$ $\text{Deph}_4[\frac{\pi}{2000}]$ $A_4[\frac{\pi}{2}]$



We can easily manipulate the list structure returned by **InsertCircuitNoise** to keep the gates, active noise and passive noise sections as separate sub-circuits when passed to **DrawCircuit**.

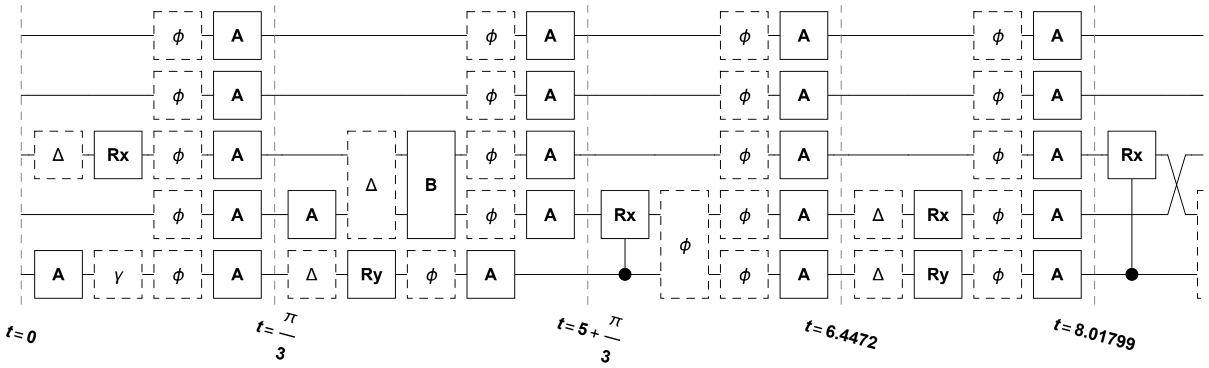
```
u = Circuit[ Rx₂[π/3] A₀[.1] B₁,₂ Ry₀[.1] C₀[Rx₁[.2]] Rx₁[.3] Ry₀[π/2] C₀[Rx₂[.4]] ];
```

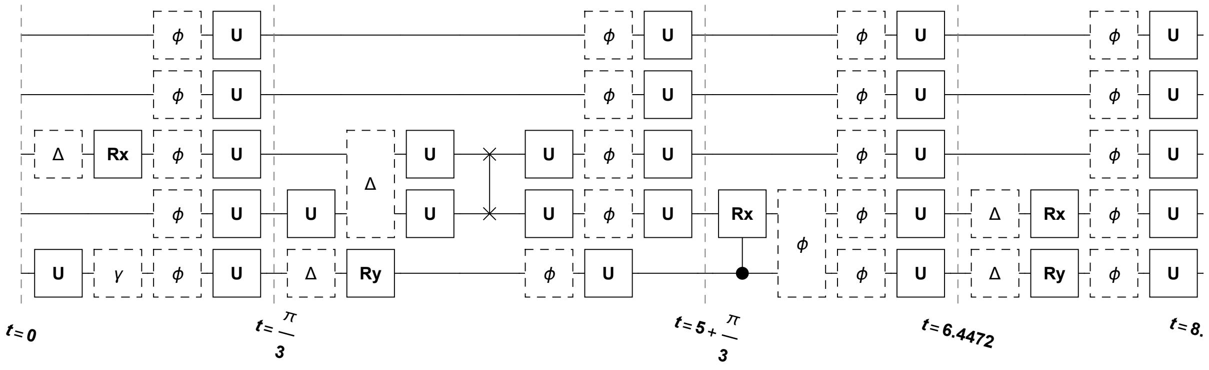
```
InsertCircuitNoise[u, myDevSpec];
DrawCircuit[
    Flatten[%[[All, {2, 3}]], 1],
    SubcircuitLabels → Flatten @ Table[{"gates", "active", "passive"}, 5]
]
```



Notice the alias gates **A** and **B**, defined for this device specification, are present in the output. To substitute them with their definitions in the canonical gate set (which does not change the effective circuit or noise), use **ReplaceAliases**

```
DrawCircuit @ InsertCircuitNoise[u, myDevSpec, ReplaceAliases → False]
DrawCircuit @ InsertCircuitNoise[u, myDevSpec, ReplaceAliases → True]
```





Naturally the input circuit and schedule must be compatible with the given hardware specification.

```
badgate = Rx₀[θ];
InsertCircuitNoise[
    Circuit[badgate Ry₀[.1] C₀[Rx₁[.2]] Rx₁[.3] Ry₀[π/2] C₀[Rx₂[.4]]],
    myDevSpec]
```

```
InsertCircuitNoise: The given subcircuits contain gates not supported by the given device specification. See
    ?GetUnsupportedGates.
```

```
$Failed
```

```
InsertCircuitNoise[{
     {0, Circuit[Rx₂[.1] Ry₀[.1] C₀[Rx₁[.2]]]},
     {.1, Circuit[ Rx₁[.3] Ry₀[π/2] C₀[Rx₂[.4]]]}},
    myDevSpec]
```

```
InsertCircuitNoise: The given schedule is either invalid, or incompatible with the device specification, either
    through unsupported gates, or by prescribing overlapping (in time) sub–circuits.
```

```
$Failed
```

ExtractCircuit[]

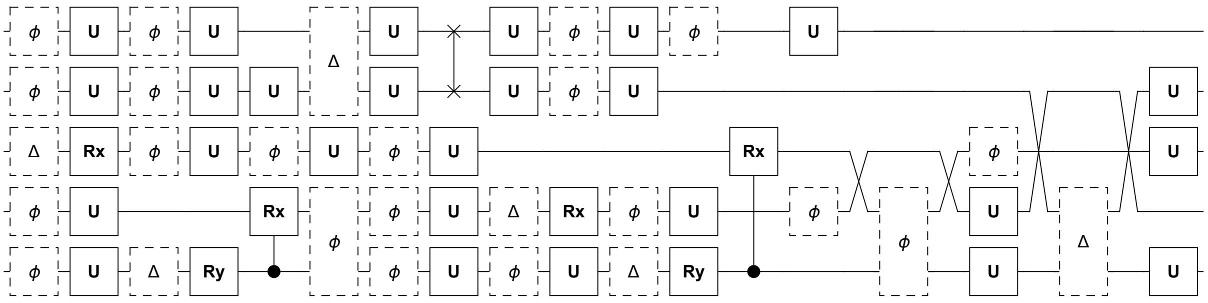
```
? ExtractCircuit
```

ExtractCircuit[] returns the ultimate circuit from the outputs
of InsertCircuitNoise[], GetCircuitSchedule[] and GetCircuitSchedule[].

The structured outputs of **InsertCircuitNoise[]** and **GetCircuitSchedule[]** can be converted to a (flat) noisy circuit using **ExtractCircuit[]**

```
sched = InsertCircuitNoise[{
    {0, Circuit[ Rx₂[ π / 3] ]},
    {2, Circuit[Ry₀[.1] C₀[Rx₁[.2]]]},
    {13, Circuit[ Rx₁[.3] B₃,₄]},
    {40, Circuit[ Ry₀[π / 2] C₀[Rx₂[.4]] B₀,₃]}},
      myDevSpec,
      ReplaceAliases → True
  ];
```

```
circ = ExtractCircuit[sched];
DrawCircuit[circ]
```

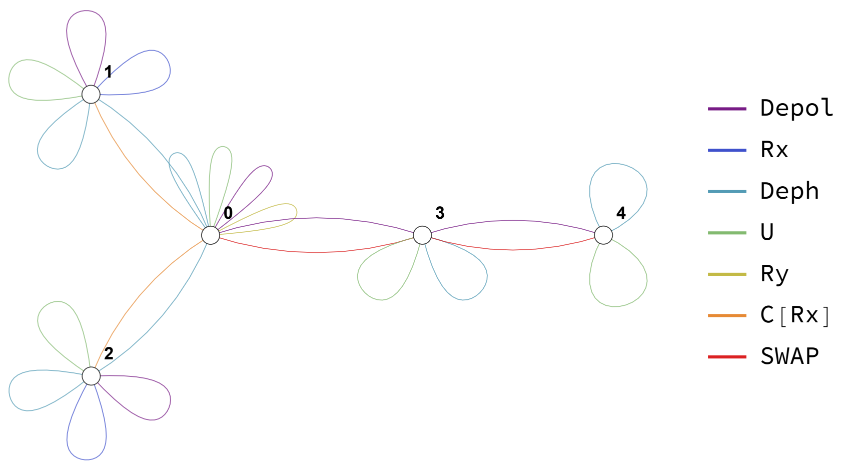


This can then be given directly to functions which accept circuits, like **ApplyCircuit[]** (assuming no parameters or aliases remain) and **DrawCircuitTopology[]**

```
ρ = CreateDensityQureg[ myDevSpec[NumTotalQubits] ];
ApplyCircuit[circ, ρ];
CalcPurity[ρ]
```

```
0.523441
```

```
DrawCircuitTopology[circ]
```



Namespaces

The QuESTlink API is now divided into four namespaces/contexts.

QuEST` contains the main library functions:

```
? QuEST`*
```

▼ QuEST`

ApplyCircuit	CollapseToOutcome	GetUnsupportedGates
ApplyPauliSum	CreateDensityQureg	InitClassicalState
CalcCircuitMatrix	CreateDensityQuregs	InitPlusState
CalcDensityInnerProduct	CreateDownloadedQuESTE-nv	InitPureState
CalcDensityInnerProducts	CreateLocalQuESTEnv	InitStateFromAmps
CalcExpecPauliProd	CreateQureg	InitZeroState
CalcExpecPauliSum	CreateQuregs	InsertCircuitNoise
CalcFidelity	CreateRemoteQuESTEnv	IsDensityMatrix
CalcHilbertSchmidtDistance	DestroyAllQuregs	MixDamping
CalcInnerProduct	DestroyQuESTEnv	MixDephasing
CalcInnerProducts	DestroyQureg	MixDepolarising
CalcPauliSumMatrix	DrawCircuit	MixTwoQubitDephasing
CalcProbOfOutcome	DrawCircuitTopology	MixTwoQubitDepolarising
CalcPurity	ExtractCircuit	Operator
CalcQuregDerivs	GetAllQuregs	PlotDensityMatrix
CalcTotalProb	GetAmp	SetQuregMatrix
CheckCircuitSchedule	GetCircuitColumns	SetWeightedQureg
CheckDeviceSpec	GetCircuitSchedule	SimplifyPaulis
Circuit	GetPauliSumFromCoeffs	ViewCircuitSchedule
CloneQureg	GetQuregMatrix	ViewDeviceSpec

QuEST`Gate` contains the gate symbols recognised by functions like **ApplyCircuit[]**, **CalcCircuit-Matrix[]** and **DrawCircuit[]**

```
? QuEST`Gate`*
```

▼ QuEST`Gate`

Da-m-p	De-p-o-l	H	Kra-u-s	P	R	Ry	S	T	X	Z
De-p-h	G	Id	M	Ph	Rx	Rz	SW-A-P	U	Y	

QuEST`Option` contains the optional arguments to functions in **QuEST`**

```
? QuEST`Option`*
```

▼ QuEST`Option`

Compactify	LabelDrawer	ShowProgress	WithBackup
DistinguishBy	PlotComponent	ShowRepetitions	
DistinguishedStyles	ReplaceAliases	SubcircuitLabels	
DividerStyle	ShowLocalGates	SubcircuitSpacing	

QuEST`DeviceSpec` contains the keys for defining a device specification

```
? QuEST`DeviceSpec`*
```

▼ QuEST`DeviceSpec`

Aliases	Gates	NumTotalQubits	UpdateVariables
DeviceDescription	InitVariables	PassiveNoise	
DurationSymbol	NoisyForm	Qubits	
GateDuration	NumAccessibleQub-its	TimeSymbol	