



## **Inq and Table Layout**

### **Mini Guide**

Copyright 2011 © Inqwell Ltd.

## Table of Contents

Document History.....	2
Confidentiality Notice.....	2
1 Introduction.....	3
2 Defining a Table.....	5
3 Filling Cells.....	6
4 Evenly Sized Buttons.....	7
5 Naming the Table .....	8
6 Creating Hierarchies .....	8
7 Decoration .....	8
8 Table Properties .....	8
8.1 Accessing Columns and Rows.....	9
8.2 HGap and VGap.....	9
9 Outstanding Issues.....	12

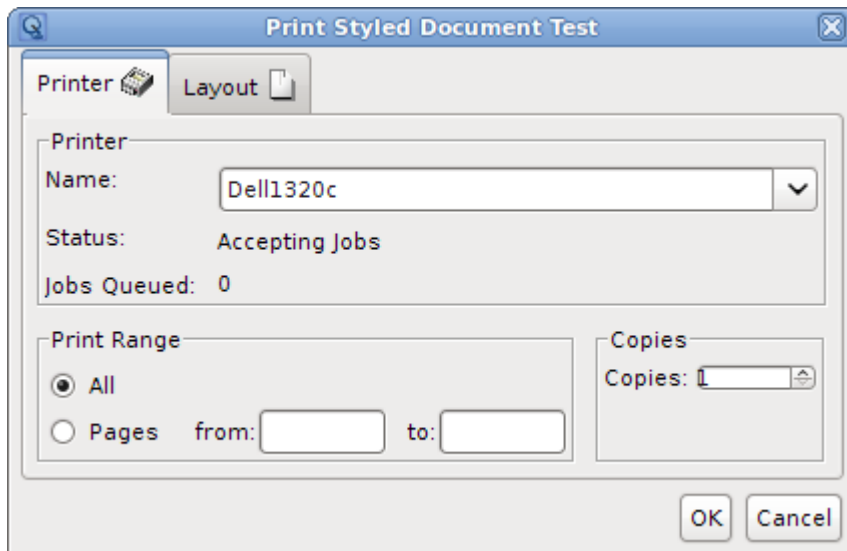
## Document History

Version	Date	Notes
0.1	04/10/10	Initial Version

## 1 Introduction

This document discusses the integration of the TableLayout layout manager into the Inq language. Use of TableLayout greatly increases the flexibility of GUI creation in the Inq client as well as simplifying the layout string required to produce a similar result when using Inq's Row/Column mechanisms in complex GUIs.

This document assumes the reader is familiar with Inq client GUIs in general. Further information on the Inq language is available at the [Inqwell](http://www.inqwell.com) website.



The dialog shown above is the print dialog contained within the standard Inq distribution. It uses the nested row/column layout mechanism we have had for a long time, however even this fairly simple specimen presents quite a few challenges in order to produce the results shown.

```

1   Geometry xy:vf Column
2   {
3     Geometry xy:vf Border Caption "\"Printer\""; Margin d:3 Row
4     {
5       // We really need a table-based layout scheme for aligning
6       // labels with their components, when the components are
7       // not of all the same height.
8       Geometry xy:fv Column
9       {
10        Label cbPrinters
11        ---
12        Label lStatus
13        --
14        Label lJobsQd
15      }
16      --
17      Column
18      {
19        Geometry xy:vf cbPrinters
20        ~
21        <>
22        Geometry xy:vf lStatus
23        ~
24        <>
25        Geometry xy:vf lJobsQd
26      }
27    }
28  Row
29  {

```

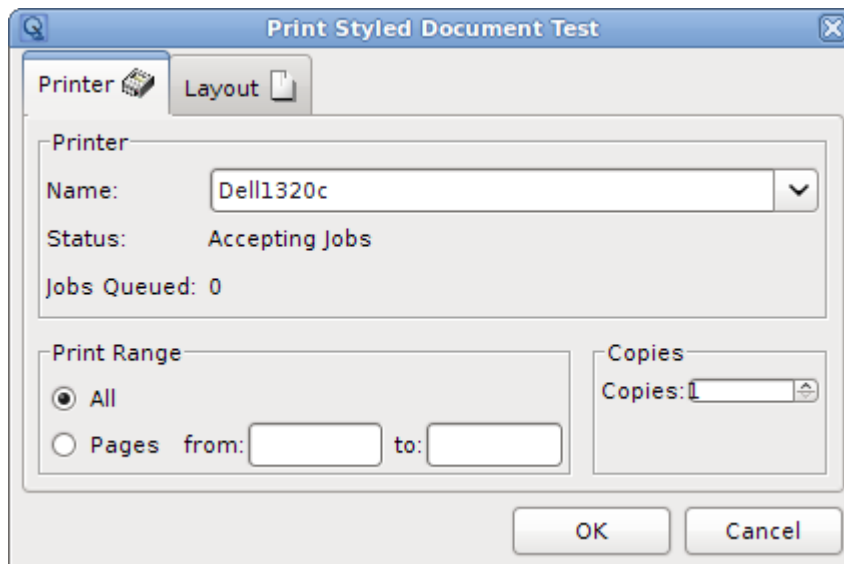
```

30     Geometry xy:vf Border Caption \"Print Range\"; Margin d:3 Row
31     {
32         Column
33         {
34             Geometry xy:vf rAll
35             Geometry xy:vf Row
36             {
37                 rRange
38                 ~
39                 Geometry d:f Row { Label tfStart # tfStart }
40                 ~
41                 Geometry d:f Row { Label tfEnd # tfEnd }
42                 printGroup
43             }
44         }
45     }
46     Geometry xy:vf Border Caption \"Copies\"; Margin d:3 Row
47     {
48         Geometry xy:vf Align t Row
49         {
50             Geometry xy:vf Label sCopies # sCopies
51         }
52     }
53 }
54 }

```

The layout string for the 'Printer' tab is as above. Even so, looking at the dialog carefully you can see that the **0** isn't quite lined up with **Jobs Queued**, nor **Accepting Jobs** with **Status**. For a while now a better layout method has been awaited. To provide this the open source layout manager [TableLayout](#) has been used with appropriate additions to the Inq layout syntax.

Here is the same dialog using TableLayout:



The misalignments are now fixed but more importantly the layout required is conceptually much simpler:

```

1     Geometry xy:vf Border Caption \"Printer\"; Margin d:3
2     Table Columns labels=GRID_PREF, 5, widgets=GRID_FILL
3         Rows 0.33,
4             0.33,
5             0.34;
6     {
7         Column labels
8         {
9             Label cbPrinters

```

```

10     Label lStatus
11     Label lJobsQd
12     }
13     Column widgets
14     {
15         cbPrinters
16         lStatus
17         lJobsQd
18     }
19     }
20     Row
21     {
22     Geometry xy:vf Border Caption \"Print Range\"; Margin d:3
23     Table Columns GRID_PREF, 10, GRID_PREF, GRID_FILL, 5, GRID_PREF, GRID_FILL
24         Rows GRID_PREF,
25             GRID_PREF;
26     {
27         Row
28         {
29             rAll
30         }
31         Row
32         {
33             rRange
34             ~           // advance past 10 pixel space column
35             Label tfStart # // Label of tfStart and avoid Label id id grammar trap
36             tfStart
37             ~           // advance past 5 pixel space column
38             Label tfEnd #
39             tfEnd
40             printGroup // button group - not a component, just for context
41         }
42     }
43     Geometry xy:vf Border Caption \"Copies\"; Margin d:3
44     Table Columns GRID_PREF, GRID_FILL
45         Rows GRID_FILL;
46     {
47         Cell 0 0 Align t Label sCopies
48         Cell 1 0 Align t sCopies
49     }
50     }

```

In both examples the layout is taking place within a `Y_AXIS` box, that is its children are laid out vertically. This box is the 'printer' child of its parent tab pane and the default layout mechanism remains a `Column` in which further `Columns` or `Rows` can be nested.

## 2 Defining a Table

The `Table` keyword opens a `TableLayout`, which has a fixed number of columns and rows to define it. The syntax requires that the columns (later a cell's x coordinate) are defined first followed by the rows (a cell's y coordinate). Cell definitions are separated by commas and are expressions that when evaluated are

- a number such that  $0 < cell < 1$  to specify a relative amount of the overall dimension
- a number such that  $cell \geq 1$  for an absolute pixel size (TS: need a function to convert font string widths to pixels perhaps)
- `GRID_PREF` a predefined constant meaning that the size will be the largest of the preferred sizes of the components in the row or column
- `GRID_MIN` – the largest of the minimum sizes of the components in the row or column
- `GRID_FILL` meaning all the remaining space available

These alternatives and how they are used by TableLayout are summarised in the [tutorial](#), from which:

*TableLayout uses a complex algorithm to determine the preferred layout size. The entire algorithm is beyond the scope of this article, but the fundamental idea behind the algorithm is to add the preferred sizes of all rows and columns to arrive at the container's preferred size. The preferred size of a column is fixed if the column is given an absolute size. For scalable, fill, and preferred columns the preferred width is determined by the column's percentage or fill/prefer attribute and the preferred widths of all components contained either partly or wholly in the column. Since a component can occupy many scalable columns and a single column can contain many such components, the preferred size can be tricky to determine.*

*However, the final behaviour is simple. Any component in an absolute column will be given an absolute width. Any component partly or wholly in a scalable, fill, or preferred column will be given a width equal or greater than its preferred width. The component will be given a greater width only if necessary to ensure that another component is given its preferred width.*

In this example, three tables have been used, one each for the principle component groupings. These have been deployed in the outer Column as a Table and a Row of two Tables. Note that only swing components can be bordered, TableLayout does not support decoration of (groups of) cells, so nesting tables like this will be commonplace. Furthermore, grouping components into sub-tables makes any one table simpler and reusable when laid out separately.

The first Table is 3x3 with labels and their components in two of the columns. These columns are defined as GRID\_PREF and GRID\_FILL. It is quite common to have one or more GRID\_FILL cells as these take up all the available space in their dimension. If the container is resized then GRID\_FILL cells are the ones affected, so we get the desired result of fixed-size labels aligned with varying sized input/output components.

Notice that the middle column is a fixed size of 5 pixels. This is to give a little space between the labels and their associated component. The middle column doesn't actually contain anything. This is different to how spacing is achieved using nested Row/Column, where successive ~ characters each represent 5 pixels of space (and actually results in a 'fixed space' component being created). The rows are given fixed but relative sizes that sum to 1. [Experiment using GRID\_PREF for all rows].

Any cell definition can be given a symbolic name by prefixing it with *name* =. The columns of interest have been given the names *labels* and *widgets* so we can refer to them later when addressing cells. Names must be unique within their axis. Overall, the column and row specifications have been laid out in the text so as to aid the reader's visualisation of the grid. A semi-colon is required to end the cell definition section. The table's content follows between {...}.

### 3 Filling Cells

A cell is addressed by specifying its column and row number. It is often the case that components will be laid out in logical groups occupying a single row or column. An axis and coordinate can be specified with the CoLumn or Row keyword followed by a coordinate reference, which may be a *name*, an integer literal where zero is the first cell or *expression*; that evaluates to an integer (the semicolon is required). Of course, it is convenient to name rows and columns so that the grid can be altered without upsetting the content text. In particular, since the spacing model of a TableLayout uses empty cells these can be added without further maintenance.

In the first table we are interested in addressing particular columns and placing components within them. Reference to components is as for existing layouts – the component name within the collection passed to the layout function. Without further qualification, within a Column or Row the minor coordinate starts at zero and increments automatically by one for each component. Likewise, the successive CoLumn (or Row) constructs also autoincrement if not explicit. Use of CoLumn and Row can be mixed however the implicit counters are reset to zero if the major axis is changed.

In the second table ("Print Range") only implicit coordinates are used. In this case, cells that are to be left empty can be stepped over using the ~ character. The existing syntax for creating on-the-fly labels applies, so when necessary the # character can be used to avoid the Label <id> <id> grammar trap. Use of # does

nothing and does not autoincrement the coordinate. Again as before, specifying a button group in the layout adds the group to the Inq parent but has no other effect.

The last Table created illustrates specific cell addressing and the use of TableLayout's cell alignments. Outside of Row or Column individual cells can be addressed with the `Cell` keyword. Two or four cell references then follow (names, integer literals or expressions as above) to specify the column and row of the starting (or only) and optional ending cells. TableLayout supports a number of what it calls justifications that can be specified independently for the two axes. In Inq terms we reuse the `Align` keyword so we call them alignments. When the space available in a cell is more than that required to accommodate the component's preferred size the default behaviour is to enlarge the component to fit the cell, so called FULL justification. In the Inq layout syntax, within a Table the `Align` keyword is followed by one or two literal characters to specify the alignment in only the y or y and x directions respectively. These are t (top) , c (centre), b (bottom) and l (left), c (centre), r (right) further e (leading), a (trailing) and f (full). There is one special case: `Align c` will set centre alignment for both y and x axes.

Referring to the [tutorial](#), we see that components can occupy a range of cells (not used in this example). A four argument `Cell` construct can be used to specify a cell range. Alternatively, the layout syntax supports `HSpan` and `VSpan` qualifiers whose single argument identifies the end cell by name, literal or expression. `HSpan` and `VSpan` are valid inside a Row or Column.

## 4 Evenly Sized Buttons

In dialogs it is aesthetically nicer if the OK and Cancel buttons are the same size. This is something that the current layout method cannot achieve but is possible using TableLayout like this:

```
layout(., dialog, "tab { bPrinter }
    Geometry xy:vf Margin d:3
    Table Columns GRID_FILL, 0.2, 5, 0.2
    Rows GRID_FILL;
    {
    Cell 1 0 Align b bOk
    Cell 3 0 Align b bCancel
    }
");
```

In this case we say that both columns for the buttons will have 20% of the available space (in fact an arbitrary figure although reducing it will make the dialog wider) and put a FILL column to their left (assuming we want the buttons pushed over to the right). The `Align` clause is only there to keep the buttons at the bottom if their table is resized (in fact the table's `Geometry` prevents this).

## 5 Naming the Table

As for nested Row/Column, a Table can be named by including its identifier after the Table keyword and before the grid specification.

```
Table copies Columns GRID_PREF, GRID_FILL
  Rows GRID_FILL;
{
  ...
```

In this snippet the table will be known as `copies` when added to the Inq parent node. This means the table can be accessed after the layout is complete – in particular property access is supported for any named coordinate so that the technique of setting a row or column's height can be used to toggle its visibility.

Note that, unlike boxes, tables cannot be declared as a data type. The setting up of the grid with optional names is not supported (and would be clumsy) using properties.

## 6 Creating Hierarchies

Within a table the following constructs are permitted as cell content:

- `Table ... { ... }` creates a table as discussed above.
- `Card <identifier> { ... }` creates a swing JPanel with a CardLayout whose immediate children are placed within the Inq Card, supporting their subsequent visibility via the `layoutVisible` property.
- `SplitX [:<weight>] [<identifier>] { ... }` creates a left/right split pane whose content is expected to be two components. Optional literal float *weight* expresses proportion of resize, zero meaning all to the left and one for all to the right. Similarly `SplitY`.
- `Separator` creates a separator. Only valid within a Row or Column when respectively a vertical or horizontal separator is created and added as the current cell.
- `<identifier> { ... }` when identifying a component to be added to a cell, following matched braces will add their content to the component. The component must be eligible to accept children and hence should be a predefined Card, Box or Split. Note that while Table can be nested in the layout grammar, for ease of parsing returning to nested Row or Column cannot be done in this way. Instead, declare the Box as a variable and descend using this technique. Inq recognises the container is a Box and switches to the nested Row/Column grammar.

## 7 Decoration

Within a Table the original component qualifiers of `Caption`, `Border`, `Margin`, `Nofocus` and `Scroll` are supported. `Geometry` is not supported – that function now resides in the table's `Columns` and `Rows` definition.

## 8 Table Properties

If during layout a Table is given a name it will be added to the Inq parent and therefore accessible in general script. Although tables can only be created during layout parsing, should this change in future the data type will be `gGrid`, to be distinct from `gTable` which is a swing `Jtable`.

## 8.1 Accessing Columns and Rows

The columns and rows properties provide access to any of the coordinates that were named during the layout. Vector access is not supported and is in any case brittle.

Suppose, using the example above, we wanted to toggle the visibility of the number of jobs queued. To do this we must name the table itself and the row containing those components:

```
Table printBasics Columns labels=GRID_PREF, 5, widgets=GRID_FILL
  Rows 0.33,
        0.33,
        jobs=0.34;
{
  Column labels
  {
    Label cbPrinters
    Label lStatus
    Label lJobsQd
  }
  Column widgets
  {
    cbPrinters
    lStatus
    lJobsQd
  }
}
```

After layout the following is valid:

```
any $this.props.jobs = printBasics.properties.rows.jobs;
double $this.props.jobsVisible = $this.props.jobs;
```

The above line creates a variable at `$this.props.jobs` which represents the size of row `jobs` in the grid `printBasics`. Initially this variable has the value `0.34`. A row or column's size is always a double (even if it is `GRID_PREF`, `GRID_FILL` or `GRID_MIN`) and since we don't want to know its original value anywhere else other than in the layout string we remember it in `$this.props.jobsVisible`.

Then:

```
$this.props.jobs = 0; // Make invisible
```

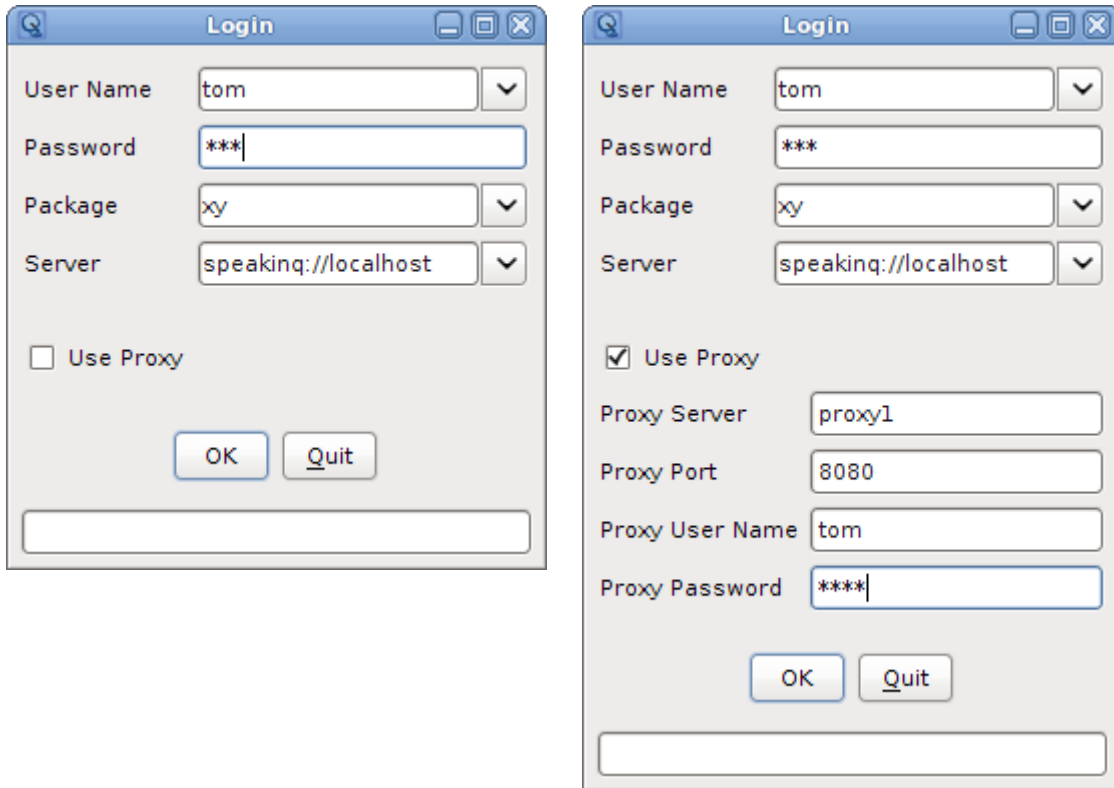
and

```
$this.props.jobs = $this.props.jobsVisible; // Make visible
```

## 8.2 HGap and VGap

The use of empty cells to effect component spacing is useful for specific cases, such as a surrounding empty border or gaps between particular components. `TableLayout` supports the `HGap` and `VGap` properties to provide a fixed space between respectively all columns and rows. In a named table these properties can be accessed in script as `printBasics.properties.hGap` and `printBasics.properties.vGap`. They can also be set during layout by using prefix `Gap <expression>` before `Columns` or `Rows` in the definition section, where `<expression>` evaluates to an integer.

Here is an example combining all the capabilities of Inq TableLayout. It is a Login window with an expandable section for proxy server details:



The layout string to produce this is as follows:

```

1 Table loginDetails Columns 5, labels=GRID_PREF, 5, widgets=GRID_FILL, 5
2 Gap 5 Rows 5,
3     GRID_PREF, // user
4     GRID_PREF, // pwd
5     GRID_PREF, // pkg
6     GRID_PREF, // srvr
7     GRID_PREF, // use proxy
8     proxy=GRID_PREF, // proxy details (nested table)
9     GRID_PREF, // buttons
10    5;
11 {
12     Column labels
13     {
14         ~ // Step over 5 pixel border
15         lusr // Add all labels
16         lpwd
17         lpkg
18         lsrv
19         useProxy // the checkbox
20         Hspan widgets Table proxyDetails Columns labels=GRID_PREF, 5, widgets=GRID_FILL
21             Gap 5 Rows GRID_PREF, // pxy svr
22                 GRID_PREF, // pxy port
23                 GRID_PREF, // pxy user
24                 GRID_PREF; // pxy pwd
25         {
26             Column labels
27             {
28                 lpxysrv
29                 lpxyprt
30                 lpxyusr
31                 lpxypwd
32             }
33             Column widgets
34             {
35                 tpxysrv
36                 tpxyprt
37                 tpxyusr
38                 tpxypwd
39             }
40         }
41     }
42     Column widgets // the input components of the main section
43     {
44         ~ // Step over 5 pixel border
45         tusr // Add the input components
46         tpwd
47         tpkg
48         tsrv
49     }
50 }
51 Table Columns 5, GRID_FILL, 0.2, 5, 0.2, GRID_FILL, 5
52 Rows 5, GRID_PREF, 10 , GRID_PREF ;
53 {
54     Cell 2 1 ok
55     Cell 4 1 quit
56     Cell 0 3 6 3 Margin d:3 tStatus
57 }

```

The relevant points of the configuration and use of TableLayout are summarised as follows:

- At line 2, the Rows are given a spacing of 5 pixels, setting TableLayout's vGap property. This produces the desired spacing of the label/component pairs without the need for empty cells and makes filling the cells using Column addressing more concise as a result.
- The expandable section is itself a Table, making this an example of Table nesting. The reason for doing this is so that the visibility of these components can be controlled by adjusting the height of the nested Table's containing cell in the outer Table, which we have named proxy in anticipation of this, lines 8 and 20. While a nested Table is not strictly necessary, the alternative would require altering

the height of all the rows containing the various components we want to hide or show.

- Further at line 20, the HSpan qualifier causes the proxy Table to occupy the columns between the current column *labels* and the named column *widgets* inclusive. Notice also that the nested table does not need to define any overall border; that is being provided by the outer table.
- The OK and Cancel buttons are given a table of their own at line 51 (remember that there is an implicit Column provided by Inq for the top-level components, two Tables in this case). The reason for doing this rather than accommodating them in the `loginDetails` Table is because to centre them as shown would have required a more complex Column configuration, with the consequent need to use HSpan across all the other components. In general, decisions like this will be made on a case-by-case basis.
- Finally, this Table also contains a status area (in fact a text field) which can conveniently span all the columns including the border space by using a four-argument Cell addressing clause. An alternative way of margining this component using the `Margin d:3` qualifier is shown. Further information about this and other qualifiers together with Inq's original nested Row/Column layout model can be found at the [GUI Basics](#) section of the Inqwell website. Control of visibility as well as other advanced techniques using `TableLayout` are discussed in part 2 of the [TableLayout Tutorial](#).

Although further explanation is beyond the scope of this document, the following fragments of script effect the visibility of the proxy details section:

```
// Proxy visibility row
any login.vars.proxy = login.loginDetails.properties.rows.proxy;
.
.
gEvent(useProxy, call showHideProxy());
.
.
local function showHideProxy()
{
  $this.vars.proxy = $this.saved.useProxy ? GRID_PREF : 0;
  show($this, true); // force resize of window
}
}
```

## 9 Outstanding Issues

Property access to support the dynamic creation of rows or columns – useful for creating guis in a fully dynamic way – is not yet supported.

The ability to see the grid by drawing lines between cells as shown in the `TableLayout` tutorial may be added at a later date.