

THE NEW GRAPHICAL PROGRAMMING LANGUAGE

# FLOW STONE



 DSP Robotics  Ruby Inside

## USER GUIDE

VERSION 3.0.8

# CONTENTS

---

CHAPTER 1      **1 Introduction**      **10**

---

<b>ABOUT THIS GUIDE.....</b>	<b>11</b>
WHAT IS FLOWSTONE?.....	12
HOW IT WORKS.....	12

CHAPTER 2      **2 User Interface**      **14**

---

<b>TOOLBOX.....</b>	<b>16</b>
COMPONENT BROWSER.....	16
FILTER PANE.....	17
FILTER BUTTONS .....	17
SEARCH BOX.....	18
QUICK FIND.....	18
CLEAR ALL FILTERS.....	19
FAVOURITES.....	19
DISPLAY SCALE.....	19
LOCAL TOOLBOX.....	20
<b>TAG BAR.....</b>	<b>22</b>
EDITING TAGS.....	22
TAGGING COMPONENTS.....	23
USING TAGS.....	24
<b>NAVIGATOR.....</b>	<b>25</b>
JUMPING.....	25
PANNING.....	26
BOOKMARKS.....	26
<b>SCHEMATIC WINDOW.....</b>	<b>27</b>
ZOOMING.....	27
SCROLLING.....	28
PANNING.....	28
RESETTING ZOOM AND PAN.....	28

CHAPTER 3      **3 Components & Links**      **29**

---

<b>COMPONENTS.....</b>	<b>34</b>
ADDING A COMPONENT.....	34
SELECTIONS AND THE ACTION PANEL.....	38
MOVING COMPONENTS.....	38
NAMING COMPONENTS.....	39
DELETING COMPONENTS.....	40
RESIZING.....	40
MULTIPLE SELECTIONS.....	41
CONNECTOR LABELS.....	42
CUT, COPY AND PASTE.....	42
<b>LINKS.....</b>	<b>43</b>

CREATING A LINK.....	43
ALLOWED LINKS.....	44
MOVING A LINK.....	44
DELETING A LINK.....	45
LINK ORDER.....	46
BENDING LINKS.....	47
RESOLVING CONTROL POINTS.....	48
STRAIGHT LINKS.....	49
AUTO LINKING.....	50
SMART LINKING.....	51
REMOVING MULTIPLE LINKS.....	52
SWAPPING LINKS.....	52
WIRELESS LINKS.....	53
FOLLOW WIRELESS.....	54

CHAPTER 4

**4 Modules**

**57**

<b>KEY DIFFERENCES.....</b>	<b>59</b>
APPEARANCE.....	59
FRONT PANEL.....	60
PROPERTIES.....	60
TOOLBOX.....	61
<b>BASIC OPERATIONS.....</b>	<b>62</b>
MOVING INTO A MODULE.....	62
INPUTS AND OUTPUTS.....	63
TEMPLATE CONNECTORS.....	64
INPUT AND OUTPUT NAMES.....	64
MAKE MODULE.....	65
PROPERTIES.....	65
WIRELESS MODULES.....	66
<b>FRONT PANEL.....</b>	<b>67</b>
ENABLING THE FRONT PANEL.....	67
EDITING THE FRONT PANEL.....	68
SELECTING.....	69
MOVING.....	70
RESIZING.....	70
JUMPING.....	71
SUB-PANEL EDITING.....	72
HIDING ITEM BOUNDARIES.....	73
DRAW ORDER.....	73
OUT OF VIEW ITEMS.....	73
GROUPED ITEMS.....	75
CLIENT AREA.....	75
HIDING THE FRONT PANEL.....	76
VISIBILITY IN PARENT MODULE PANELS.....	76
<b>PROPERTIES.....</b>	<b>78</b>
ENABLING THE PROPERTIES PANEL.....	78
ADDING PROPERTY ITEMS.....	79
CONTROL TYPES.....	80
EDITING THE PROPERTIES PANEL.....	82

# CONTENTS

RESIZING.....	82
CUSTOMIZING.....	82
<b>SYNCHRONISING.....</b>	<b>84</b>
PASTE SYNCHRONISE.....	84
SYNCHRONISE ALL.....	85
SYNCHRONISE PAINTER.....	85
REMOVING SYNCHRONISATION.....	86
FINDING SYNCHRONISED MODULES.....	86
<b>FOCUS MODE.....</b>	<b>87</b>
FOCUS ON A MODULE.....	87
FOCUS ON THE PREVIOUS MODULE.....	87

## CHAPTER 5

### **5 Data Types & Signal Flow 88**

---

<b>STREAM DATA.....</b>	<b>90</b>
DIGITAL SIGNAL PROCESSING IN A NUTSHELL.....	90
MONO.....	91
POLY.....	91
WHEN TO USE POLY OR MONO.....	91
STREAM CONNECTORS.....	93
POLY AND MONO SECTIONS IN AUDIO APPLICATIONS.....	94
BOOLEAN CONNECTORS.....	95
POLY INT.....	95
SSE.....	95
MONO 4.....	96
PERFORMANCE.....	96
<b>TRIGGERED DATA.....</b>	<b>97</b>
HOW IT WORKS.....	97
WHAT IT'S USED FOR.....	98
TRIGGERED DATA TYPES.....	98
<b>EVENT DATA.....</b>	<b>101</b>
HOW IT WORKS.....	101
EVENT DATA TYPES.....	102
<b>CONVERTING BETWEEN DATA TYPES.....</b>	<b>103</b>
STRING <> BOOLEAN.....	103
INT <> BOOLEAN.....	104
STRING <> FLOAT <> INT.....	104
INT/FLOAT > POLY/MONO.....	105
MIDI <> STRING.....	106
STRING SHORTCUTS.....	107

## CHAPTER 6

### **6 Exporting 110**

---

<b>CREATING STANDALONE APPLICATIONS.....</b>	<b>111</b>
LIBRARY DEPENDENCIES.....	113
<b>CREATING PLUGINS.....</b>	<b>114</b>

INPUTS AND OUTPUTS.....	114
CREATE VST/VSTi DIALOG.....	115
STORING VST EXPORT PREFERENCES.....	117
PRESETS.....	117
TIMING INFO.....	118

CHAPTER 7

**7 Advanced GUI Editing 119**

<b>MODULE GUI.....</b>	<b>120</b>
MODULE GUI COMPONENT.....	120
MGUI CONNECTORS.....	121
GUI CONNECTOR TYPES.....	122
COORDINATE SYSTEM.....	123
<b>DRAWING.....</b>	<b>124</b>
DRAWING ON A PANEL.....	124
DRAWING ORDER.....	125
CHAINING GUI COMPONENTS.....	126
<b>MOUSE HANDLING.....</b>	<b>127</b>
MOUSE AREA.....	127
MOUSE CLICKS.....	127
MOUSE DRAGGING.....	128
MOUSE MOVES.....	129
DRAG ACCUMULATE.....	129
<b>REDRAWING.....</b>	<b>131</b>
REDRAW CONTROL .....	131
PRECISION REDRAWS.....	131

CHAPTER 8

**8 Ruby Component 133**

<b>INTRODUCTION.....</b>	<b>134</b>
OVERVIEW.....	134
<b>INPUTS AND OUTPUTS.....</b>	<b>135</b>
ADDING OR REMOVING.....	135
CHANGING TYPE.....	135
INSERTING, DELETING AND MOVING.....	136
NAMING .....	136
<b>CODE EDITOR BASICS.....</b>	<b>137</b>
THE OUTPUT PANE.....	137
THE RUBYEDIT CLASS.....	139
INPUT DATA.....	139
OUTPUT DATA.....	141
<b>THE EVENT METHOD.....</b>	<b>143</b>
METHOD DEFINITION.....	143
CONNECTOR REFERENCING.....	144
EFFECT ON CODE EXECUTION.....	145

# CONTENTS

<b>SCHEDULING EVENTS.....</b>	<b>146</b>
SCHEDULING AN EVENT.....	146
SENDING TO AN INPUT.....	147
SCHEDULING METHODS.....	147
CLEARING EVENTS.....	148
CLOCK ACCURACY.....	148
<b>RUBY VALUES.....</b>	<b>149</b>
THE RUBY VALUE TYPE.....	149
PASSING RUBY VALUES.....	149
<b>PERSISTENCE.....</b>	<b>152</b>
USER STATE MANAGEMENT.....	152
<b>DEBUGGING.....</b>	<b>154</b>
ERROR REPORTING.....	154
THE WATCH METHOD.....	155
<b>DRAWING.....</b>	<b>156</b>
THE DRAW METHOD.....	156
DRAWING CLASSES AND METHODS.....	157
PENS, BRUSHES & COLORS.....	157
BASIC SHAPES.....	158
LINES AND CURVES.....	159
GRAPHICS PATHS.....	161
TEXT.....	163
BITMAPS.....	166
REDRAW.....	168
CLIPPING.....	168
SMOOTHING.....	169
VIEW PROPERTIES.....	170
CHANGING THE VIEW.....	172
<b>ADVANCED BRUSHES.....</b>	<b>173</b>
LINEAR GRADIENTS.....	173
PATH GRADIENTS.....	177
HATCH BRUSHES.....	184
TEXTURE BRUSHES.....	186
<b>ADVANCED PENS.....</b>	<b>188</b>
PEN ALIGNMENT.....	188
LINE JOINS.....	189
DASHES.....	191
BRUSHED LINES.....	192
LINE CAPS.....	193
<b>INTERACTION.....</b>	<b>198</b>
OVERVIEW.....	198
HANDLING MOUSE EVENTS.....	199
MOUSE CLICKS.....	200
MOUSE CAPTURE & DRAGGING.....	201
KEY MODIFIERS.....	202
MOUSE MOVE.....	203
MOUSE CURSOR.....	204
<b>CONTROLS AND DIALOGS.....</b>	<b>206</b>

IN PLACE EDIT CONTROLS.....	206
DROP LISTS.....	207
MESSAGE BOXES.....	208
<b>SOUNDS.....</b>	<b>210</b>
PLAYING.....	210
WAITING FOR COMPLETION.....	210
LOOPING.....	211
STOPPING.....	211
<b>UTILITY METHODS.....</b>	<b>212</b>
<b>CODE SNIPPETS.....</b>	<b>213</b>
ADDING CODE SNIPPETS.....	213
DEFAULT CODE.....	213
SAVING SNIPPETS.....	214
ORGANISING SNIPPETS.....	214
<b>EXTERNAL DLLs.....</b>	<b>215</b>
THE WIN32API EXTENSION.....	215
CREATING A FUNCTION OBJECT .....	215
MAKING THE CALL.....	217
<b>MIDI.....</b>	<b>219</b>
READING MIDI OBJECTS .....	219
CREATING MIDI OBJECTS.....	221
<b>FRAMES.....</b>	<b>223</b>
MONO TO FRAME.....	223
THE FRAME CLASS.....	224
FRAME TO MONO.....	225
FRAME SYNC.....	226
PROCESSING FRAMES IN A DLL.....	227
<b>RUBY TIMEOUT.....</b>	<b>229</b>
GLOBAL TIMEOUT.....	229
LOCAL TIMEOUT.....	229
<b>RUBY LIMITATIONS.....</b>	<b>230</b>
SINGLE INTERPRETER.....	230
STANDARD RUBY LIBRARIES.....	230
DECLARATION ORDER.....	231
DECLARATION PERSISTENCE.....	231
GEMS.....	231
<b>RUBY DLL.....</b>	<b>233</b>
CHANGES.....	233
BUILDING THE DLL.....	234

**9 DSP Code Component 235**

<b>DSP CODING.....</b>	<b>236</b>
THE DSP CODE COMPONENT.....	236
INPUTS AND OUTPUTS.....	236

# CONTENTS

SYNTAX COLOURING.....	237
EDITOR.....	238
LOCAL VARIABLES.....	238
ASSIGNMENTS.....	239
EXPRESSIONS.....	240
CONDITIONAL STATEMENTS.....	241
COMMENTS.....	241
<b>ADVANCED FEATURES.....</b>	<b>242</b>
ARRAYS.....	242
MEM INPUT.....	243
HOP.....	243
LOOP.....	243
STAGES.....	245
DEBUGGING.....	247
<b>ASSEMBLER.....</b>	<b>249</b>
SYNTAX.....	249
OPCODES.....	249

## CHAPTER 10

## **10 DLL Component 251**

---

<b>INTRODUCTION.....</b>	<b>252</b>
<b>THE COMPONENT.....</b>	<b>253</b>
DEFINING INPUTS AND OUTPUTS.....	253
CONNECTOR TYPES.....	254
<b>THE DLL.....</b>	<b>256</b>
DATA TYPES.....	256
INTS.....	257
FLOATS.....	258
BOOLEANS.....	258
STRINGS.....	258
FLOAT ARRAYS.....	259
INT ARRAYS.....	260
BITMAPS.....	262
FRAMES.....	263
HELPERS.....	264
EXAMPLE 1 – FLOAT ADD.....	265
EXAMPLE 2 – STRING UPPERCASE.....	266
EXAMPLE 3 – AUDIO DELAY.....	267
<b>CONNECTING.....</b>	<b>268</b>
A NOTE ABOUT OUTPUTS.....	269
<b>DEBUGGING.....</b>	<b>271</b>
DEBUGGING USING VISUAL STUDIO.....	271
CODE – TEST – DEBUG CYCLE.....	272
<b>SHARING.....</b>	<b>274</b>
SHARE THE DLL SEPARATELY.....	274
EMBED THE DLL.....	274
PROS AND CONS OF EMBEDDING.....	275



EXPORTING.....275

CHAPTER 11

**11 Options 276**

---

**THE OPTIONS DIALOG.....277**  
 APPLICATION.....278  
 NAVIGATOR.....280  
 TOOLBOX.....281  
 SCHEMATIC.....282  
 MODULES.....283  
 EXPORT.....284  
 ADVANCED.....285

1

# Introduction

ABOUT THIS GUIDE AND SOFTWARE OVERVIEW

# About This Guide

This manual provides a detailed description of the FlowStone software and its functions. Its purpose is to show you how the software works and what its capabilities are.

If you are looking for tutorials then see the Tutorials section of the DSP Robotics web site:

<http://www.dsrobotics.com/tutorials.html>

For information about individual components that ship with the software, see the Component Reference guide. These documents can be found on the Manuals section of our web site at:

<http://www.dsrobotics.com/manualsarea.php>

Additional information and articles about the software can be found at:

<http://www.dsrobotics.com/support.html>

If you have any comments about this guide please email them to [info@dsrobotics.com](mailto:info@dsrobotics.com).

## What is FlowStone?

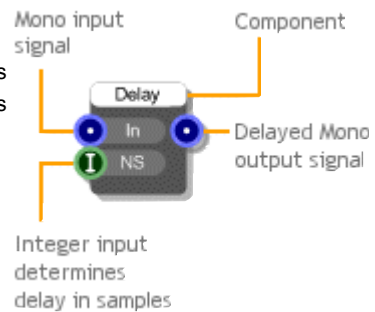
FlowStone is a graphical computer programming language for the rapid development of software applications. Using the software you have complete flexibility to create exactly the kind of application you want. You can add customised controls to modify parameters in real time and group these controls together to make powerful user interfaces.

Your completed creations can then be exported as completely independent executable applications or audio plugins. These applications can then be used on any PC running Microsoft Windows or used at the centre of your own embedded system.

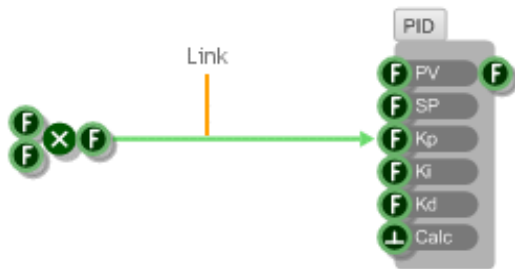
## How It Works

Conceptually, FlowStone is very simple. The software provides a set of building blocks called components. Each component performs a different function. A component can take data in, process it, and pass it out. A component can therefore have inputs or outputs or both.

The inputs and outputs are called Connectors. There are different types of connector for different types of data. Each connector has its own symbol so that you can easily identify the data type.



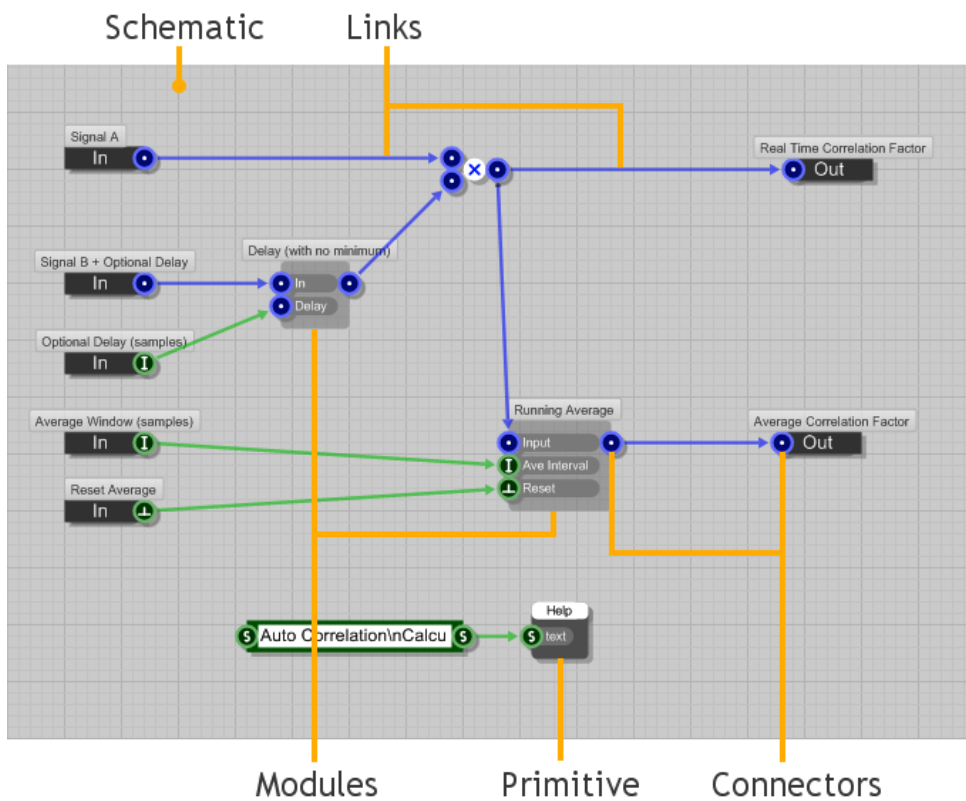
Data is passed between components by connecting them together with links. A link starts at a component output connector and ends at an input connector. In general, data passes through a link from start to end (left to right). However, in some cases, data is also passed from end to start (more on this later).



Components and links are laid out on a 1024x1024 square grid which we call a schematic.

In order to allow for more sophisticated schematics we have a special type of component called a module. Modules are special types of component in that they are defined by their own schematic, containing other components and modules. Modules can also have an interactive front panel with its own controls and custom graphics.

Any component that is not a module we call a Primitive.



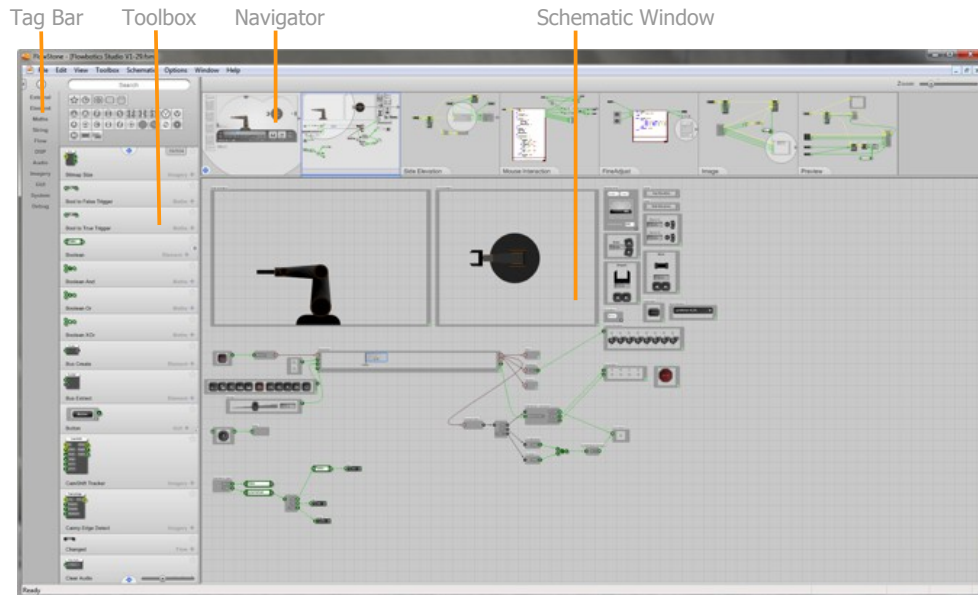
By combining different primitives and modules you can create a vast range of different behaviours and ultimately modules that are individual applications in their own right. These can then be exported as a standalone executable application for use outside of the software.

# 2 User Interface

A FIRST LOOK AROUND

Before you can begin using the software you need to know a little bit about the user interface. Let's start by taking a look at the main application window and how it's laid out.

Across the top of the window is the menu bar. All the applications functions can be accessed from here and it's the first place you should look if you're new to the software and you want to get an idea of what you can do with it. As your mouse passes over menu items help text is displayed on the status bar at the bottom of the application window.



You won't stay with the menu bar for very long though. In FlowStone there are usually several ways to execute the same action so in time you'll find yourself using direct interaction, context menus or shortcut keys instead.

Running down the left-hand side of the application is the **Toolbox**, which provides all the components for building your schematics. To the left of this is the **Tag Bar**. This allows you to quickly access components based on a set of configurable tags.

Across the top of the window is the **Navigator**. As it's name suggests, this is used for navigating through your schematic. You'll find this invaluable once your schematics start to become more complex.

The majority of the application workspace is taken up by the **Schematic Window**. This is where your schematics are created.

# Toolbox

The toolbox provides the building blocks for a schematic – the components. There are already over 450 components to choose from and they are growing in number all the time.

Of course if there are hundreds of components then you'll want to be able to put your hands on the component you're after quickly and with little effort. Thankfully there are several mechanisms in place to make this exceptionally easy.

## Component Browser

### Exploring

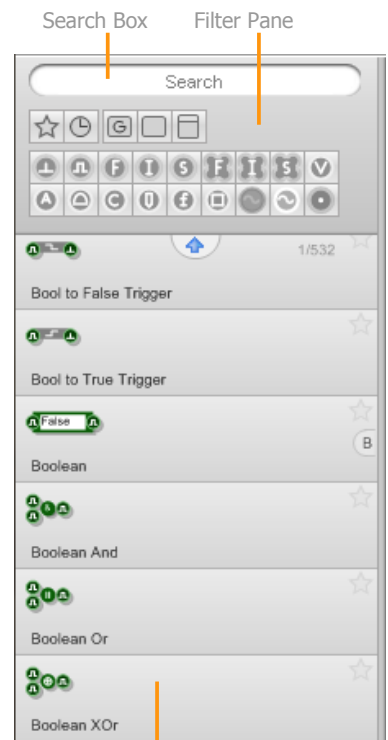
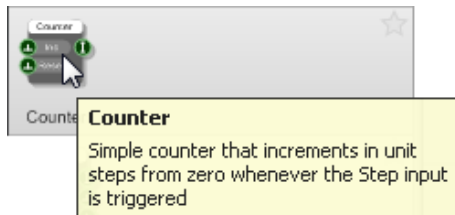
The main part of the toolbox is the component browser. This shows every component and its name. You can scroll through the components by clicking on the page up and down buttons.



You can also scroll by using the mouse wheel, by pressing the PGUP and PGDN keys or using the cursor Up and Down keys to move between items.

### Instant Help

You can get help for a component by hovering your mouse over it. A pop-up window will appear giving a short outline of the purpose of the component.



Component Browser



## Filter Pane

The filter pane appears at the top of the toolbox. It allows you to find what you're looking for much more quickly than just browsing by giving you the ability to filter the displayed components based on certain criteria.

There are two parts to the filter pane, the Search Box and the Filter Buttons. We'll talk about these in the next two sections.

## Filter Buttons

The filter buttons allow you to locate the component you're looking for with just a few clicks.

### Favourites and Recent

The first two buttons on the top row change the list of components so that only Favourites or Recently used components are shown.



The Recently used filter shows the components that you used last. They are ordered such that the most recently used is at the top of the list.

The Favourites filter shows only components that you have marked as a favourite (more on this later),

### Category Filters

The next 3 buttons on the top row filter the list by category. Every component falls into one of 3 categories:



- a Module with a Front Panel or GUI (more on Front Panels in the Components chapter)



- a Module without a Front Panel or GUI



- a Primitive

By clicking one of these you'll only see components which fall in that category. You can then browse the list or apply more filters if necessary to home in on the component you're looking for. You can hold SHIFT and click to select two of these at a time. This allows you to show all modules with or without a GUI for example.

### **Type Filters**

The second group of filter buttons are the filter types. You'll see a button for each connector type here. Click on one of these and the component list will change to show only components which have a connector of the selected type.

You can apply more than one type filter at the same time. To do this, hold CTRL or SHIFT while clicking. This will show components that use at least one of each of the selected types. So if you select Int and Float types for example, only components that have Int AND Float connectors will be shown.

Not all the types are shown by default. You can change this by selecting Edit Filter Types from the Toolbox menu or by clicking on the '+' button that appears when you hover the mouse pointer over the filter type buttons.

Click on the type buttons to add or remove them then click away or select the Edit Filter Types menu item to finish editing.

## **Search Box**

The Search Box appears at the top of the toolbox and provides a quick way to go straight to the component you want. Just type some text in the box and as you type the components will reduce to show only those that match.

The search looks at the component name and the help text. Search results are prioritised by component name match first and then by help text.

You can quickly jump to the search box at any time by pressing CTRL+F

## **Quick Find**

Another way to quickly locate a component is to use the Quick Find feature. This relies on you knowing the name of the component, or at least the start of the name.

Simply move the mouse cursor so it's over the toolbox component browser then just start typing the name of the component you want. The currently selected component will change as you type in order to match the name with what you've typed. After a couple of seconds the search text you typed gets cleared automatically and so if you make a mistake just pause for a second and start typing again.

This can be a very quick way to locate components and can be used in conjunction with any filters that you may have applied.

## Clear All Filters

To clear ALL the filters (including the search) you can either right-click on the filter pane or press the ESC key on your keyboard.

## Favourites

You can create a list of Favourite components that you use most often. By using the Favourites filter in the Filter Pane you have a very quick way to get at the components you need with the minimum of effort.

To mark a component as a favourite, simply click on the 'star' icon in the top-left corner of the component as it appears in the toolbox. The star is now highlighted.

To remove a component from the favourites list all you need to do is click the 'star' icon again.



Click to Add as Favourite



Component is in Favourites  
Click to remove

## Display Scale

All items in the toolbox are displayed to scale. This makes it much easier to identify components by their appearance.

You can set the scale by using the slider at the bottom of the toolbox.



You can also change the scale in regular steps by using the cursor Left and Right keys on the keyboard.

Changing the scale to its lowest level will show only the component names in the toolbox. This is useful if you prefer browsing by text only.

## Local Toolbox

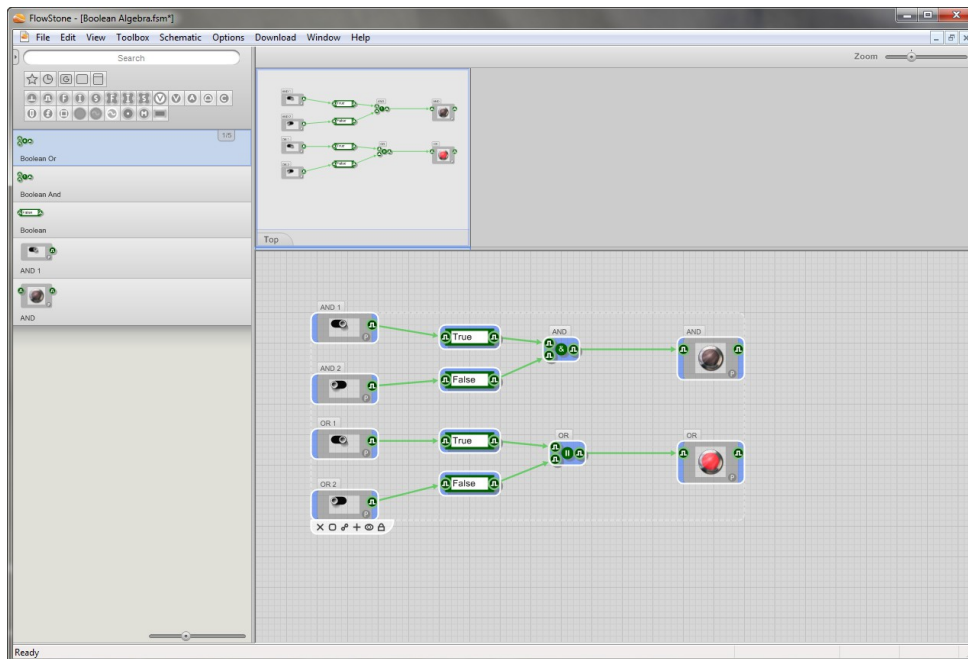
Sometimes you may prefer to work from a more limited set of components. This is often the case for educators who don't want their students to be overwhelmed by the vast array of components on offer. You can do this by creating what we call a Local Toolbox.

A Local Toolbox is a reduced toolbox that is specific to a schematic. It will save with the schematic and it will only appear when that schematic is the one you're using. If you switch schematics the toolbox will switch too.

### Creating a Local Toolbox

There are two ways to do this. The first way is to select the components in the schematic that you want to have in your toolbox then choose Create Local Toolbox from the Schematic menu or press SHIFT + CTRL + T.

The toolbox will change so it only contains the modules and primitives that you selected. The Tag Bar is automatically hidden as components in the Local Toolbox don't have tags.



The second way to create a local toolbox is to select a module and then choose Create Local Toolbox from the Schematic menu or press SHIFT + CTRL + T. On this occasion the Local Toolbox will contain all the components that are inside the module.

Note that the local toolbox has no dependency on the components from which it was created. This means that you can create a local toolbox then delete the components that it was made from.

So for example, a teacher could create a schematic that they want students to build, make a local toolbox from that schematic and then delete everything on the schematic. The student can then take that schematic and be asked to recreate the original from the building blocks in the local toolbox.

### **Removing a Local Toolbox**

To remove a local toolbox, choose Remove LocalToolbox from the Schematic menu or press SHIFT + CTRL + T once again.

### **Restoring a Local Toolbox**

You can restore the last local toolbox by selecting Restore Local Toolbox from the Schematic menu or by pressing SHIFT + CTRL + T once again. This allows you to easily switch between local and full sized toolboxes.

### **Restoring a Local Toolbox**

You can restore the last local toolbox by selecting Restore Local Toolbox from the Schematic menu or by pressing SHIFT + CTRL + T once again. This allows you to easily switch between local and full sized toolboxes.

### **Saving and Loading a Local Toolbox**

A local toolbox is automatically saved with its schematic. So when you open a schematic again, any local toolbox is restored.

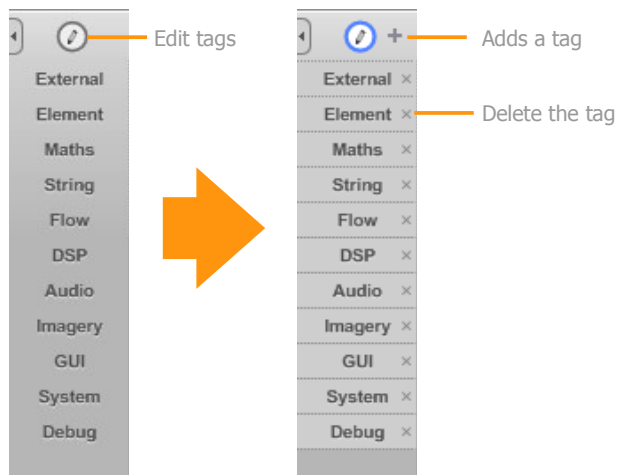
As we said earlier, you can delete everything in the schematic and still save it with the local toolbox.

# Tag Bar

The toolbox incorporates a tag system which allows you to organise components based on some shared characteristic. Tags are added and displayed on the Tag Bar. You can then apply tags to individual components and use the Tag Bar to instantly filter the toolbox based on a particular tag or set of tags.

## Editing Tags

There are some tags already provided but you can completely change these if you want to. To make any kind of changes you first need to unlock the tag bar for editing. To do this click the Edit button, this will put the Tag Bar into edit mode.



### Adding Tags

To add a tag, click the + button and a new one will be added at the end of the list. Type the name and press return on your keyboard to finish.

### Moving Tags

You can move tags around when you're in edit mode. Just click, hold and drag them up and down to change the order.

## Editing and Deleting Tags

To edit a tag, simply double-click on it. Type in your changes then click away or press return to finish. Tags can be deleted by clicking on the x button on the right-hand side of the tag.

## Creating a Separator

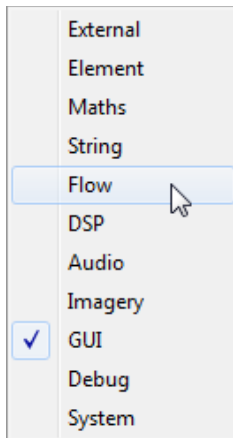
If you have a lot of tags you may want to separate them into banks. To do this you can create a separator to break up each set. Adding a separator is easy. Just add a tag as normal but for the text use three subtract sign characters ie '---'. The separator will span the whole width of the bar when you exit edit mode.

## Tagging Components

Once you have some tags you can then add them to components in the toolbox. To change the tags for a component click the tag button for that component. This is located at the bottom-right corner of each component in the toolbox.



A pop-up menu will appear with all the tags on the tag bar. Click on an unchecked tag to add it or a checked tag to remove it.



Components can have more than one tag. This means that you don't have to pigeon hole components based on one particular characteristic. For example, the Float Array Abs component could have an Array tag and a Maths tag. The tags for each component are shown to the left of the tag button.

## Using Tags

Now that you have your tag bar set up and your tags applied to components you can use the tag bar to filter the toolbox and help you locate components more quickly.

### Filtering by a Tag

To filter by a single tag, just click on it. The tag will show as selected and the toolbox will filter accordingly. Tags are applied in combination with any filters set on the Filter Pane.

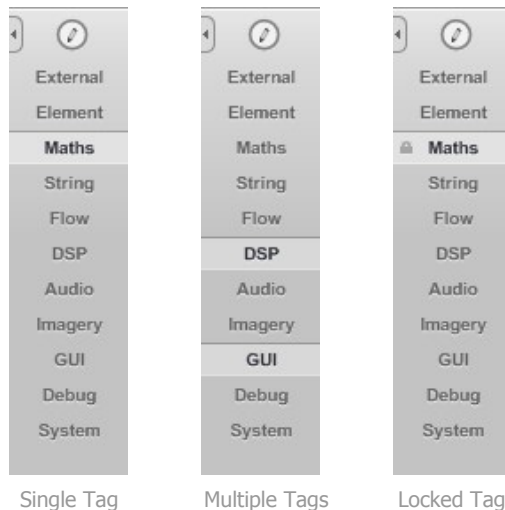
If you click on a different tag the previous one will automatically be deselected. To filter by more than one tag, hold SHIFT as you click.

### Locking Tags

It may be that you only want to work with a particular set of components. In this case you can lock a tag in place by double-clicking on it. The tag will remain selected until you click on it again.

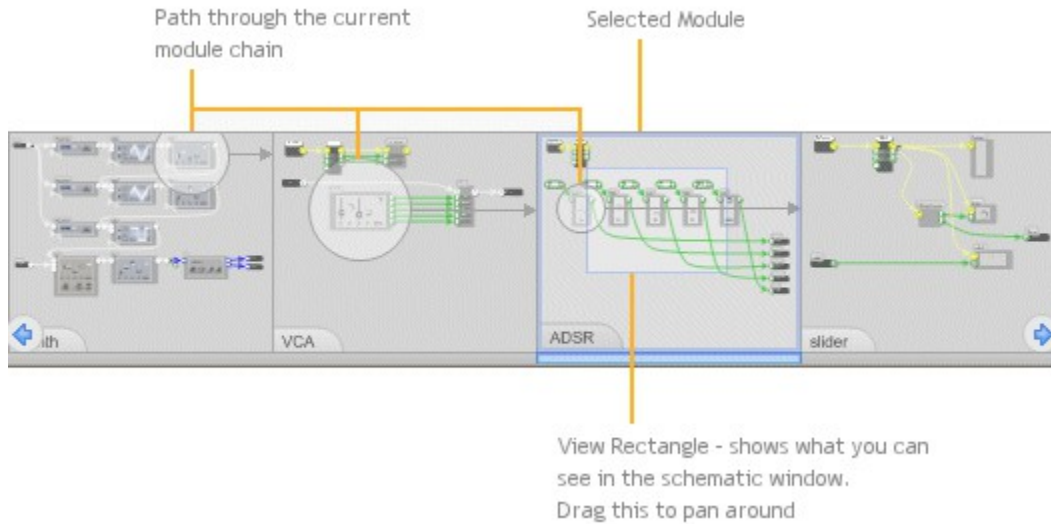
### Clearing Tags

You can clear tags by clicking on them or you can clear all tags in one go by right-clicking anywhere on the tag bar.





# Navigator



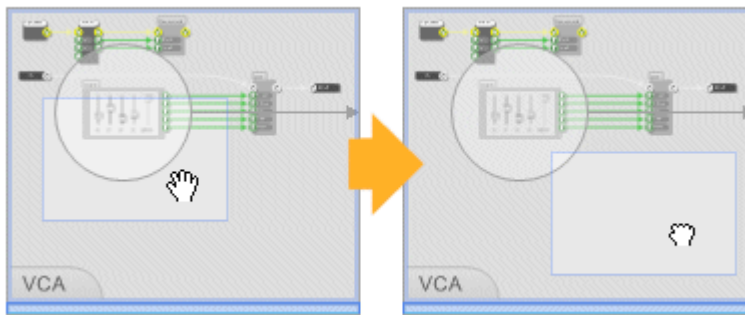
The Navigator allows you to see where you are in your schematic. This is extremely useful when you have several layers of modules because the Schematic Window only shows the module that you're editing.

## Jumping

The Navigator is not just visual, it's interactive too. You can jump to any module in the current chain by clicking on it. You can also use the number keys to do this (providing you are not using the PC keyboard for MIDI input). Number 1 will always take you to the Top level. The PGUP and PGDN keys will move you up and down the current hierarchy.

## Panning

The current module in the chain is highlighted. Inside you'll see a rectangle representing the portion of the module that you can see in the schematic window. This is called the View Rectangle. You can click and drag this to pan around the schematic.

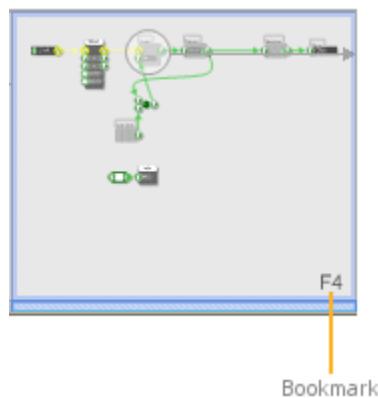


## Bookmarks

You can bookmark particular parts of your schematic so that you can easily return to them. To do this simply go to the part of your schematic you want to bookmark, hold SHIFT and press one of the function keys (F1...F15). The Navigator will show the associated function key.

To return to the bookmark at a later time just press the function key. You can also then flick back to the point that you came from by pressing the function key one more time.

To remove a bookmark all you need to do is jump to it then hold SHIFT and press the same function key again.



# Schematic Window

The schematic window is where everything comes together. Components can be dragged here from the toolbox. You can connect components by dragging links between them.

The schematic window has all the features that you'd expect in an editor: undo, copy and paste, multiple selection, zooming, and context sensitive help are all fully supported.

## Zooming

The Schematic Window is fully zoomable. There are several ways you can zoom the schematic window:

### Mouse Wheel

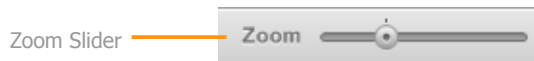
The easiest way to zoom is to use the mouse wheel. Hold down CTRL then roll the mouse forward to zoom in and back towards you to zoom out.

When mouse wheel zooming, the software will zoom towards or away from the point where you position your mouse. This allows you to zoom and pan in one movement.

If your mouse supports it, you can also return the schematic to the default zoom level by pressing the mouse wheel button. The schematic pan position is also returned to its default.

### Zoom Slider

The Zoom slider is located on the right-hand side of the tool bar. Moving the slider to the left will zoom out and to the right will zoom in.



The default zoom position is shown as a small notch above the slider. The slider will snap to this position as you move past to make it easy to return to the default. The default zoom level can also be achieved by double-clicking on the slider.

### Context Menu

Right-click on an empty part of the schematic and select Zoom In or Zoom Out

### **Keyboard**

Simply press the + (zoom in) or - (zoom out) keys.

### **Scrolling**

You can use the mouse wheel to scroll the schematic window up or down. Hold SHIFT and move the mouse wheel to scroll horizontally.

### **Panning**

In addition to using the Navigator, you can quickly pan around the schematic window by grabbing the background and moving it using your mouse. To do this:

1. Left-click on an empty part of the schematic and hold the mouse button down.
2. Now right-click and hold that button down too. The cursor will change to show a hand grabbing the schematic.
3. Now move the mouse and the schematic will move with it. Release both mouse buttons to finish dragging.

You can also pan by holding the space bar down. The cursor will change to the hand. Keep the space bar down and drag the schematic around.

### **Resetting Zoom and Pan**

If you ever need to reset the zoom and pan back to their default positions you can do this by selecting Reset from the View menu or by pressing the HOME button.

# 3 Components & Links

THE BASIC ELEMENTS OF A SCHEMATIC

Components and links are the bricks and mortar of a FlowStone schematic. Understanding how to edit and manipulate these basic elements is essential for working with the software as most of the interaction you'll have with FlowStone will be through the schematic.

We introduced the concept of components and links right at the start of this guide. We'll go over this again now but in a little more detail this time.

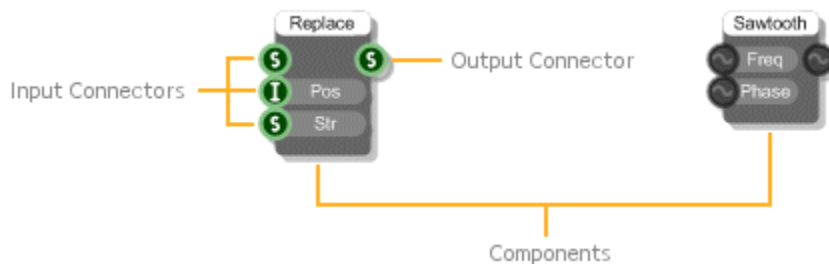
## Components

Components provide the functionality in a schematic. Each one performs some well defined task. They are represented by rectangular blocks with circular adornments called Connectors which may appear on the left-hand or right-hand sides (or both).



The connectors on the left-hand side are called Input Connectors. These provide the component with information that it uses when performing it's defined task.

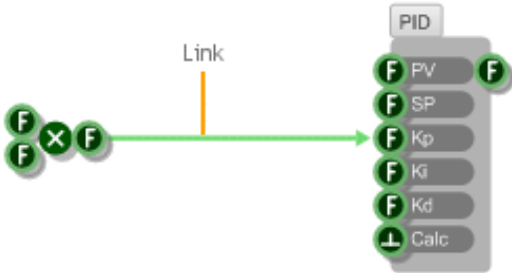
The connectors on the right-hand side are called Output Connectors. These provide information about the outcome of the of the task performed by the component.



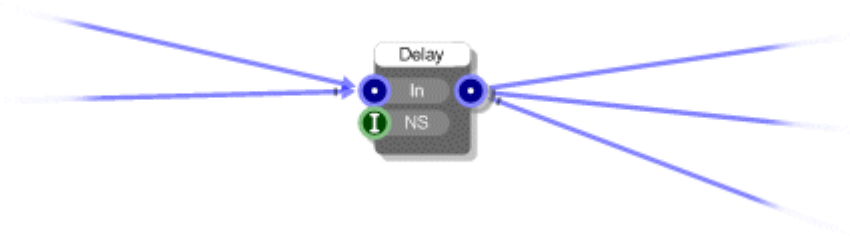
In the example above the String Replace component takes 3 inputs: the original text string, the position at which the replacement is to be made and the text string to be inserted. The output is the modified string.

### Links

Links define how information flows between components in a schematic. A link passes from the output connector of one component to the input connector of another component. The direction of information flow is generally left to right or from output to input but in some cases information can pass from input to output as well.



You can have multiple links from the same output connector. You can also have multiple links passing to the same input connector. However, you can only have one link going from the same output connector to the same input connector.



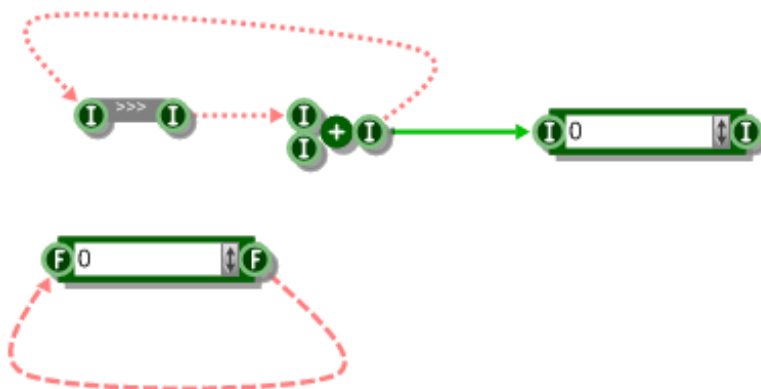
When multiple links arrive at the same input connector then, for some connector types, the data from each link is added together. In other cases the data is just merged.

## Infinite Feedback

Links can be connected from a component into itself to create feedback paths. However, in some circumstances this may create an infinite feedback loop. If this happens the software will 'freeze' the affected link(s). A frozen link is shown as a red dashed or dotted line.

Frozen links continue to function but data flow through them is reduced (but not restricted altogether) to prevent the software locking up.

[Note that links between stream, poly or mono connectors never become frozen as feedback is always allowed in these cases.]



A dashed line indicates that the infinite loop is in the forward direction i.e. left to right as changes are triggered. A dotted line indicates that the direction is backwards i.e. right to left as data is being gathered.

Frozen links are relatively rare. However, when they do occur they can be easily located because every module that contains a frozen link somewhere below it is highlighted with a red and white striped border and an adornment saying that there is an error inside.



If you follow the red modules down the hierarchy you'll eventually get to the frozen link(s).



**Handling Frozen Links**

So how do you deal with frozen links once you've found them?

You may be able to get the behaviour you were wanting to achieve by inserting a Trigger Blocker or by using a Sample and Hold component that triggers from somewhere else.

It could be that the feedback may only be required in one direction at a time so you could use select components to make sure that data is only sent in one direction or the other.

Some components have built in handling to prevent infinite feedback from occurring. The Switch components for example (Float Switch, String Switch etc.) have this kind of behaviour. Inserting one in between frozen links can often solve the problem.

Sometimes a frozen link can simply be removed and the same functionality is retained. The link may have been added as overkill or in error.

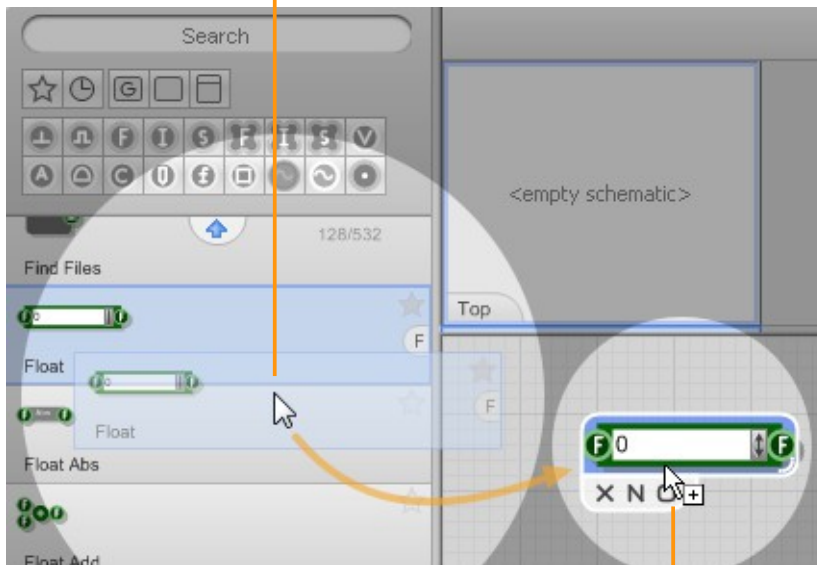
# Components

## Adding a Component

### Dragging From the Toolbox

To add a component to a schematic, simply go to the toolbox, pick the component you want and drag it into your schematic.

Click on the component you want then hold and drag across to the schematic window

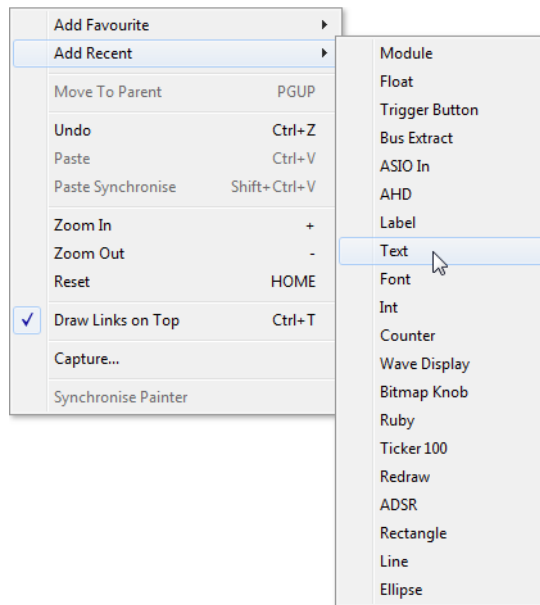


Release the mouse button to drop the component onto the schematic

## Adding from the 'right-click' Menu

You can also add components from the Favourites or Recent lists by right-clicking on the schematic - saving you from having to visit the toolbox altogether.

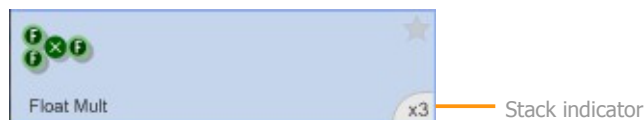
Right-click on an empty part of the schematic then choose Add Favourite or Add Recent depending on which list you want to use, then just choose a component from the resulting popup sub menu.



## Stacking Components

Sometimes you may want to add more than one component at a time. You can do this by stacking up components before dragging them all to the schematic.

To increase the number in the stack just click on the component the same number of times as the number you require. You'll see a counter appear in the bottom-left corner of the component in the toolbox.

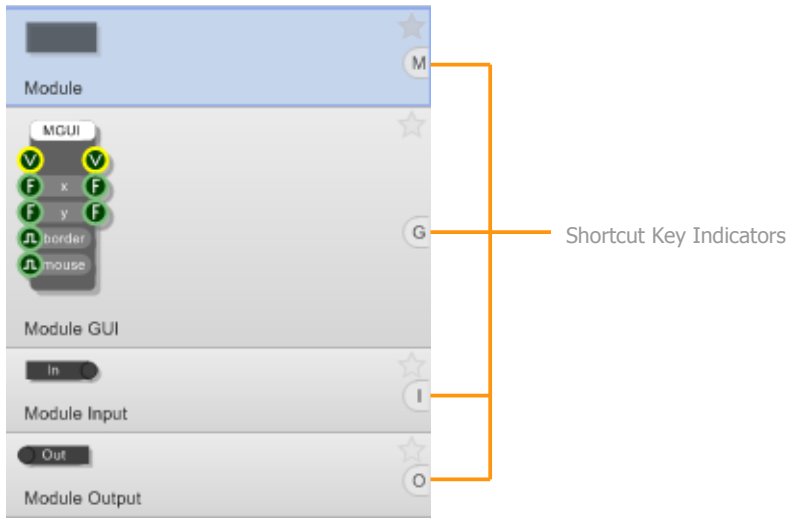


To decrease the number in the stack right-click on the component. When you have the number you need just drag them across as before.

### Keyboard Shortcuts

As you become familiar with the software you'll find that there is a certain small group of components that are used much more frequently than others. In order to save you time constantly going back and forth to the toolbox you can make use of keyboard shortcuts.

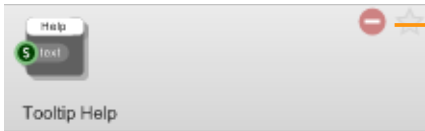
Where a component has a shortcut key assigned this is displayed on the component in the toolbox.



### Disabled Components

On some occasions it is not possible for you to add a particular component to your schematic. This is because some components are only allowed once within a particular module or the whole schematic. This can also occur if you're running the Free or Enterprise editions and you reach one of the limits for components of a particular type.

If this happens the component will have a red 'no entry' symbol over the top of it.



Symbol indicates that the component cannot be added at the current location in the schematic

### Hitting Return

One final way to add components from the toolbox is to use the return Key on the keyboard. This is handy if you've just done a search say. Hitting return when in the toolbox (or the search box) will always add the currently selected component to the schematic.

## Selections and the Action Panel



To select a component simply click on it. When you do this you'll see the action panel. This contains a number of buttons. the 'X' button will delete the component. The 'N' button will allow you to name the component.

Other buttons may appear depending on the type of component selected - more on these later.



The commands in the action panel can also be invoked from the menu bar and the context menu. To get the context menu, right-click on the component.

Note that as well as the selected component(s), any links that start and end in a component that's selected will also be considered part of the selection

## Moving Components

### To move a component:

1. Move your mouse over it. The cursor will change to Move/Select.
2. Click and hold the mouse button. the cursor will change to Move.
3. Drag to the desired position and release the mouse button.



1. Move mouse over

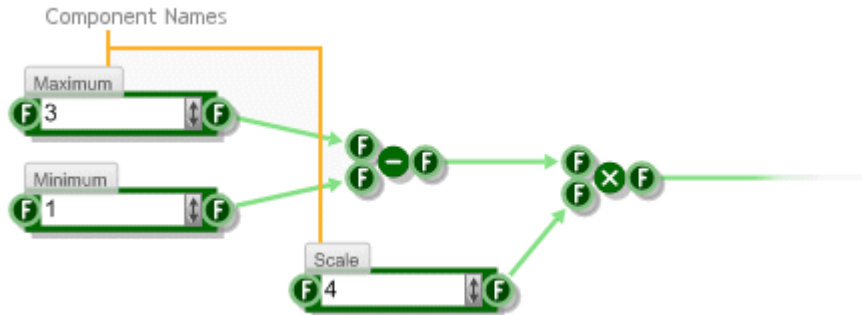


2. Click, hold and drag

For fine movements you can use the nudge feature. With a component selected use the cursor keys to move it up, down, left or right by one grid square.

## Naming Components

You can give a component a name. This is just a label that can be used to remind you the role of a component in your schematic.

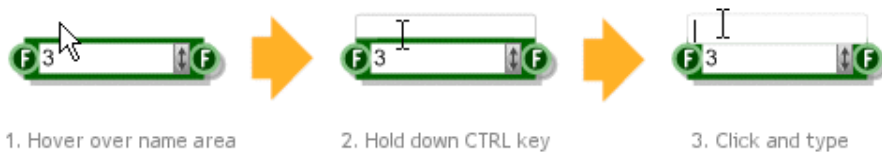


To give a component a name, click the 'N' button on the action panel or right-click on the component and select Rename. Alternatively you can select the component and press CTRL+R.

An edit box will appear above the component. Type the name here then press ENTER, TAB or just click on another part of your schematic. You can press ESC at any time to cancel.

### Direct Interaction

For a more interactive way to add names, hover the mouse over the area just above the component and hold CTRL. The frame of the name box will be highlighted. Click in the box and proceed as before.



### Editing Names

To edit a name you can use the action panel and right-click options as before or you can just click on it as shown below.



## Deleting Components

Deleting a component is extremely easy. The quickest way to do this is to select it then press the DEL key. You can also press the 'X' button on the action panel or right-click on the component and select Delete.

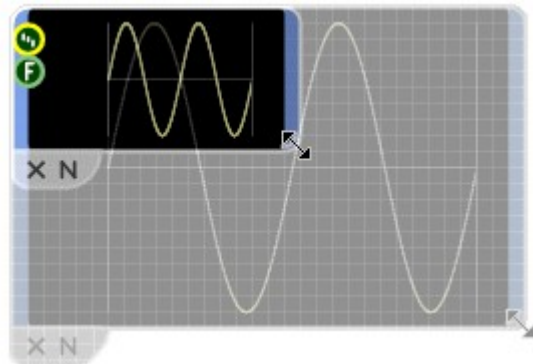
## Resizing

Some components can be resized to make them bigger or smaller. Some components only resize horizontally, others resize vertically as well.

If a component can be resized, the resize control will appear in the bottom-right corner of the selection highlight. The control is a white arc that traces the corner of the selection.



Resize control



To  
resize,  
move

your mouse over the resize control. The mouse pointer will change to the resize cursor. Click and hold then drag to resize.



## Multiple Selections

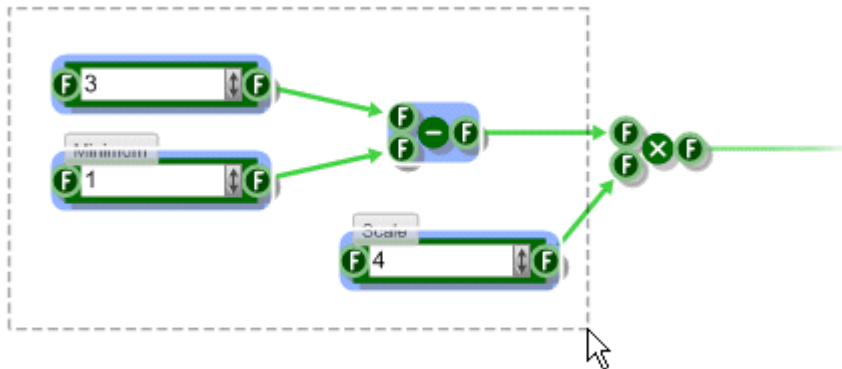
You can select multiple components at the same time. This is useful if you want to move a group of components but maintain their relative spacing or if you want to delete a whole load of things in one go.

There are two ways to create a multiple selection. The first way is to hold down SHIFT and then click on each of the components in turn. If a component is already selected, clicking on it will remove it from the selection.

A quicker way to make a multiple selection is to drag select. This involves dragging out a rectangle to enclose all the components that you want to select.

### To drag select:

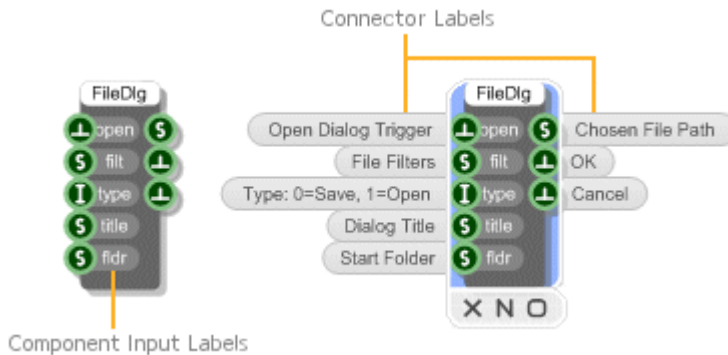
1. Click on a blank part of the schematic to the top-left of the components you want to select.
2. Holding the mouse down, move the mouse down and to the right. You'll see a dotted rectangle appear as you move.
3. The components will show as selected as you move the mouse so when the selection is what you want release the button.



## Connector Labels

Most components have a short label for each of their input connectors. There isn't enough room on the component body for both inputs and output labels. The outputs are usually less ambiguous and fewer in number and so input labels are usually preferred.

Component labels are usually very short and provide a quick reminder of what each connector is for. If you need more information you can select the component (by clicking on it). Where available, additional connector labels will appear to the left and right of the component.



## Cut, Copy and Paste

FlowStone supports the standard Cut, Copy and Paste operations for moving or duplicating parts of a schematic. Both single and multiple selections can be cut, copied and pasted.

You can access these operations from the Edit menu or by right-clicking on a selection in the case of Cut or Copy or anywhere on your schematic in the case of Paste.

Copying a selection will place a duplicate of the components and links on the Clipboard. The clipboard is an invisible buffer that retains what was last copied to it. Cutting is like a combined copy and delete as the original selection is removed from the schematic.

Having Cut or Copied a selection you can then go to any other part of your schematic or to any other schematic you have open and paste in a copy of the clipboard contents. Copies are pasted at the current mouse location. You can hold ALT while pasting to have pasted elements stack up at the location of the source elements.

# Links

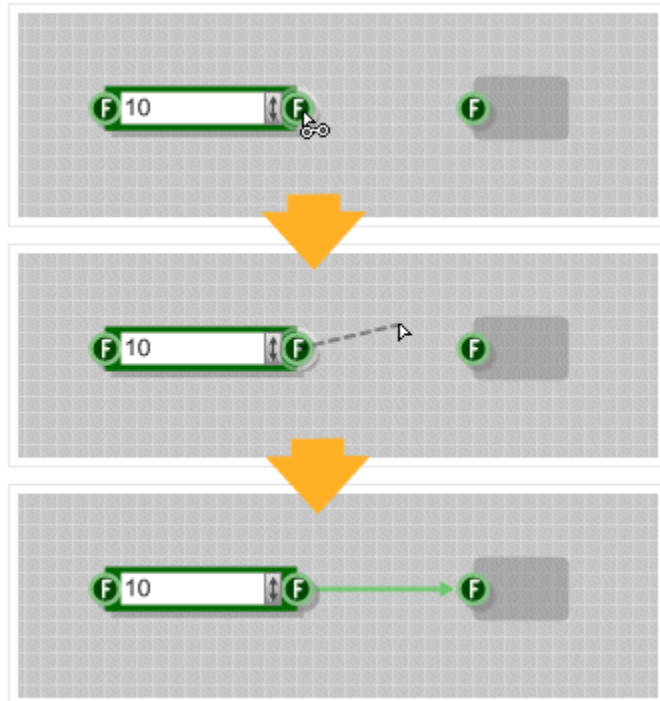
In this section you'll learn how to move a link to another connector or delete it completely. You'll also see how to change link order and how to bend a link.

## Creating a Link

Links must start at an input connector and end at an output connector. You can link another component or to itself, to create a feedback path say.

### To create a link:

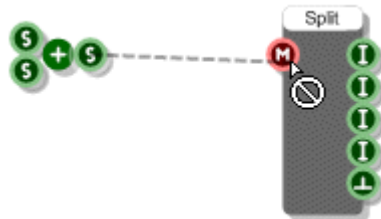
1. Move your mouse over an output connector. The connector highlight will show as you pass over it.
2. Click and hold, then drag towards the input connector. You'll see a dotted line representing the potential link.
3. Keep dragging until your mouse passes over the input connector. The potential link will snap to the connector to indicate that the link can be formed.
4. Release the mouse button to create the link.



## Allowed Links

Sometimes it is not possible to make a link between a pair of connectors. When this happens the mouse cursor will change to show this.

In general links can be made only between connectors of the same type. However, there are several exceptions to this. For example, Floats and Ints can be connected so can Floats and Stream connectors. A complete description of all these exceptions can be found in Chapter 5 Converting Between Data Types on page 103.



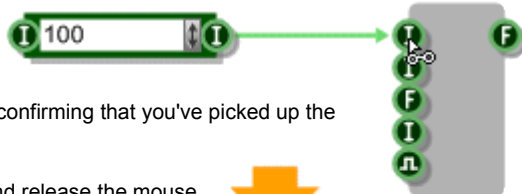
## Moving a Link

You can move the end of a link from the input connector to another input connector on the same component. You can also move it to an input connector on a completely different component.

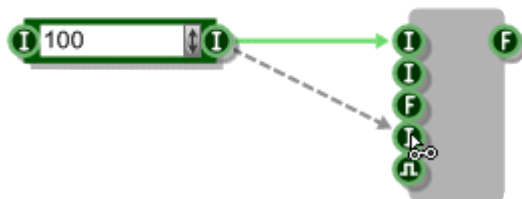
To move a link:

1. Move the mouse pointer over the input connector where the link ends. The cursor will change to the linking pointer.
2. Click and hold. You'll see a dotted outline confirming that you've picked up the link that you wanted.
3. Drag the link to another input connector and release the mouse button.

1. Move over connector and click

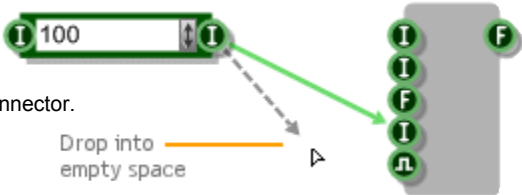


2. Drag and drop

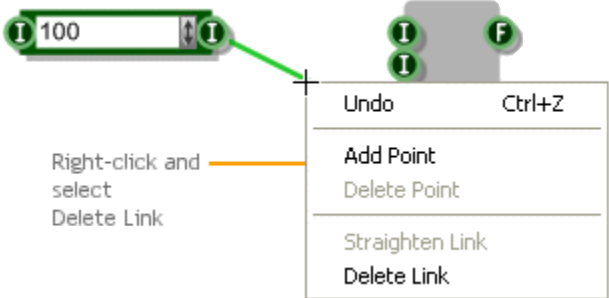


### Deleting a Link

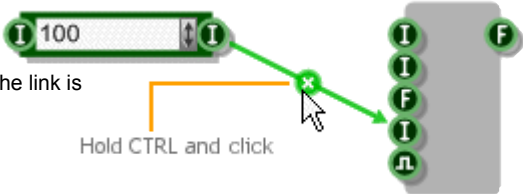
There are several ways to delete a link. You can pick it up and drop it into empty space. Just move it from the input connector (as described above) but don't link it to another connector.



You can also right-click on the link and select Delete Link from the context menu.



For really quick deleting, hold down CTRL. When your mouse passes over a link, the delete button will appear. Click the button and the link is gone.



## Link Order

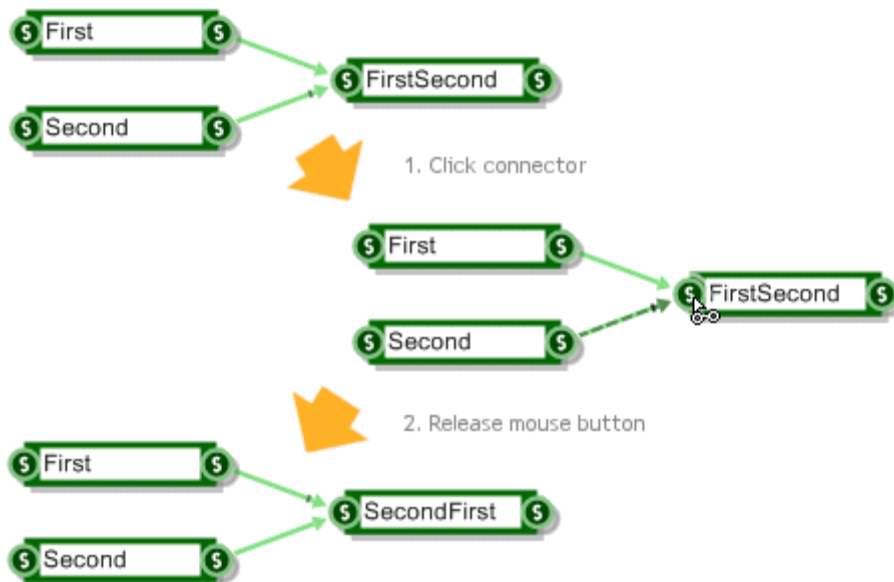
When there are many links ending at the same input connector, the order of the links can affect the behaviour of your schematic. For example, if you have two links that are passing String data to a string connector, the first string to arrive will have the second one appended to it.

The order of a link is indicated by an order marker. This is a series of notches perpendicular to the link at it's end point. The first link in the order has no marker, the second has a marker with one notch, the third has two notches etc.



### Changing the Order

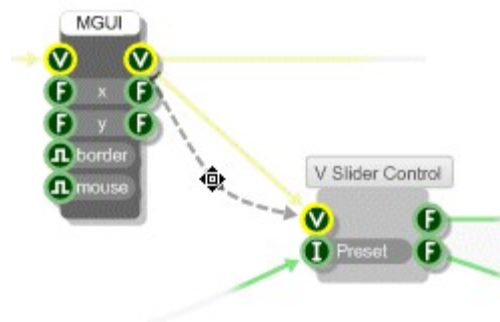
It's useful to be able to change the order of links at an input connector. To do this simply click on the connector and the current top link (the last in the order) will become the bottom link (the first in the order).



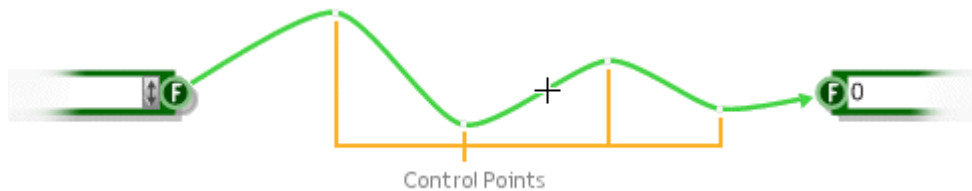
## Bending Links

In a complex schematic with many components, sometimes a straight line just won't do. Often you'll struggle to avoid creating links that run over components or each other.

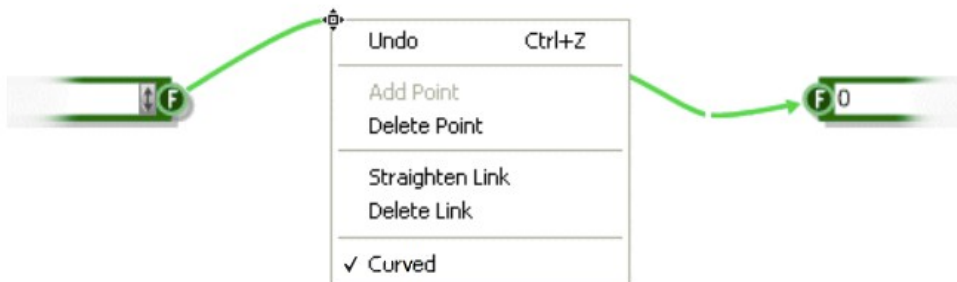
To get round this problem, you can bend any link to make your schematic clearer and easier to read. All you have to do is click on the link and pull it into place.



Each bend you put in a link creates a control point. The control points define the path of the link. If you move your mouse over a link it becomes highlighted and the control points are shown as small squares. The control points will either be black or white depending on the colour of the link.



You can drag the control points around to re-shape the link. You can also delete a control point by right-clicking on a point and selecting Delete Point from the context menu.



To remove all the control points and make the link a single straight line again, select Straighten Link from the context menu.

## Resolving Control Points

If you have many links whose control points are very near each other, or worse still points that completely coincide with each other it can be very difficult to resolve one point from another and grab and drag the one you want.

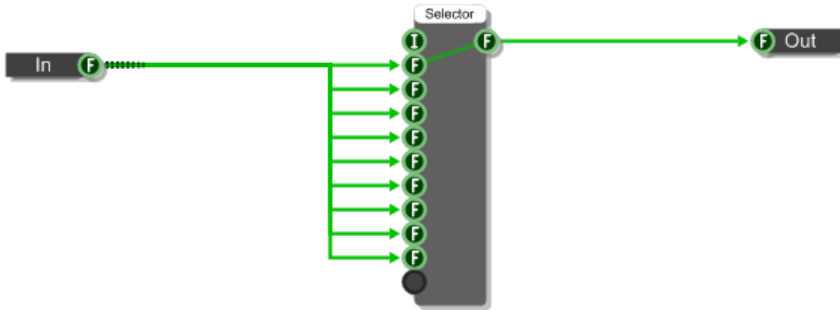
FlowStone has a very simple but effective method for helping with this problem. First hover the mouse over the link that contains the control point you want to manipulate. You'll notice that the link goes dark and starts to pulse.

Keeping the mouse near to the link at all times, you can now move to the control point you want. While the link is highlighted in this way FlowStone will assume that you are trying to edit the points on that link only.



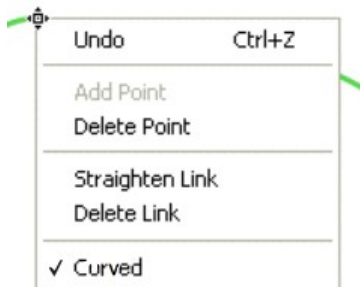
## Straight Links

In schematics with lots of links that overlap or follow a similar path it can sometimes be neater to have the links follow a straight line path instead of a curve.



You can do this in one of two ways. If the link is still a single line then if you go to bend the link (as described earlier) but hold SHIFT + CTRL as you click to add the first control point, the link will now be a non-curved (straight) link. If you now add more control points to the link there is no need to use SHIFT + CTRL as the link has already been designated as non-curved.

If you have already added control points to make a curved link then you can toggle between curved or non-curved by right-clicking on the link and toggling the Curved check box.

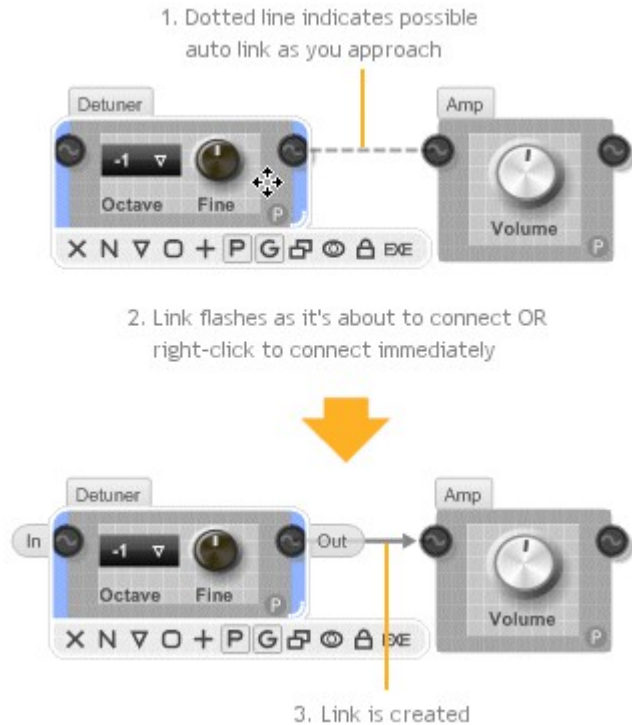


## Auto Linking

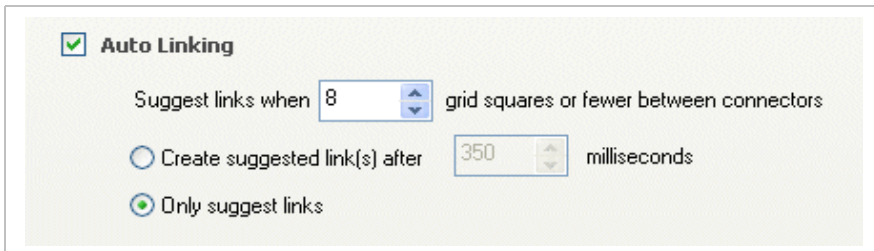
To help you build your schematic more quickly we have the Automatic Linking feature. When this feature is switched on you can create a link between two connectors by moving a component so that the required output connector is on the same horizontal and sufficiently close to the required input connector.

When these conditions are satisfied a dotted line will appear. This suggests where the link would be created. If you then pause for a fraction of a second the link will flash and become permanent.

If you don't want to have to wait for the link to be created you can instantly accept the suggested link by clicking the right mouse button.



There are several options for Auto Linking. These can be adjusted by selecting Schematic from the Options menu.



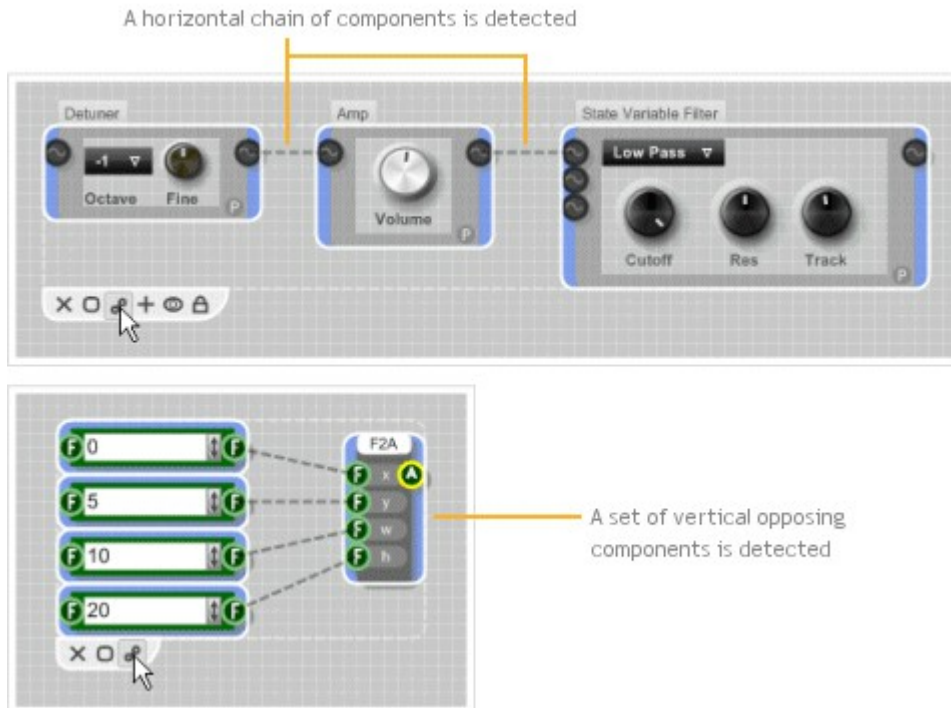
You can change the time between suggestion and link creation you can also set how close connectors have to be before a link is suggested. If you only want to use the right-click method to make a link then you can set the auto linking to only suggest links and of course you can turn the feature off altogether.

## Smart Linking

If you need to link several components together quickly then you can use the smart linking feature. Simply position your components in a logical way, select them then either click the Link Components button on the action panel or select Link Components from the Schematic Menu.

The software will make a best guess at how you want the components to be linked together. If you use the action panel button to do this then when you hover the mouse pointer over the button the software will indicate the links that would be created.

Smart linking is useful if you have several components that you want to link to one other component or if you want to chain components together.



## Removing Multiple Links

You can remove all the links in a selection in one go. Simply right-click on the selection and choose Remove Links from the pop-up menu.

This can be very useful when you have a large section of schematic that you need to disconnect

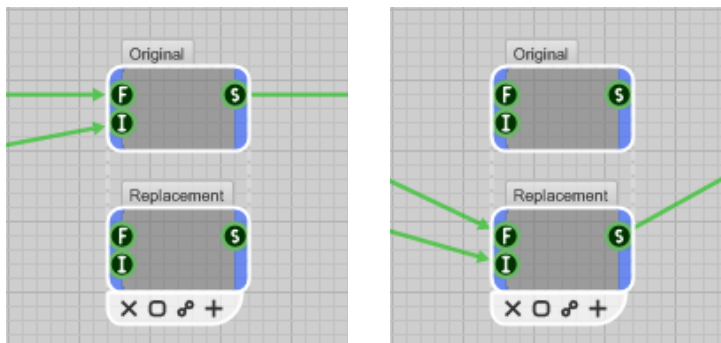
Undo	Ctrl+Z
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Paste Synchronise	Shift+Ctrl+V
Rename	Ctrl+R
Delete	Del
Remove Links	Shift+Ctrl+L
Make Module	Ctrl+M
Property	Shift+Ctrl+P
Synchronise Painter	

## Swapping Links

Sometimes you might want to replace a component in a schematic with another component. The most common reason for doing this is when you have a newer version of a module that you want to use in place of an existing one (see the next chapter to learn about modules).

You can do this by hand by moving all the links one by one but this is time consuming and can lead to errors if you forget a link or where it was in the link order.

To help we have the Swap Links feature. Simply select the two components, right-click and choose Swap Links from the popup menu. The links from one component will swap to the other and vice-versa. If the components don't have the same number of inputs or outputs then the software will transfer the links it can and drop any that it can't.



## Wireless Links

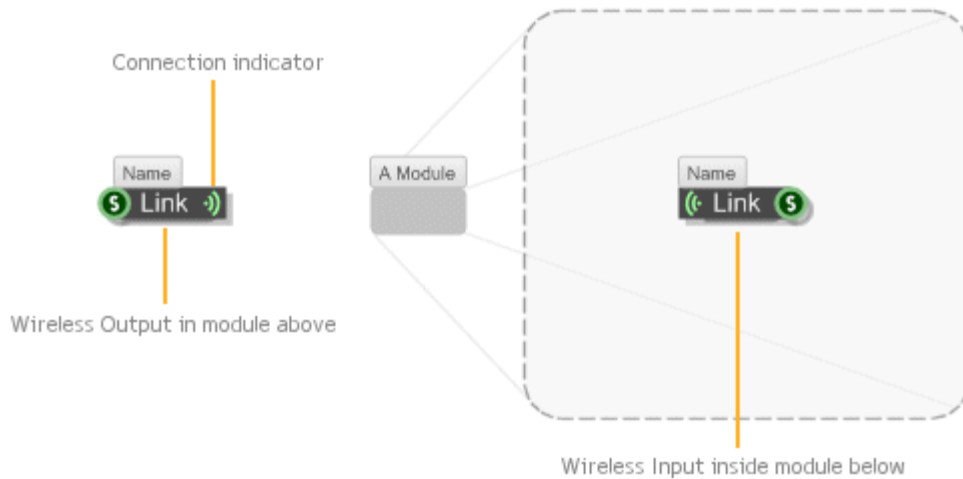
Wireless links provide a mechanism for passing data through the module hierarchy without having to create any physical link. They are just like wireless networks in the real world. You have a transmitter called a Wireless Output at one end and a receiver, a Wireless Input at the other end.



A connection is established between a Wireless Output and a Wireless Input only if the following three conditions are met:

1. The Wireless Input must appear in a module **below** the Wireless Output in the hierarchy
2. The Wireless Input and Output must have the **same label**
3. The Wireless Input and Output must have the **same connector type**

When a link is established the connection indicators on the Wireless Input and Output will light up.



Wireless links only work down the module hierarchy, you can't link back upwards. Also, the range of a wireless output only extends as far as the next wireless output below it which has the same label and connector type.

The same wireless output can connect to multiple wireless inputs and vice-versa so long as they conform to the 3 criteria described above.

There is another wireless component called a Module Wireless Output. This is used to make a wireless module. See the Modules section for more information about this.



## Follow Wireless

Wireless links are great for keeping your schematics tidy but they have one downside: they can be quite difficult to follow. To help with this we have the Follow Wireless feature.

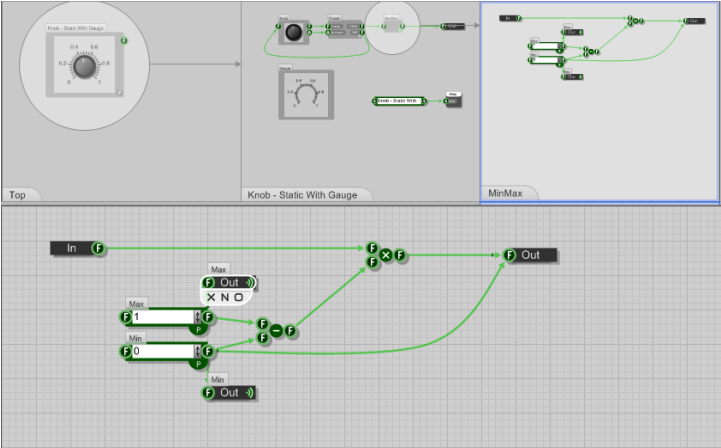
To use this simply select a Wireless Input or Wireless Output and either right-click and select Follow Wireless from the menu or press the TAB key. The schematic will jump to the first connecting wireless Output or Input. The path of the link will also be indicated on the Navigator.

The connecting wireless component will be highlighted and a label below it will show how many matching wireless components there are and the index of the one that is currently highlighted.

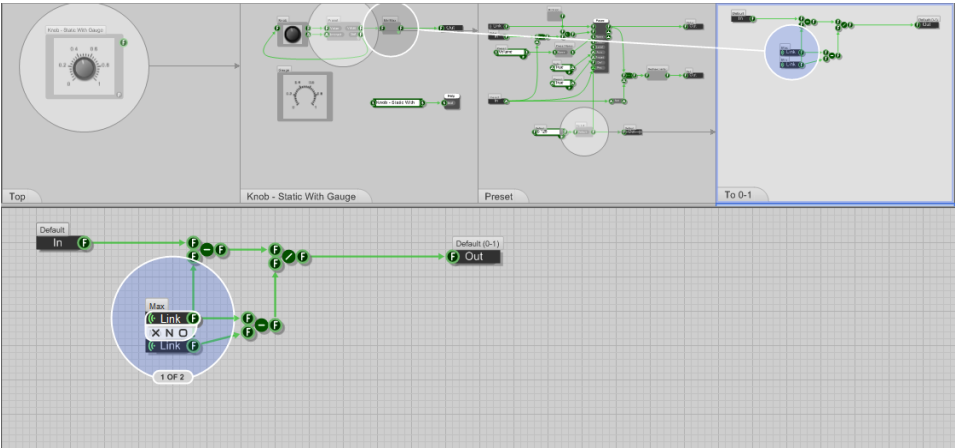
You can then right-click on an area of empty schematic and select Next Wireless Component or press TAB again. Eventually you will TAB back to the component you started with – the source Wireless component.

**EXAMPLE**

Below we have one of the standard toolbox modules. We've moved into a sub module called MinMax. We want to see what the Module Wireless Out component called Max is connecting to. All we do is select the component and press the TAB key.



The schematic jumps to the first Wireless Input component that that connects to Max. We can see from the highlight that this is component 1 of 2.



## CHAPTER 3

The schematic shows the path through the schematic to this component and the wireless connection. Because we selected a Module Wireless Output the connection is shown from the point of transmission ie the parent module.

We can clearly see that the receiver is inside the 'To 0-1' module inside 'Preset'.

If we continue to tab we'll go to the second Wireless Input and then back to the source. Again, because we're following a Module Wireless Output the source is shown as the parent module.

To stop following wireless connections simply click on the schematic or perform some other kind of edit operation.



# 4

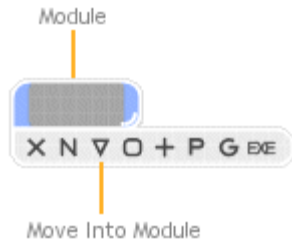
# Modules

MANAGING COMPLEXITY

You may recall from the introduction that components are broken into two types: primitives and modules. A primitive has a predefined behaviour. It's a black box whose behaviour can't be changed.

A module on the other hand has its behaviour defined by a schematic. You can add inputs and outputs and the internal behaviour can be modified to do virtually anything you want.

The module component can be found in the toolbox under the Module filter group. Drag one into a schematic and you'll see that an empty module is just a grey box.



The action panel for a module has addition buttons to reflect the additional operations that you can perform on it. Of these the most important is the Move Into Module button (represented as an inverted triangle). By pressing this button you can go inside a module to view its schematic and from there define the module's behaviour.

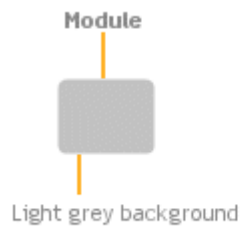
Modules are very simple in concept but they are extremely powerful. They can be used to partition off functionality into pieces that can be reused again and again. They are essential for managing complexity in anything but the very simplest of schematics.

# Key Differences

Apart from the fact that a module has its own schematic there are a few other key differences between Modules and Primitives. This section outlines those differences. More details are given in subsequent sections.

## Appearance

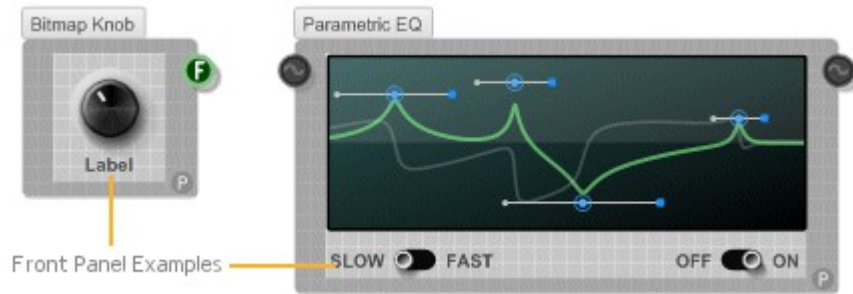
First off modules can be differentiated from Primitives by the way they look. A module has a light grey background whereas a primitive has a dark grey border with a drop shadow and title bar.



A module can also have additional adornments indicating syncing, properties or wireless capabilities. More on these later.

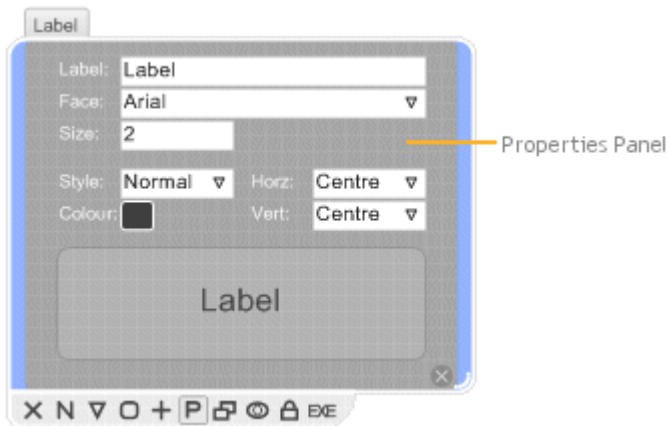
## Front Panel

Any module can have a front panel. This is shown on the module itself and provides an interactive surface for adding controls or displaying graphics. The front panel graphics and interaction is all defined in the module's schematic.



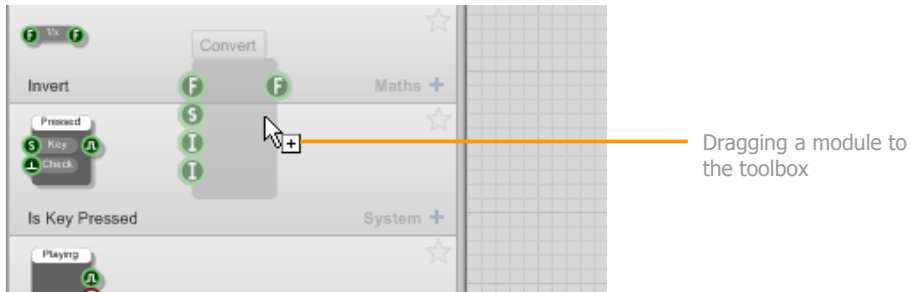
## Properties

If you want to control the behaviour of a module in a more interactive manner then instead of using input connectors you can define a properties panel. Again the properties panel is defined by the module's schematic.



## Toolbox

The final key difference between a module and a primitive is that a module can be dragged to the toolbox for use at a later date. You can drag a module on it's own or you can drag all selected modules in one go.



When you do this the module will automatically acquire any tags you have selected at the time. If no tag is selected then the module will have no tags added automatically.

# Basic Operations

You can do everything to a module that you can do to a primitive. You can delete it, name it, move it around etc. The key difference between a module and a component is that you can go into a module and view or edit it's schematic.

## Moving into a Module

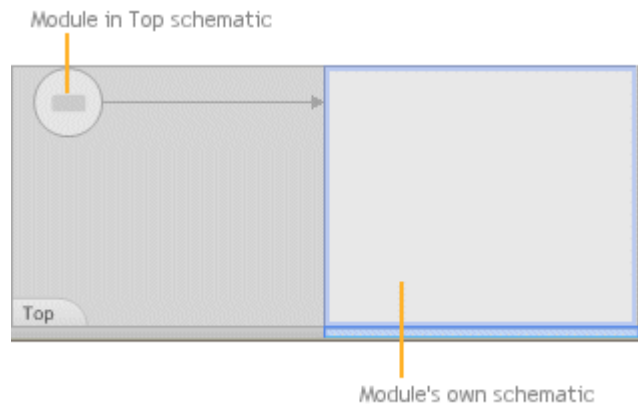
With most operations in FlowStone there are many ways to do the same thing. Moving into a module is no exception.

To move into a module, either:

1. Click the Move into Module button on the action panel
2. Double-click on the module
3. Right-click on the module and select Move into Module from the context menu
4. With the module selected, press the PGUP key

The Schematic Window will change to show the module's own schematic, the Navigator will also change to reflect this.

To move back out of the module again, either right-click and select Move to Parent or press the PGDN key. You can also double-click on an empty part of the schematic or of course use the Navigator.

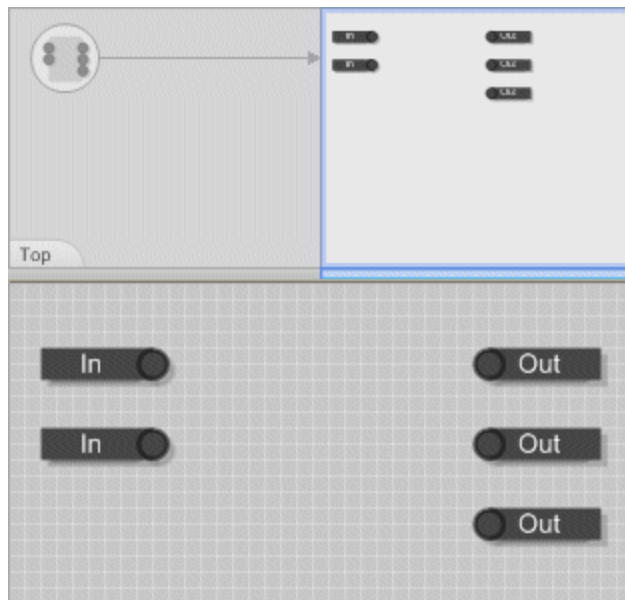


## Inputs and Outputs

To get information into and out of a module you need to add inputs and outputs. To do this, just drag them in from toolbox in the usual way (you'll find them under the Module group). The example opposite shows a module with two inputs and three outputs.

For a quick way to add Module Inputs and Module Output components you can use keyboard shortcuts. A select few components are used much more frequently than the others. To save you going to the toolbox each time you can just press a particular key and a new component is dropped at your mouse position.

To add an input press 'I', to add an output press 'O'. Note that these shortcuts will not work if you are using your PC keyboard for MIDI input.



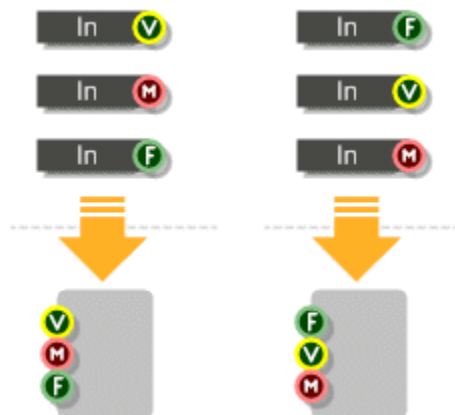
### Input/Output Ordering

It's usual to keep the inputs on the left and the outputs on the right but it's up to you.

What does matter is the position of inputs relative to one another as this determines the order of the inputs on the module itself. By swapping the vertical positions of the input and output components you can easily change the order of inputs for a module.

If components have the same vertical position the horizontal position is taken into account, with the left-most input being higher in the order.

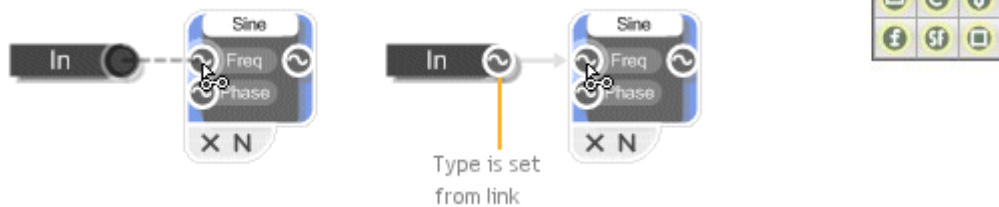
The same rules apply to the outputs.



## Template Connectors

The connectors for the Module Input and Module Output components are what we call Template Connectors. These have no defined type. There are two ways you can assign a type to a template connector:

1. Automatically pick up the type from another connector by creating a link to that connector.
2. Set the type explicitly by right-clicking on the connector and choosing the type from the context menu.



## Input and Output Names

If you give your inputs and outputs names then these will be displayed when you select your module. In addition, input names will be automatically displayed on the module body itself.

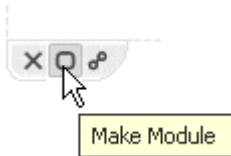


If you want to be even more complete you can provide both short and long descriptions. To do this you need to name the module input or output component in the form `<long name>\n<short name>`. So for example, the label “Maximum height of rectangle\nmaxh” would give an input or output a short name of “maxh” and a long name of “Maximum height of rectangle”.



## Make Module

You'll find that as you build a schematic you'll want and need to section parts off into separate modules in order to manage the increasing complexity. You can do this easily using the Make Module feature. Simply select a number of components then click the Make Module button on the selection action panel (or right-click on the selection and select Make Module).



The software will create a new module and place the selected components and links inside (the layout of all components is preserved). A module input or output will be created for each link that connects to the selection. The old links are then connected to the inputs and outputs on the new module instead of the selection.

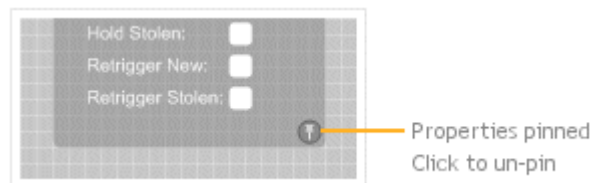
## Properties

We mentioned earlier that modules can have properties. These affect how the module looks or behaves. When a module has properties the properties button will be shown in the bottom-right corner of the module. This is in the form of a small circle with a letter P in the centre.



When you click the properties button the module will expand to show the properties panel. With the panel open you are free to make any changes you want. The panel will then remain open until you click on some other part of your schematic.

If you want to make the properties panel display permanently you can pin it open by holding CTRL as you click the P button. The P will change to a pin icon to show that the properties will stay open.



## Wireless Modules

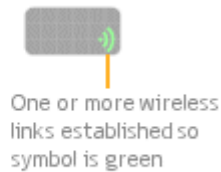
Most modules will have fixed output connectors that you physically link up to other connectors. However, it is sometimes useful to make a module output wireless. Instead of using a Module Output component you use a Module Wireless Output component.



By adding wireless outputs to your module the module becomes a wireless module. The module will behave in the same way as a Wireless Output component establishing wireless links with matching Wireless Input components lower down in the module hierarchy.

As with Wireless Outputs a match is determined by the type of connector and the component label.

Wireless modules can be identified by the wireless symbol which appears on the module body. This will appear grey when no links have been established. However, if one or more Module Wireless Outputs within the module have established connections with matching Wireless Inputs the wireless symbol will light up.



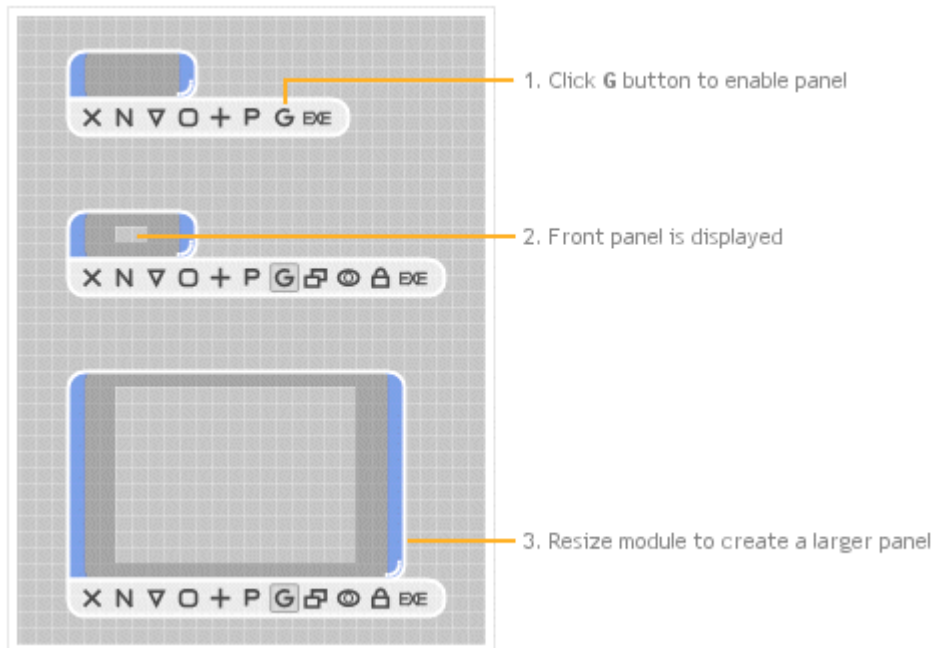
# Front Panel

Every module has the option of having a front panel. Front panels allow you to add interactive elements to your schematic and they are the mechanism by which you provide a graphical user interface (GUI) for your creations.

## Enabling the Front Panel

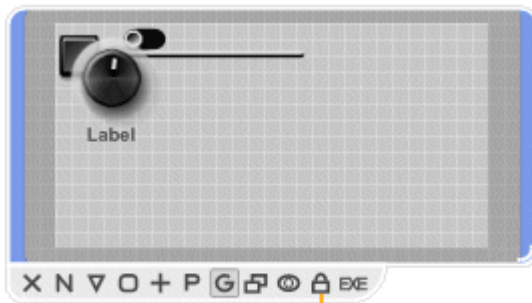
The front panel is shown on the module itself. By default the front panel is disabled. To enable it, select the module then click the **G** (GUI) button (you can also right-click on the module and select Enable Front Panel).

If you resize the module the front panel will resize accordingly.



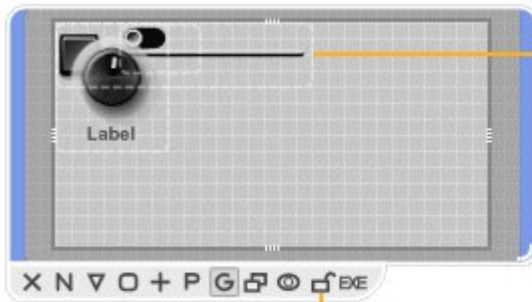
## Editing the Front Panel

When you place knobs, sliders and other controls inside a module that has a front panel then these items will instantly appear on the front panel. However, everything will be stacked up in the top-left corner.



Click to unlock the front panel

In order to arrange the items you need to unlock the front panel. First select the module then click the padlock button on the action panel (you can also right-click on the module and select Edit Front Panel or hold CTRL and press E).



Panel items that you can interact with have dotted outlines

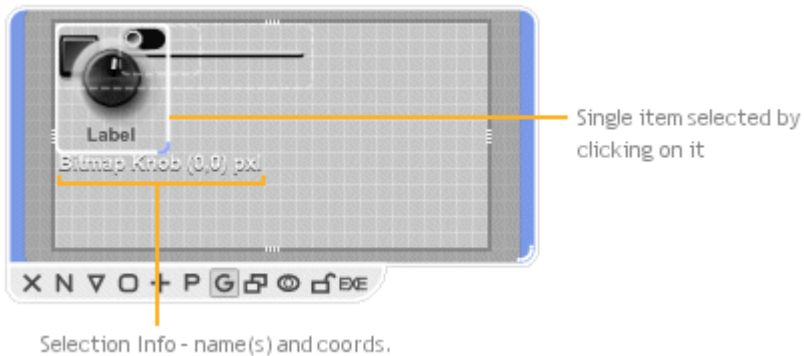
Panel is now unlocked for editing

The front panel will stay unlocked until you lock it again. However, you'll only be able to edit the panel while the module is selected. When the module becomes deselected it will operate as if the panel was locked.

## Selecting

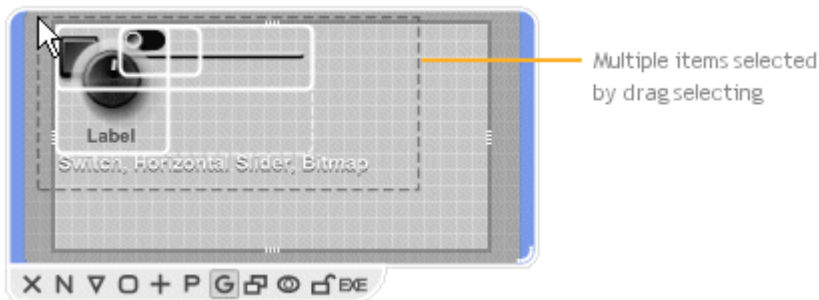
You can select items in the front panel in the usual way by clicking on them. If you hold SHIFT you can click to add or remove items from the selection. You can also click on an empty part of the front panel and drag to select all items in a rectangular area. To select all the items you can use CTRL+A.

If one item is underneath another you can hold ALT and click to alternate between selecting the top item and the one below.



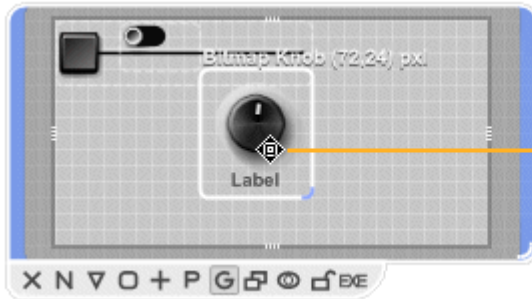
### Selection Info

Either above or below the selected items you'll see some text showing information about the selection. The first part this is the module names for the items in the selection. The second part is shows you the coordinates of the top-left corner of the selection. The coordinates are in the form "(x,y)" followed by "sqr" or "px". If sqr is showing then the coordinates are in grid squares. If px is showing then the coordinates are in pixels at the default zoom level. You can switch between grid squares and pixels by clicking on the selection info text.



## Moving

Having selected items you can then move them by dragging them around. By default the position snaps to the grid but you can prevent this by holding CTRL as you drag.



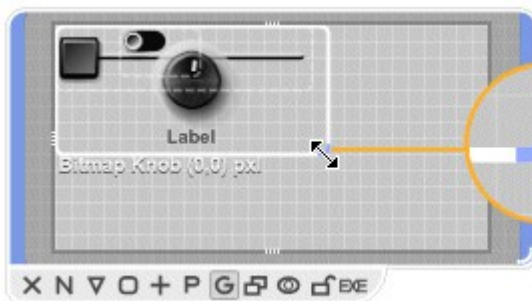
Click and drag the item to move it  
Hold CTRL to prevent snap to grid

You can also use the cursor keys to nudge a selection by one grid square at a time in any direction. If you hold CTRL while nudging you will move a distance equivalent to 1 pixel at the default zoom level.

If you're zoomed in or out then you'll move by a distance that is equivalent to 1 pixel at the current zoom level. This allows you to have ultra fine control over the placement of items on a front panel.

## Resizing

You can resize front panel items. This works in the same way as for components in a schematic. First select the item you want to resize. The bottom-right corner of the selection will show a blue resize control. Click and drag this to resize.

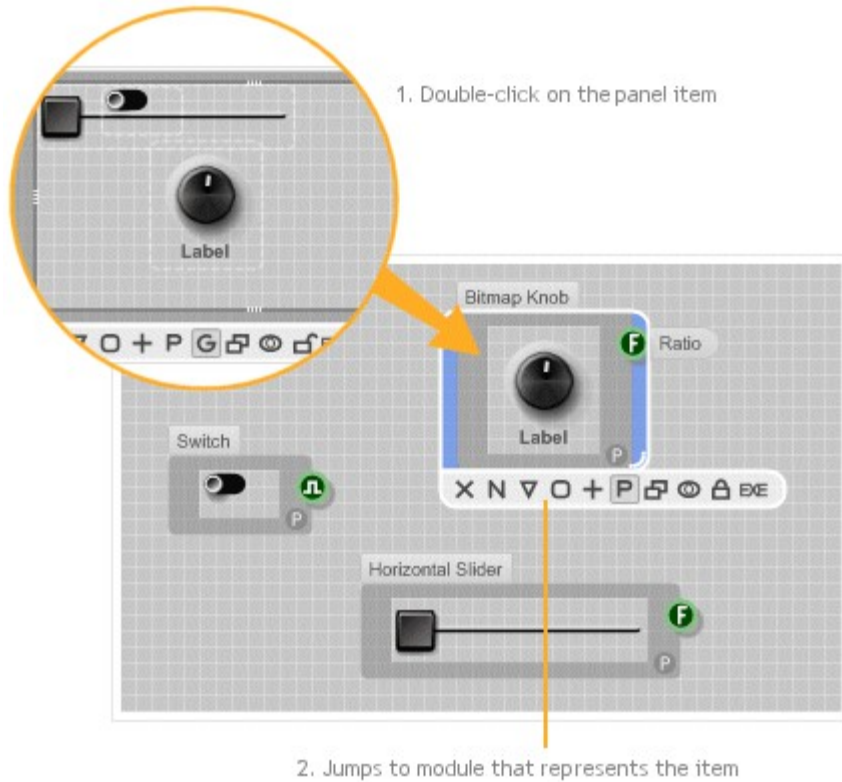


Click and drag the blue resize handle

Note that because each item in the front panel is actually the front panel of a module somewhere lower down in the hierarchy, resizing an item will resize the corresponding module in the schematic.

## Jumping

If you want to jump straight to the module that is represented by a front panel item then all you need to do is hold SHIFT and double-click on it.



## **Sub-Panel Editing**

Each front panel item is a module with its own front panel and so may contain its own front panel items. You can edit these sub panel items in place. Simply double-click on an item and the editor will mask out everything around the item and allow you access to the sub panel items below.

Double-clicking on the masked area will move you back up one level. Alternatively, you can click on an empty part of the schematic to deselect the module and when you return to edit again you'll be back at the topmost level.



## Hiding Item Boundaries

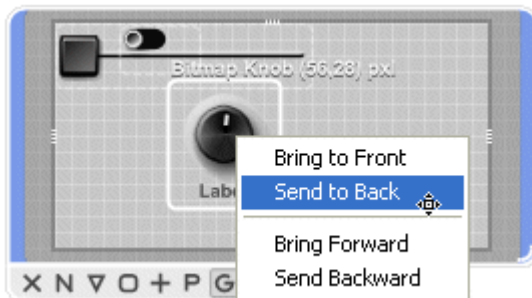
When editing a front panel, each item has its boundary marked with a rectangle. If you're editing a sub panel item then you may have some masking applied as well.

Sometimes these things can get in the way of you seeing exactly how your layout will look. To help with this you can hide all the boundaries and masking by pressing the X key. Press this key again to return things to normal.

Note that with the boundaries hidden you won't now be able to see what item (if any) is selected. However, you can easily toggle between showing and hiding boundaries by pressing the X key multiple times.

## Draw Order

If you have several items that overlap with each other then you can determine which ones appear on top of the others by using the order menu. Right-click on a selected item (or items) and a pop-up menu will appear.

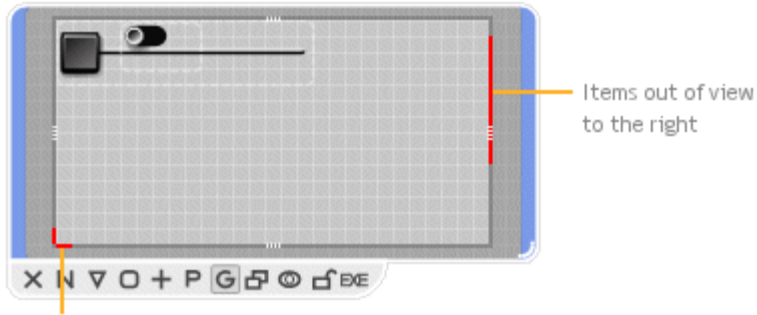


You can then choose whether to bring a selection forward in the draw order or send it backwards. You can send the item straight to the back or bring it to the front.

## Out of View Items

It is possible for panel items to move out of view. This can happen when resizing the panel or when copying and pasting modules. When this occurs you'll see red markers on the edge of the panel's exterior. These also indicate the location of the items.

CHAPTER 4



Items out of view to the bottom-left

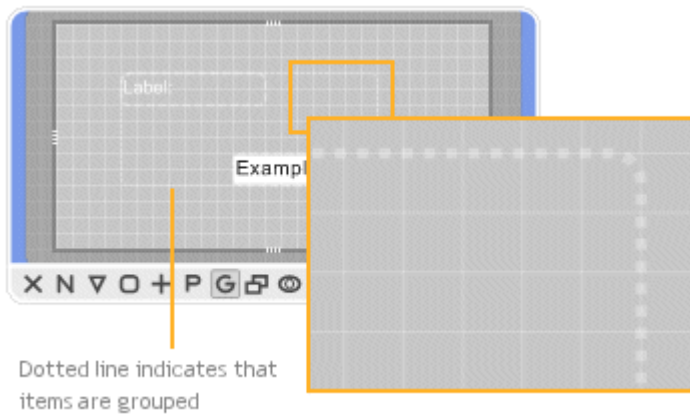
Items out of view to the right

To get the items back into view simply right-click on the module and select Bring Panel Items Into View. This option is also on the Schematic menu on the menu bar.

## Grouped Items

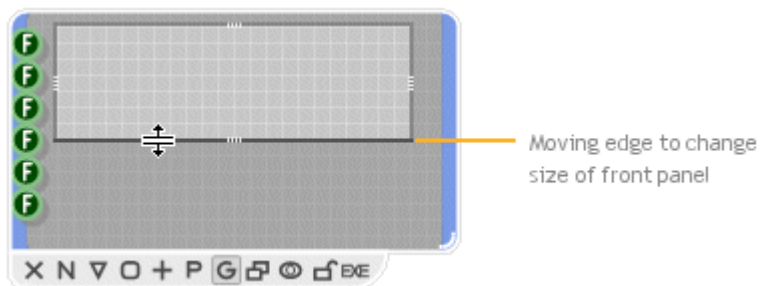
Sometimes front panel items are grouped together. This is determined by the way that the modules are constructed. We'll talk about how this comes about later on so for now we'll just talk about what happens when it occurs.

Grouped items are indicated by a thin dotted line around the items. The only way that grouped items behave differently from those that are not grouped is that they are constrained to appear in the same draw order relative to each other. For example, if you send one item in the group backwards, all the other items in the group will move backwards too.



## Client Area

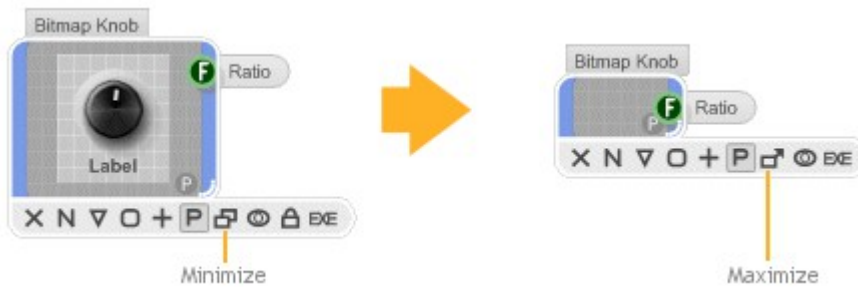
On occasions you may need to change the size of a front panel without changing the size of the module. This can happen when the number of module inputs or outputs imposes a minimum size on the module but you want the height of front panel to be smaller.



When the front panel is unlocked you'll see a thick dark grey border around the panel edge. This defines the client area. You can drag the borders around to change the size of the client area. When an edge is moved from its standard position it changes to a darker grey colour. Note that when this happens the edge becomes detached from the module boundary and will not resize when the module resizes unless it is forced to do so. You can reattach an edge to the module by dragging it back to its original position

## Hiding the Front Panel

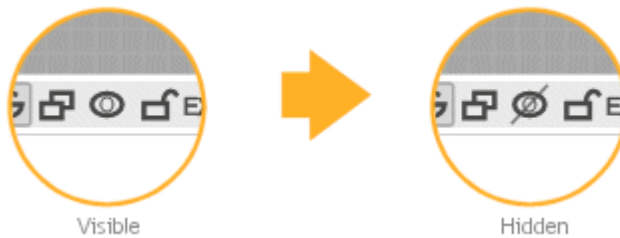
Sometimes you want a module to have a front panel but you don't want to display it. This may be because it is too big and you want your schematic to be compact. You can hide the front panel by minimizing the module. First select the module then click on the Toggle Minimize button on the front panel (or right-click on the module and select Minimize Front Panel).



The module will appear as it would be if no front panel were present. You can maximize the module again to show the front panel if you wish by performing the same action.

## Visibility in Parent Module Panels

There are some circumstances under which you don't want a module's front panel to appear in parent module front panels. For example, you may be using a knob to control a variable for experimentation purposes or you may want to hide a background image to stop it from getting in the way while you arrange other panel items.



You can hide a module by selecting the module and clicking on the eye button in the action panel (or you can right-click on the module and select Show In Parent Panel). The eye button will show with a strikethrough to indicate that the module will not be shown in parent module front panels.

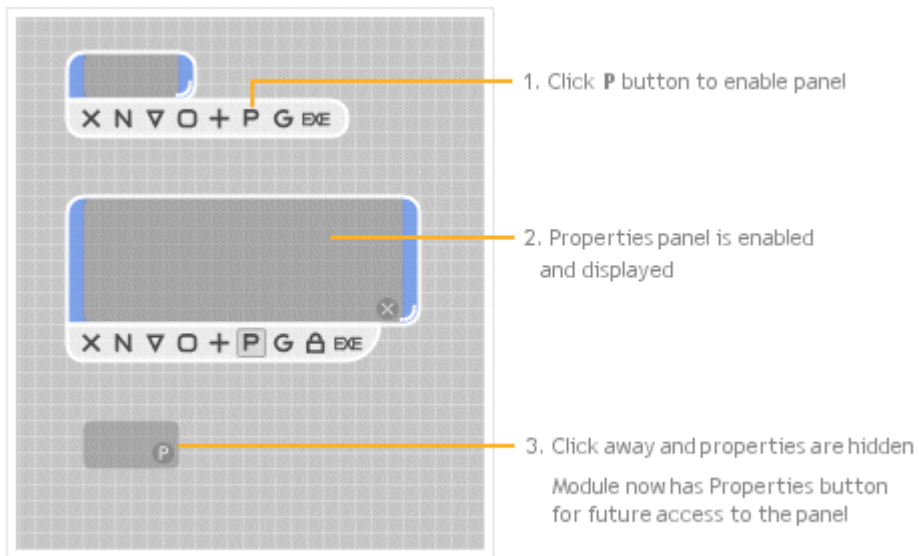
The Show In Parent Panel operation will work on multiple selections too.

# Properties

We've already seen how to access and change the properties of a module but how do they get there in the first place? In this section you'll learn how to add properties to your own modules.

## Enabling the Properties Panel

To enable the properties panel all you need to do is select the module and click the **P** button on the action panel. You can also right-click on the module and select Enable Properties.

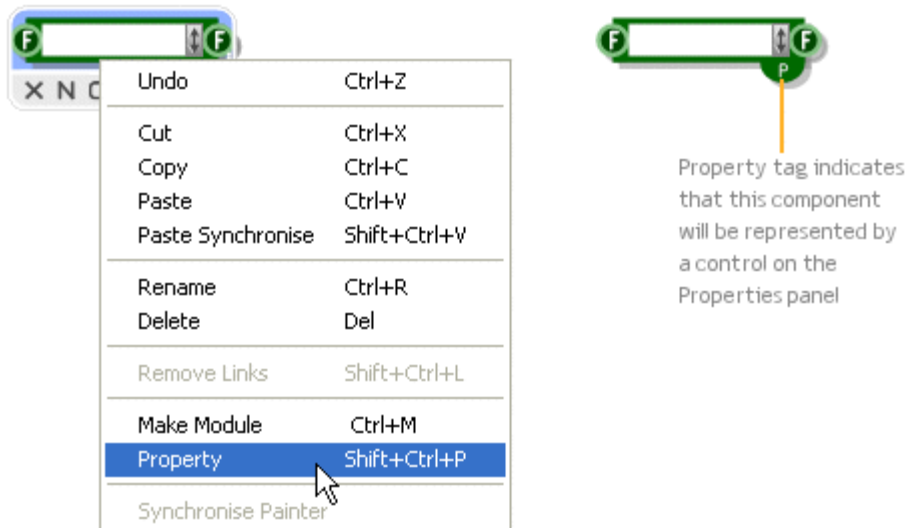


## Adding Property Items

The items on the property panel each map onto a particular component somewhere within the module or it's sub-modules. There are six different types of components that can be enabled for display on the properties panel. They are: Int, Float, String, Boolean, Index Selector and Trigger Button.



To enable one of these components for display on the properties, right-click on the component and select Property. You'll see a small tag with a letter P in the middle appear in the bottom-right corner of the component. This indicates that this component is a Property and will be represented by a control on the Properties panel.



If you add a label to the component this will be used on the front panel to identify the component. If you now go back to the properties panel you'll see the property control for the component.



## Control Types

Each of the five property enabled component types has a control that represents it on the Properties panel. There are just three different types of control.

Float, Int and String components are represented by Edit controls. These have a label which takes it's text from the label of the component.



Boolean components are shown as a check box. Again the label for the check box uses the text from the label of the Boolean component

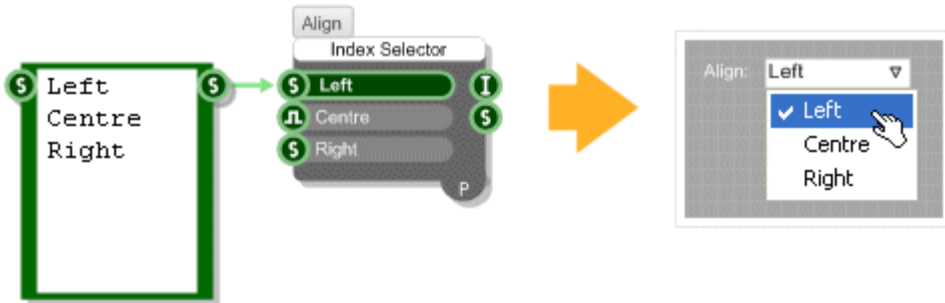




The Trigger Button component is represented by a button. The button text is taken from the label assigned to the Trigger Button component.

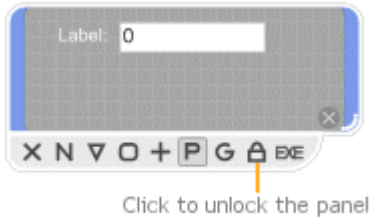


The Index Selector component is used for selecting from a list of options so this is represented as a drop-down list. The component label once again supplies the text for the control label.



## Editing the Properties Panel

To edit the positions of items on the panel just unlock the panel, exactly the same as you would for editing a front panel.



Editing is then exactly the same as for the front panel. You can select items, drag and nudge items around or resize them. See the section on the module front panel for more details.

## Resizing

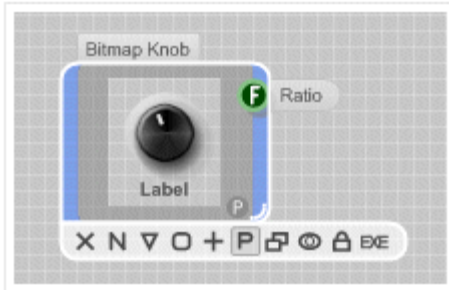
The size of the properties panel can be adjusted independently of the size of the module. When the panel is open any resizing of the module will only apply to the properties panel. When you click away or close the properties panel the module will return to its original size.

## Customizing

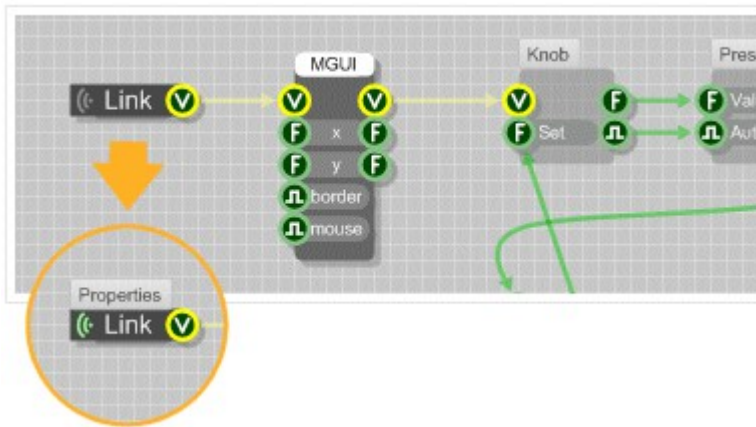
You are not just restricted to the three control types described earlier. Because the panel works just like an extra front panel you can in fact add any kind of controls or graphics you like. All you need to do is connect your custom GUI to a Wireless Input with the label 'Properties'.

The example below shows how to show a knob on the properties panel.

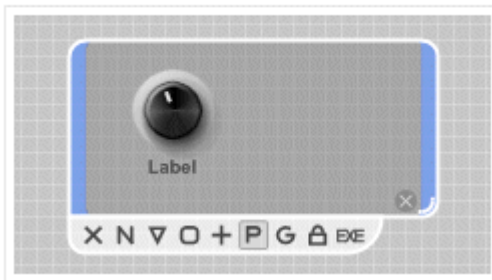
1. Drop knob inside your module



2. Go inside knob module and rename Wireless Input



3. Go back up and knob is now on properties panel



# Synchronising

You'll often find that you have many copies of the same module scattered around your schematic. This is one of the great benefits of modules – you create one and then you can use it in many other places. However, what happens when you want to change the behaviour of the module?

Well, you certainly don't want to have to change all your copies one by one. You could change just one of them and then copy and paste it again but that's not great either. It would be much better if you could just transmit the changes to all your modules as you make them. That's exactly what Module Synchronising or Module Syncing is designed to do.

When one or more modules are synchronised any changes you make to one module are instantly made to all the others, it's that simple.

## Paste Synchronise

You can put copies of modules in sync as you make them. Instead of pasting a copy of the modules, choose Paste Synchronise from the Edit menu or use SHIFT+CTRL+V.



When you Paste Synchronise for the first time you'll notice that both the original module and the copy now have the module syncing symbol. The pasted copy will be selected. When a synced module is selected you will also see the sync count. This indicates how many other modules are synced with this one. The count is of the form xN where the x symbol represents multiplication and N is a number.


## Synchronise All

If you didn't think in advance that you wanted copies of a module to be in sync then don't worry. There's an easy way to put them all in sync after they have been created. Just right-click on one of the modules and select Synchronise All. All the modules that match the module you clicked will be put into sync, regardless of where they reside in your schematic.

Note that if any of the original copies have been altered in any way then they cannot be brought into sync.

## Synchronise Painter

For more selective syncing we have the Synchronise Painter feature. This is a mode that you put the software into which allows you to click on the modules that you want to put in sync.

Right-click on the module that you want to synchronise with and select Synchronise Painter. The cursor will change to the synchronise painter  cursor. To make another module sync with this one all you need to do is click on it.

If the module is identical and so can be synced the module border will flash blue to indicate that the operation succeeded.



You can navigate through your schematic while the Synchronise Painter is switched on. You can use the Navigator or the module action panels. You can also hold CTRL to temporarily disable painting while you double-click on a module to go inside it.

To exit Synchronise Painter mode you can right-click on your schematic and uncheck the Synchronise Painter option. Alternatively, you can hit the ESC key. If you try and perform any other editing operation whilst the Synchronise Painter is on it will immediately switch off and you'll return to standard editing mode.

## Removing Synchronisation

You will on occasions need to be able to stop a set of modules from syncing with each other. It's actually a good idea to remove syncing when you know that you don't need to make any more changes.

There are three ways to do this. You can use the Synchronise Painter as described above. For a really quick way to remove syncing you can right-click on a synced module and select Un-synchronise All. This will remove syncing from all of the modules that are synced with the one you clicked.

You can also just remove syncing for a particular module on it's own by right-clicking on it and selecting Un-synchronise.

## Finding Synchronised Modules

If you want to locate modules that are synced with a particular module then simply select the module and press the TAB key.

The software will jump to the next module that is in sync. Continue pressing tab to cycle through all the modules that are synchronised. Eventually you will return back to the module that you started with.

You can also hold SHIFT and press TAB to cycle through the synced modules in the opposite order.

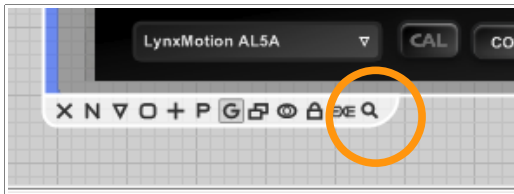
# Focus Mode

If you're building an application to export sometimes you may want to preview it or even just use it without all the editing interface getting in the way. You can do this using a feature called Focus Mode.

Focus Mode applies to a particular module. It collapses the application window around a module and allows you to have it as the main focus – as if it was the application.

## Focus on a Module

To put a module into Focus Mode, select the module then click the focus mode button on the action panel (looks like a magnifying glass). Alternately right-click on the module and select Focus Mode.



To exit Focus Mode click the close button, press the ESCAPE key.

## Focus on the previous Module

You may want to preview your application or module when you're in some other part of the schematic without having to navigate back to find it each time. You can do this by selecting Focus Mode from the View menu or by right-clicking on an empty part of schematic and selecting Focus Mode.

The last focused module will be selected in this case regardless of any other module that may be selected in the schematic at the time.

For quick switching Focus Mode in this way you can use the keyboard shortcut SHIFT + ESCAPE.

# 5 Data Types & Signal Flow

CONNECTOR TYPES EXPLAINED



FlowStone supports over 30 different types of data. Each type has its own connector with a unique symbol for easy recognition. Although there are many different types of data they all fall into one of three different categories: Stream, Triggered or Event.

**Stream** data covers all digital audio and control signals. These signals are fluctuating at sampling rate and so any components which process them will also perform calculations at sampling rate.

**Triggered** data works in a completely different way. Where stream data is continuously flowing, triggered data only flows in response to some event. Usually the event is a user interaction or data arriving from some external source like an interface card or a timer.

**Event** data is similar to triggered data in that it flows in response to some event. However, the way in which it flows is quite different.

Note that in the vast majority of cases data flows from left to right, by this we mean from output connector to input connector. This is true for all Stream and all Event data but there are a few exceptions with some of the triggered data types.

# Stream Data

If you are interested in audio applications or processing data at high data rates then you'll need to work with stream data. Streams deliver high data rate digital signals which you can then process in the software. This is know as Digital Signal Processing (or DSP). If you're not familiar with DSP then the next section will give you a quick overview.

## Digital Signal Processing in a Nutshell

What is DSP? Well let's start with a **signal** which we will define as the continuous value of a some parameter over time. This could be anything from the temperature of a room to audio taken from some listening device.

If the signal is passed into your computer from an external source like a microphone then it needs to be converted from an analogue signal to a digital one. This is done by measuring the magnitude of the signal repeatedly over discrete time intervals. Each measurement is called a sample and the result is a stream of numbers which is the **digital** signal.

The process is called sampling and the rate at which you measure is called the sampling rate. The most common rate used for audio signals for example is 44100 samples per second, often represented as a frequency 44.1 KHz but higher rates are also used to get higher quality signals.

The **processing** part of DSP involves taking the stream of numbers that represent the digital signal and converting them into another stream of numbers by applying some combination of mathematical transformations. This is what FlowStone does using sections of Stream data components that you connect together.

The output signal can then be converted back to analogue and transmitted on or listened to as sound through headphones or speakers.

A key feature of the software is that it can process these samples individually at sampling rate. Many applications are not capable of this and have to process a collection of samples together in one frame in order to keep cpu usage at acceptable levels. This restriction limits the processing possibilities as there is often a requirement to process a given sample based on the sample that preceded it. This situation is called feedback and single sample feedback is a capability of FlowStone that sets it apart from other software.

There are two main stream data types: **Poly** and **Mono**. Poly is only used for audio applications where sound signals are generated from MIDI notes. If you're not generating audio in this way then you can ignore Poly completely.

## Mono

Mono carries a single signal at sampling rate. A set of mono components connected together (to form what we call a Mono section) always has data flowing through it. Once connected on the right to a Direct Sound Out or ASIO Out component a Mono section will run constantly even if there is a zero level signal passing through.



**Mono connector** - a single channel of fast moving stream of floating point data that fluctuates at sampling rate. Provided it is connected to a sound output component like Direct Sound Out it is always active.

## Poly

Poly connectors can carry many digital signals at one time. A Poly section only uses processing when there are signals passing through it. Poly is only used for audio applications and is generated in response to MIDI notes.

The number of signals is determined by the number of notes being played. If there are no signals then there is no cpu hit. However when there are one or more notes playing you'll get proportional usage of cpu for each signal generated (although because of the way FlowStone uses SSE, you only get an increase on every 4<sup>th</sup> note).



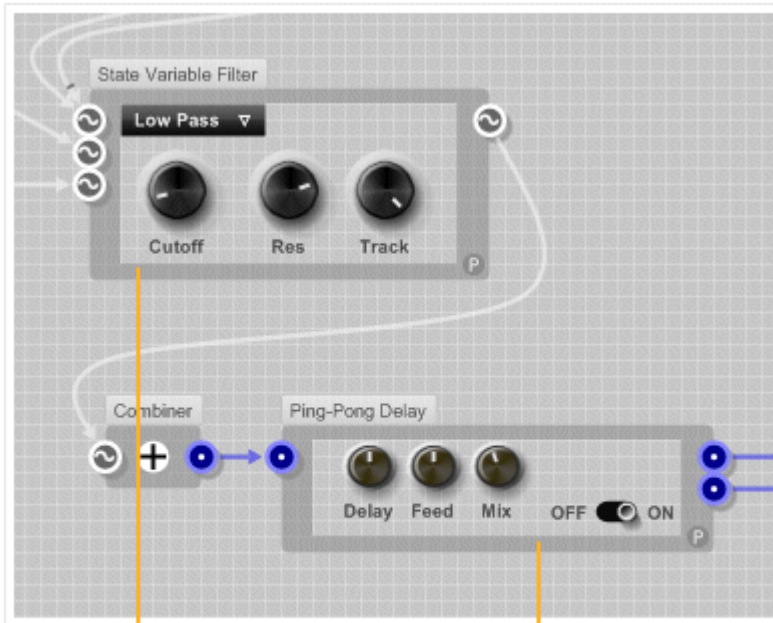
**Poly connector** - represents multi-channel audio signals. Each channel is an independent fast moving stream of floating point data that fluctuates at sampling rate. The data is multi-channel because there is one channel for each note that you play.

## When to Use Poly or Mono

You would use a combination of Poly components (a Poly section) to model the polyphonic part of a synth. With a Poly section you can generate separate audio signals for each voice or note that you play. You can then merge the Poly signals to Mono and have a Mono section where you would apply effects to the combined signal as a whole.

CHAPTER 5

Poly and Mono connectors allow you to easily see what kind of data is flowing through different parts of your schematic. This is important because when adding to your schematic you need to know whether the processing you introduce will be applied cumulatively for each voice (Poly) or constantly for one signal (Mono).



Filter is in Poly section so applies to individual notes but only when they are playing

Ping-Pong Delay is in Mono section and applies to all notes at once, processing constantly

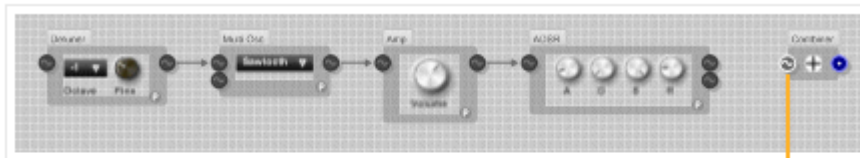
## Stream Connectors

You can create modules that can be used in a Poly or a Mono section by using Stream connectors. These look like dark grey versions of the poly connector.

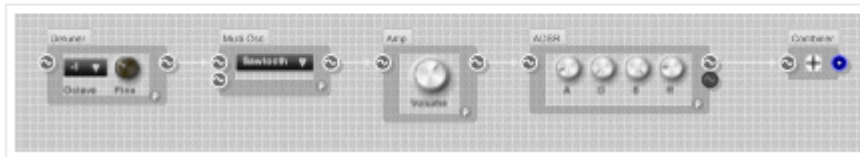


When a Stream output connector is linked to a Poly or Mono connector it instantly picks up that type. The type change then flows from left to right back through any other stream connectors changing them to the new type accordingly.

1. Stream section (indicated by dark grey links and connectors)



2. Link to Poly connector

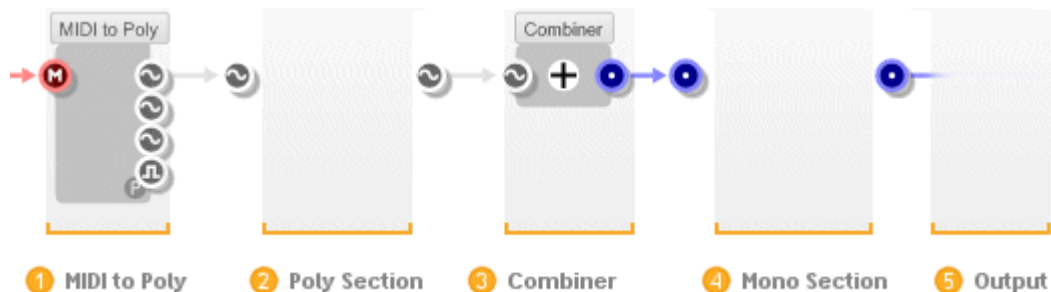


3. Turns to Poly section

## Poly and Mono Sections In Audio Applications

To get any data to flow through a poly or mono section you have to connect it up correctly. In most cases you'll want to use MIDI input from a keyboard or other source to generate notes. You then want each note to make a sound that you can hear.

The most common setup would be as shown below:



1. The MIDI to Poly module converts incoming MIDI note data into Poly signals
  2. The Poly section processes the signals
  3. The Combiner module then combines the independent Poly signals into one Mono signal
  4. The Mono section applies processing to the result
  5. The final signal is sent to your Sound Card via a Direct Sound Out or ASIO Out component within FlowStone or via the host audio setup if used within a plugin
- NOTE:** If you don't have Direct Sound Out or ASIO Out component no audio processing will be performed.

A complete, usable poly section always starts with a MIDI to Poly module and ends with a Combiner module. The only exception is if you're using a Signal Analyser to look at the output from a Poly section.

You can have a Mono section without a Poly Section. This is what you'd have if you were creating an effect that performs some DSP on an incoming signal. However, if you want to hear anything you have to connect a Mono section to an Output device via a Direct Sound Out or ASIO Out component.

## Boolean Connectors

There are Boolean equivalents of the Poly, Mono and Stream data types. They are only seen on a few components, most notably the Poly and Mono comparison components.

The MIDI to Poly module has a Poly Boolean output which shifts from false to true when a note is played and can therefore be used as a gate signal.



Poly Boolean - multi-channel mask, one mask for each note playing



Mono Boolean - single channel mask, constantly running



Stream Boolean

## Poly Int

The Poly Int data type is the integer equivalent of the Poly type. It is used only in the Graph to Poly to allow sample rate indexing of Float Arrays.



PolyInt - multi channel integer, one channel for each note playing

## SSE

If a cpu supports SSE then it has a set of instructions built in that allow mathematical operations to be performed on multiple sets of data at the same time. FlowStone makes full use of this when processing stream data. The result is that you can effectively process up to 4 channels at the same time for the same cpu cost as just one.

## Mono 4

If you really want to get the best performance out of a Mono section then you need to consider using the Mono4 data type. It's not always possible to use this but when you can it makes a huge difference to performance.

Mono sections only have one sound process. Because FlowStone uses SSE this process could be doing four times the work for the same cpu cost. To take advantage of this we have the Pack and Unpack components. These allow you to literally pack four mono channels into one Mono4 stream.



Mono4 - four Mono signals processed as one

## Performance

To help you gauge the effect of changes to the stream parts of your schematic we have a cpu meter. This can be found on the right side of the status bar at the bottom of the application window. The cpu meter is very basic. It only measures cpu performance of the stream sections, GUI performance is not measured.





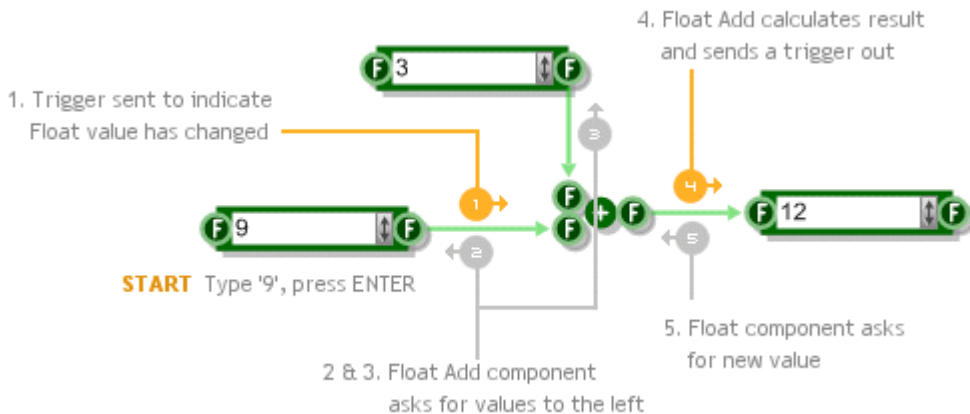
# Triggered Data

## How it Works

Triggered data only flows through a schematic when an event occurs that 'triggers' a change. Until this happens sections of triggered data remain in a steady state.

When an event does occur a message called a Trigger is sent out through output connectors on one or more components. The trigger flows through the links until it reaches another component. That component then assesses whether it's state would be affected. Any necessary recalculations are then performed and a trigger is sent out through any output connectors on that component. The process continues creating a waterfall effect through the schematic.

The example below shows a very simple example of two numbers being added together. When you type in a Float component this results in one of the events that we've been talking about and a trigger flurry begins.



Note that during the recalculation part a component may ask the components that provide values to it's inputs for their latest values. This in itself can cause a flurry of messages to the left.

## What it's Used For

Triggered data has two main functions. First it provides a way of performing calculations that don't need to be done at sampling rate. Second it allows you to build extremely detailed and complex user interface elements which only need to respond when you interact with them.






As a general rule the fast DSP is handled by the Stream data and the slower DSP and user interface is handled by the Triggered data.

## Triggered Data Types

The various triggered data types can be grouped into several categories based upon their purpose.




### Primary Types

These are the most commonly used types. They represent simple data like numbers and text.

-  Float - a 32 bit floating point number
-  Int - a 32 bit signed integer in the range -2147483648 to 2147483648
-  String - an alphanumeric string of characters of unlimited length
-  Boolean - one of two values: true or false
-  Trigger - not really a data type (there is no data) but used to pass trigger messages










### Array Types

These types represent resizable arrays of some of the primary types. An array is just an ordered list of items of the same type. Array types have a 'four leaf' outline pattern instead of a circle.

-  Float Array - array of 32 bit floating point numbers
-  Int Array - array of 32 bit signed integer in the range -2147483648 to 2147483648
-  String Array - array of characters strings



## GUI Data Types

These are used only for low-level GUI editing so you won't need to know about them until you come to use the GUI components.

-  View - transports all drawing and mouse information
-  Area - an are defined by coordinates of top-left corner, a width and a height
-  Mouse - mouse events (left button up/down, mouse move etc.)
-  Colour - in argb format ('a' is the transparency level)
-  Pen - for drawing lines - defined by colour, thickness and style
-  Font - font information comprising typeface, size and style
-  String Format - alignment information for drawing text
- Bitmap - 32bit image
-  Point Array - an array of points (floating point pairs)
-  Bitmap Array - an array of bitmaps


## Memory Types

There are two types that represent buffers of data stored in memory. They are most frequently used for storing wave file data for samples.

-  Mem - a contiguous section of data in memory
-  Mem Array - an array of memory sections of unlimited length






## Ruby Types

In Ruby all data is considered to be an object. Numbers are objects, strings are objects, arrays are objects. Outside of Ruby code objects all have the same type: VALUE so when you pass Ruby objects between Ruby components you pass them through the Ruby Value connector.

-  Value - can refer to any Ruby object

### Special Types

There are a handful of data types that stand on their own.

-  MIDI - all standard MIDI messages
-  Voice - currently only used within the MIDI to Poly module but may be extended in future
-  Bus - a user defined set of data types that travel together through the same connector
-  Preset - carries information about preset parameter and program changes for plugins
-  Template - not a data type, instead this takes the type of whatever you connect it to

# Event Data

## How it Works

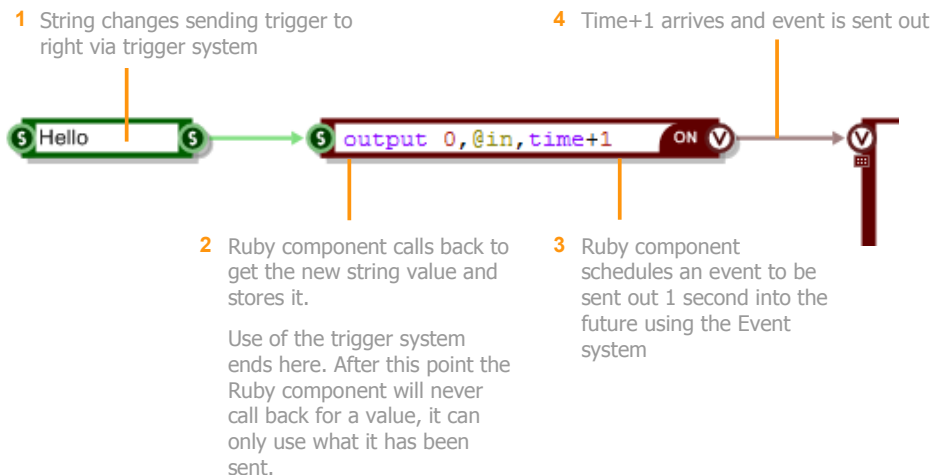
Like triggered data, Event data only flows through a schematic when something occurs that causes a change. Until this happens sections of event data remain in a steady state.

When an event does occur a message called an Event is sent out through output connectors on one or more components. The Event flows through the links until it reaches another component.

This all sounds very similar to the Trigger system. However, there are two important differences. First the Event carries data with it. The Trigger system only signals that a change has occurred, components then call back (to the left) to calculate their values. Events carry data with them. Once an event arrives at a component there is no calling back.

The second difference is that Events are scheduled. When an event is first created it is given a time stamp and will only be executed when that time is reached. This means that you can specify exactly when you want an event to occur. This timing integrates fully with the stream data so that you can schedule events to occur at a precise sample.

Currently the Event data only applies to the Ruby component. See the Ruby Component section for more information.



## Event Data Types

Outside of the Ruby Component only one data type uses the Event system, Ruby Value.

In Ruby all data is considered to be an object. Numbers are objects, strings are objects, arrays are objects. Outside of Ruby code, objects all have the same type: VALUE so when you pass Ruby objects between Ruby components you pass them through the Ruby Value connector.

We'll discuss this in more detail in the Ruby Component section of this guide.



Value - can refer to any Ruby object

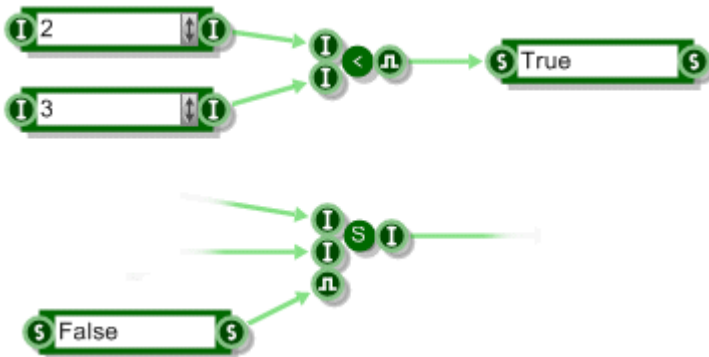
# Converting Between Data Types

In most cases you will create links between connectors of the same type. However, FlowStone supports several automatic and intuitive conversions between data types which you'll find extremely handy.

In most cases you will create links between connectors of the same type. However, FlowStone supports several automatic and intuitive conversions between data types which you'll find extremely handy.

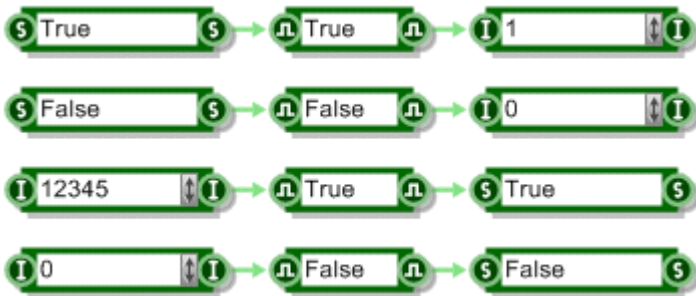
## String <> Boolean

You can get a boolean value from a string and also convert from a boolean to a string:



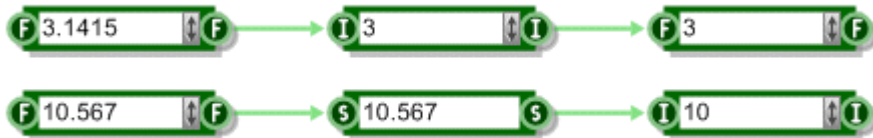
## Int <> Boolean

Boolean and Int are also interchangeable. Zero represents a false value, any other value is considered to be true.



## String <> Float <> Int

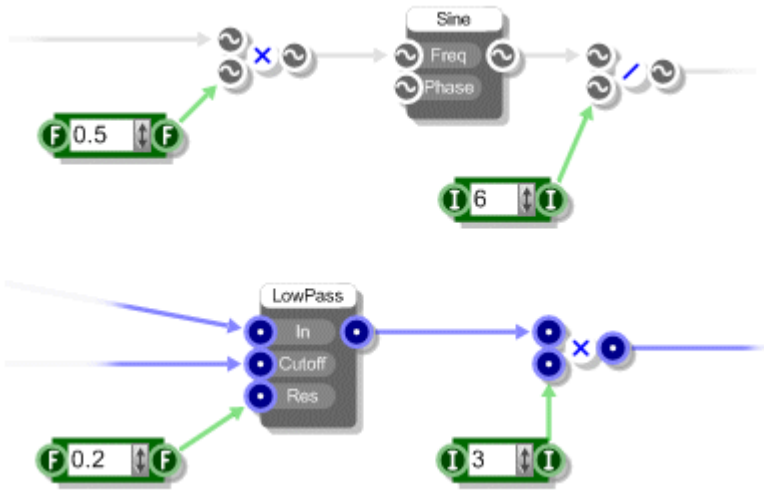
You can convert back and forward between strings, floats and ints. Rounding will obviously occur when converting from Float to Int (the decimal part is just ignored).





## Int/Float > Poly/Mono

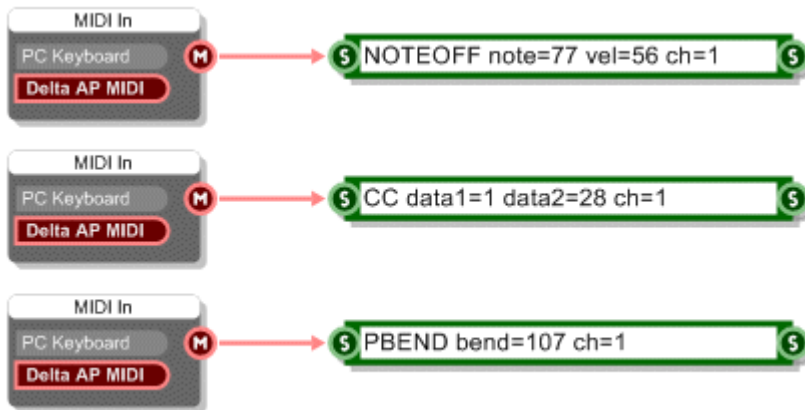
A Float or Int can be connected to Poly or Mono inputs. These act like signals that maintain a constant level. You can't however connect a Poly or Mono to a Float or Int. In fact Poly and Mono connectors can only be connected to themselves.



## MIDI <> String

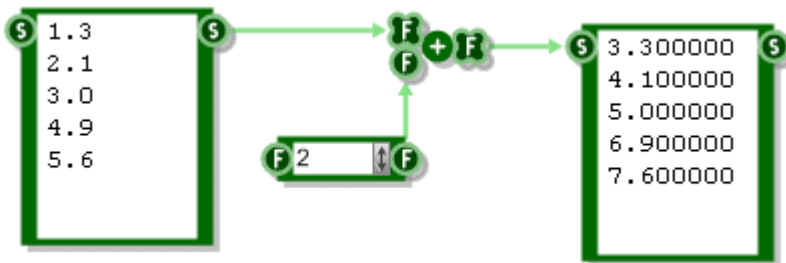
This is not so much a conversion as a handy way to check what MIDI data is coming from a MIDI output. Attach a String data component and note on/off, control change and pitch bend messages etc. are shown together with the various parameters that define them.

As this isn't a conversion as such you need to 'force' MIDI to String links by holding SHIFT + CTRL when linking.



## Float Array <> String

You can easily convert between a string and a float array.



## String Shortcuts

The String data component can be used as a shortcut for defining various GUI data types. Colours, Areas, Pens and more can all be defined in a single text string.

### Area

To create an area use the format "x,y,w,h" where x and y give the top-left corner of the area and w and h give the width and height of the area. All dimensions are in grid squares of course.



### Colour

There are two ways to specify a colour using a data String. You can use one of the 14 predefined colours which are as follows:

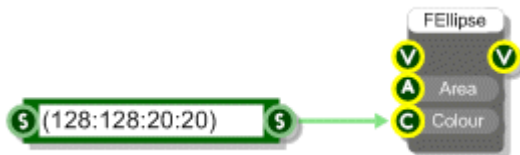
Black, White, Red, Green, Blue, Yellow, Grey

tBlack, tWhite, tRed, tGreen, tBlue, tYellow, tGrey

(the 't' colours are partially transparent)

Use the colour by name in the String component. Note that the colour names are not sensitive to case.

The second way to specify a colour is by ARGB. Use the format "(a:r:g:b)" where **a** is the transparency, **r** is the red component, **g** is the green component and **b** is the blue component. All values are integers in the range zero to 255.



### Pen

A pen has three attributes: colour, thickness and style. Using a string component you can specify a pen by providing these attributes in the format "colour,thick,style".

The colour part is exactly the same as for specifying a colour with a string (see above). The thickness is a floating point number in grid squares. The style can be any one of the following strings:

`solid, dash, dot, dashdot, dashdotdot`

You can leave the style parameter out and a solid style will be assumed.



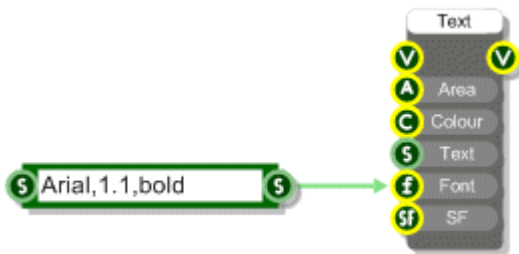
### Font

Fonts can be generated by Strings using the format "typeface,size,style". Typeface is the name of the font face e.g. Arial or Tahoma. Size is the height of the text in grid squares (it is not a point size).

Style can be any combination of the following strings (in any order):

`normal, bold, italic, underline, strike`

Some examples: bolditalic, underlineboldstrike, italicunderline. You can leave the style parameter out and a regular style will be assumed.



## StringFormat

StringFormats are specified using the format "style,horizalign,vertalign". The style can be any combination of the following strings (in any order):

```
normal, righttoleft, nowrap, vertical
```

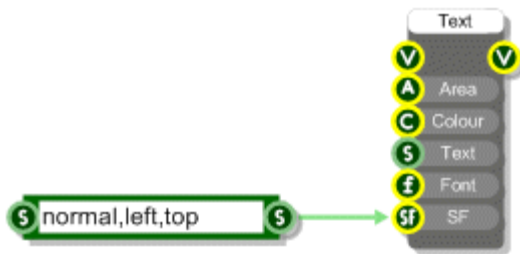
Some examples: nowrapvertical, nowraprighttoleft. You can also use 0 to indicate no style options apply.

For horizontal alignment you can use:

```
left, center or right
```

For vertical alignment you can use

```
top, center or bottom
```



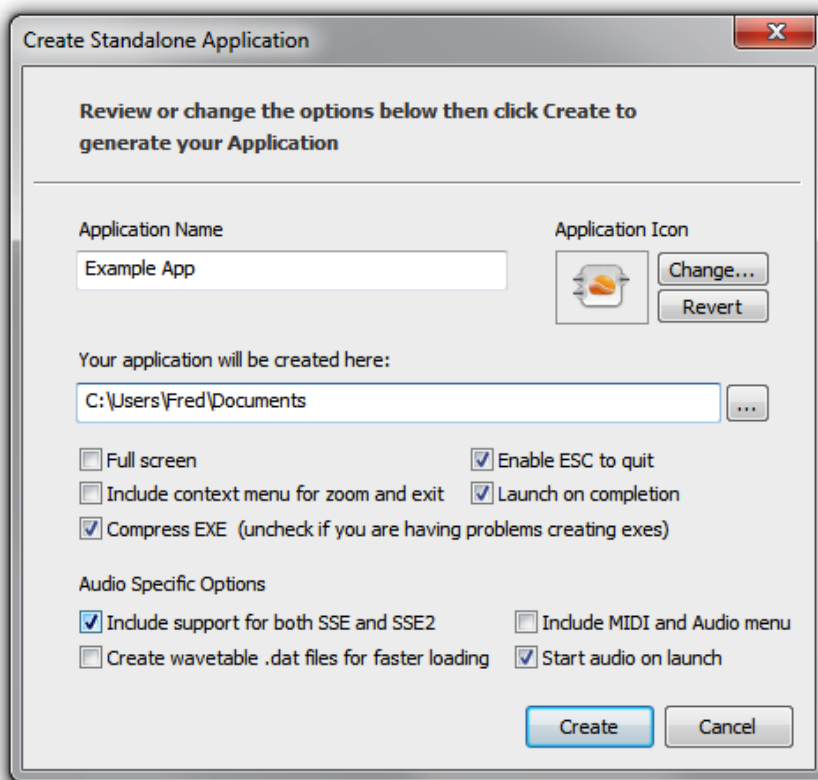
# 6 Exporting

CREATING STANDALONE APPLICATIONS AND PLUGINS

## Creating Standalone Applications

FlowStone can be used to create complete applications that will run on their own. First click the EXE button on the action panel of the module you want to export (or right-click on the module and select Create Standalone from the pop-up menu).

The Create Standalone Application dialog will appear.



### Application Name

The application name is the name of the .exe file that will be generated. By default the software will use the module label. If there is no label the dialog will show the last name that you used.

### **Application Icon**

You can choose your own icon for your application. Click Change and locate the icon you want using the resulting dialog box. Icons need to be stored in .ico files before they can be used by FlowStone. If you decide you just want to use the default icon click the Use Default button. A preview of the icon that the software will use is shown on the dialog box.

### **Your application will be created here**

Applications are created in a particular folder and you have the option to change this.

### **Full Screen**

Check this box if you want your application to launch and fill the whole screen. This is the ideal option for applications intended for embedded systems.

### **Include context menu for zoom and exit**

This option determines whether the default right click menu is present in your exported exe. This menu allows users of the exe to change zoom level, move to full screen and exit the application..

### **Enable ESC to Quit**

This option enables the Escape key as a means of exiting your generated exe. It's useful for applications that run in full screen where you don't want to have a Quit button or menu on the GUI.

### **Launch on Completion**

You can choose whether to launch the newly generated application on completion by checking this box.

### **Compress EXE**

Exported exes are automatically compressed to decrease file size. On a very small number of systems this has been found to interfere with the export process causing it not to work. If you are experiencing problems of this nature then you can disable compression.

### **Include Support for SSE and SSE2**

The PC on which you generate your exe may support SSE2 or it may just support SSE. When you export an exe the software saves code that is optimised for the SSE capabilities of your own PC. If you then give the exe to someone whose PC has different SSE capabilities from you the software needs to make adjustments on loading so that the plugin will work.



These adjustments can slow down the loading process. To avoid this you can choose to include support for all SSE setups at export time. This way there is no reduction in loading speed when exes are used on PCs with different SSE capabilities.

#### **Create wavetable .dat files for faster loading**

The Sawtooth and Sine components use wavetables. These are generated on startup but you can choose to save them to a local .dat file in order to speed up the exe loading time.

#### **Include MIDI and Audio Menu**

For audio applications FlowStone can create exes with a default menu that allows users to select MIDI input and Audio output. You can choose not to have this menu and either provide access to these options through your own GUI or not provide them at all.

#### **Launch on Completion**

You can choose whether to launch the newly generated application on completion by checking this box.

#### **Start Audio on Launch**

If this option is selected the software will attempt to open the default audio driver when the exe launches.

Click Create and within a few seconds your application .exe will be created in the target location that you specified. If you checked the Launch On Completion box the application window will appear.

## **Library Dependencies**

On most occasions your exported exe can be distributed on its own. However, some components in FlowStone are reliant on external libraries. If your project uses these components and you then export to a standalone you will need to distribute the supporting libraries together with your executable.

You will be informed of any dependency when you export.

We provide pre-packaged installers for the libraries on our web site. You are free to distribute these in the form they are provided. For more information see our web site page:

<http://www.dsrobotics.com/libraries.html>

## Creating Plugins

FlowStone can export fully independent VST plugins. VST is an industry standard format created by Steinberg. A VST plugin is a virtual digital effect processor or instrument that can be used in VST compatible applications for creating music.



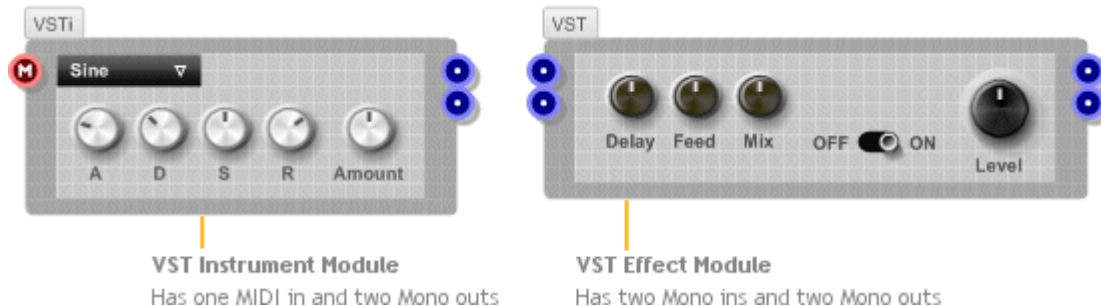
The mechanism for creating a plugin in FlowStone is very simple. All you need is a module with an acceptable combination of MIDI and/or Mono inputs and outputs. The module front panel (if you provide one) will provide the user interface for your plugin. Inside your module you need to provide the required inputs and outputs then everything that goes in between is up to you.

### Inputs and Outputs

For a VST instrument your module will need one MIDI input and two mono outputs. This allows the instrument to send stereo audio data based on MIDI signals sent from a host.

VST effects can vary. You can create a MIDI effect which takes one MIDI input and one MIDI output. For a standard audio effect you need two mono inputs and two mono outputs.

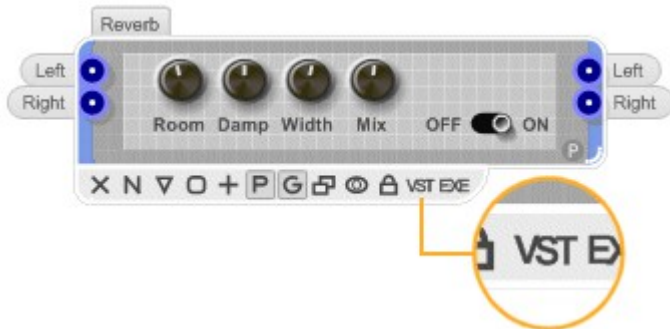
Of course you can have any number of inputs and outputs but you'll need a host that can handle the data you're requesting from or sending to it.



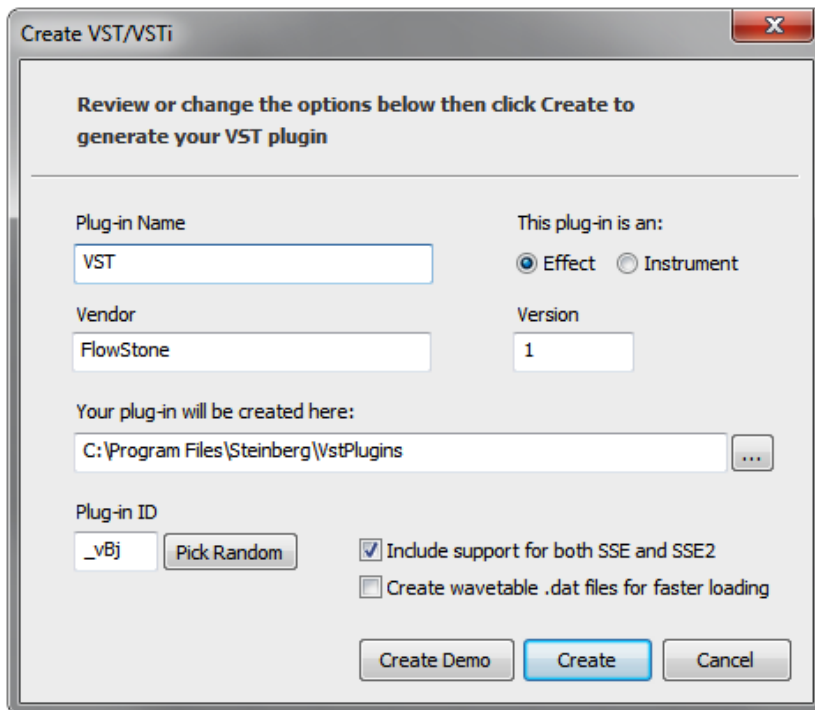
Note that the VST button, and therefore the option to create a plugin from a module, will not appear unless the combination of inputs and outputs is acceptable. You can use the VST or VSTi modules in the toolbox if you want a template module to use as a start point for your plugin.

## Create VST/VSTi Dialog

To create a VST plugin, simply click the VST button on the action panel of your plugin module or select the module and choose Create VST/VSTi from the Schematic menu.



When you click the VST button you'll be presented with the Create VST/VSTi dialog. This allows you to specify the various characteristics of your plugin.



### **Plug-in Name**

The plug-in name is the name of the dll that will be generated and the name that will be used when the plug-in is listed in your host. By default the software will use the module label. If there is no label the dialog will show the last name that you used.

### **This plug-in is an**

The plugin type will be set automatically based on the combination of inputs and outputs on your module. Basically if you have one MIDI input and two mono outputs then Instrument will be selected otherwise Effect will be selected.

### **Vendor & Version**

The vendor name is your name or your company name if you want to specify one. You can also set a version number, this must be an integer value.

### **Your plug-in will be created here**

Plugins are created in a particular folder and you have the option to change this.

### **Plug-in ID**

You can set the four character unique ID for your plugin. This is part of the VST standard. You should try to use an id that is not used by any other. However it is widely recognised that with the number of VST plugins in circulation it's almost impossible for the unique id system to work. In the majority of hosts it is not necessary to have a unique id.

### **Include Support for SSE and SSE2**

The PC on which you generate your plugin may support SSE2 or it may just support SSE. When you export a plugin the software saves code that is optimised for the SSE capabilities of your own PC. If you then give the plugin to someone whose PC has different SSE capabilities from you the software needs to make adjustments on loading so that the plugin will work.

These adjustments can slow down the loading process. To avoid this you can choose to include support for all SSE setups at export time. This way there is no reduction in loading speed when plugins are used on PCs with different SSE capabilities.

### **Create wavetable .dat files for faster loading**

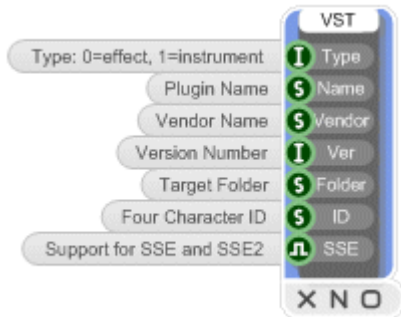
The Sawtooth and Sine components use wavetables. These are generated on startup but you can choose to save them to a local .dat file in order to speed up the plugin loading time/

Click Create and within a few seconds your plugin dll will be created in the target location that you specified.

To create the plugin click the Create button. You can also create a demo version. This will insert intermittent noise bursts into your plugins.

## Storing VST Export Preferences

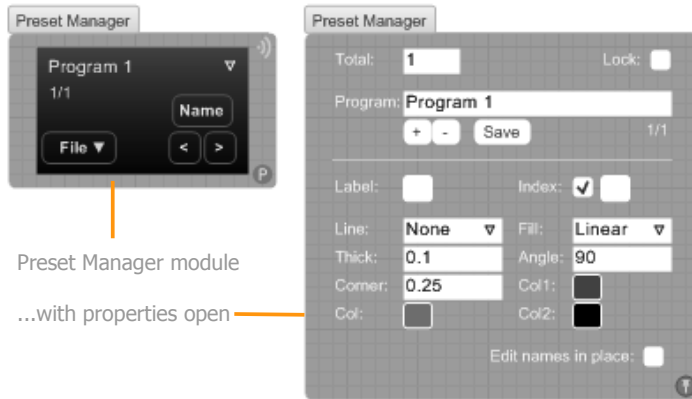
In order to preserve the settings you want to use in the Create VST/VSTi dialog you can add a VST Plugin Info component to your module. This has inputs for all the parameters in the dialog.



With nothing connected to an input the default value will be used. However, if you provide a value for any of the inputs the corresponding field in the dialog will be filled out with this value when you press the VST button.

## Presets

Most VST plugins include a set of preset data that you can use to instantly call up collections of parameter settings. FlowStone includes support for presets. All the standard controls (knobs, sliders etc.) already contain components for storing and preset data. To make use of this all you need to do is add a Preset Manager component inside your plugin module.



Use the Preset Manager properties to set the total number of programs, their names and whether the preset data is to be locked or not. When presets are locked any changes you make to them are lost when you change program – this applies to exported presets as well as inside SynthMaker.

## Timing Info

Some VST plugins depend on information that is sent from the host application where it is being used. Typical examples are the sampling rate, tempo and whether the host sequencer is playing. You can get access to such information using special components. These components are:

Sample Rate, Tempo, Time Signature, Delay Compensation, Is Playing, Bar Start Position, Sample Position, PPQ Position, VST Editor Open.

For information on how these work please refer to the components reference guide..

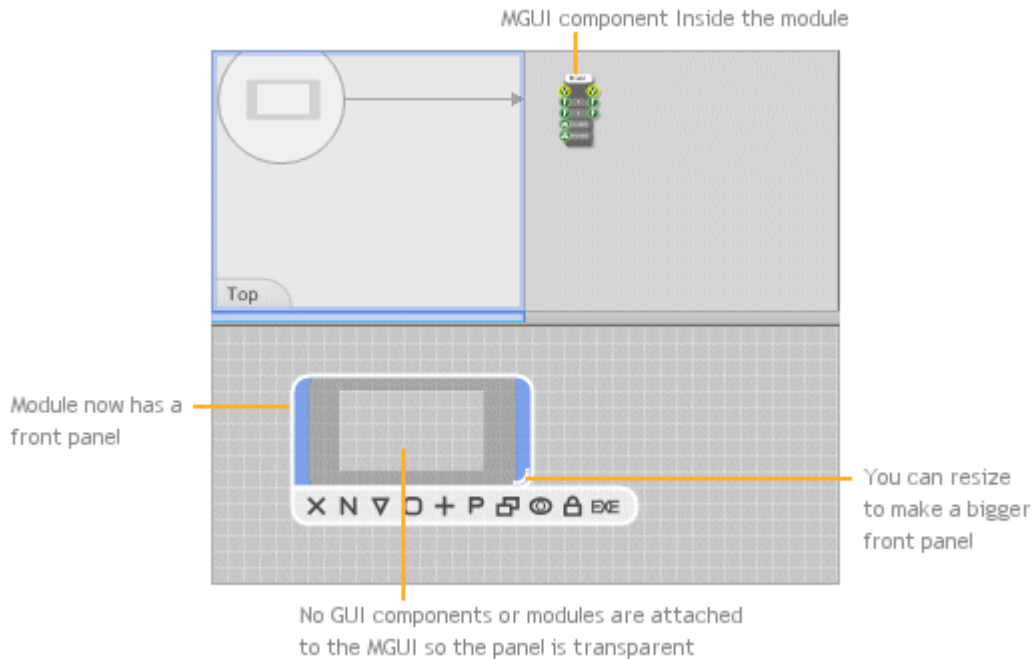
# 7 **Advanced GUI Editing**

THE GUI COMPONENTS AND HOW TO USE THEM






Create a new module then drop an MGUI component inside. You can find the MGUI component under the Module filter group in the toolbox.



If you move back up a level you'll see that the module now looks a bit different. It now has a front panel, albeit with nothing in it. You can resize the module to make the panel bigger or smaller. You may also notice that the G button has been removed from the action panel.

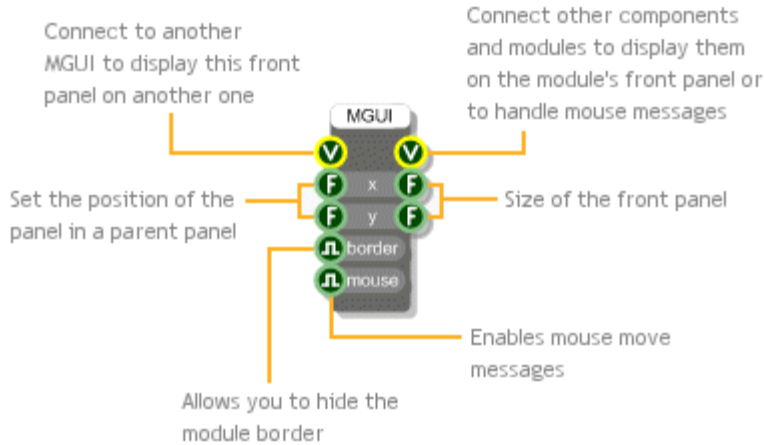
## MGUI Connectors

All GUI information is sent through View connectors. These are yellow circles with a V in the middle. The MGUI has one View output. Anything connected to this will either draw onto the front panel or handle mouse messages from it.

 View - handles all drawing and mouse messages

The two Float outputs can be used to get the size of the front panel if this is needed.











For the moment you only need to know about the output connectors. The input connectors come into play when things start getting more advanced. We'll cover this in a later section.



## GUI Connector Types

The GUI components introduce a new set of data types. Each type has its own connector and each connector has its own symbol. However, the symbols all have the yellow circle in common to show that they are GUI related.

We saw these first in the chapter on Data Types:

-  View - transports all drawing and mouse information
-  Area - an area defined by coordinates of top-left corner, a width and a height
-  Mouse - mouse events (left button up/down, mouse move etc.)
-  Colour - in argb format ('a' is the transparency level)
-  Pen - for drawing lines - defined by colour, thickness and style
-  Font - font information comprising typeface, size and style
-  String Format - alignment information for drawing text
-  Bitmap - 32bit image
-  Point Array - an array of points (floating point pairs)
-  Bitmap Array - an array of bitmaps

## Coordinate System

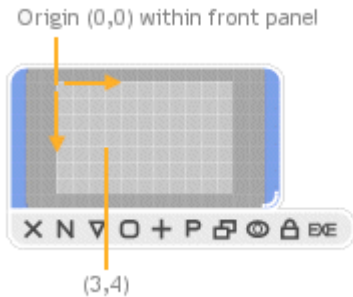
FlowStone uses a grid system to position components. Exactly the same grid system is used by the vast majority of GUI components for drawing and for mouse events.

The grid system is floating point based so you can have fractions of a grid square. This allows for more precise positioning of drawing elements.

Of course there's no getting away from the fact that the screen uses pixels and that these form a discrete grid of points. But by using special rendering techniques, FlowStone can still display graphics as if they were on a continuous surface.

### Working in Pixels

Sometimes you need to work in pixels instead of grid squares. To do this you can make use of the Grid Square to Pixel and Pixel to Grid Square components to move between the two systems.



# Drawing

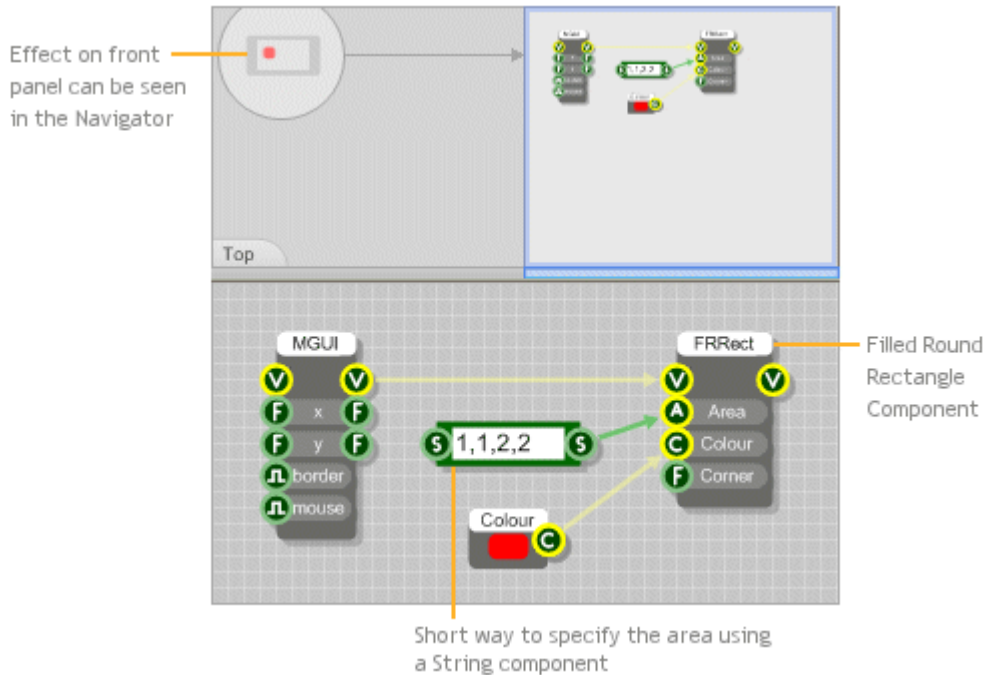
This section describes how to draw using the GUI components. With the introduction of the Ruby component you can now also draw using Ruby code. This is now the recommended way of drawing in FlowStone. For more details see the Drawing section in the Ruby Component chapter.

## Drawing on a Panel

Drawing on a front panel is a simple matter of picking a drawing primitive and connecting the View output of the MGUI to the View input of the primitive.

All the drawing primitives can be found by selecting the GUI filter group. There are primitives for drawing lines, rectangles, text, bitmaps and more.

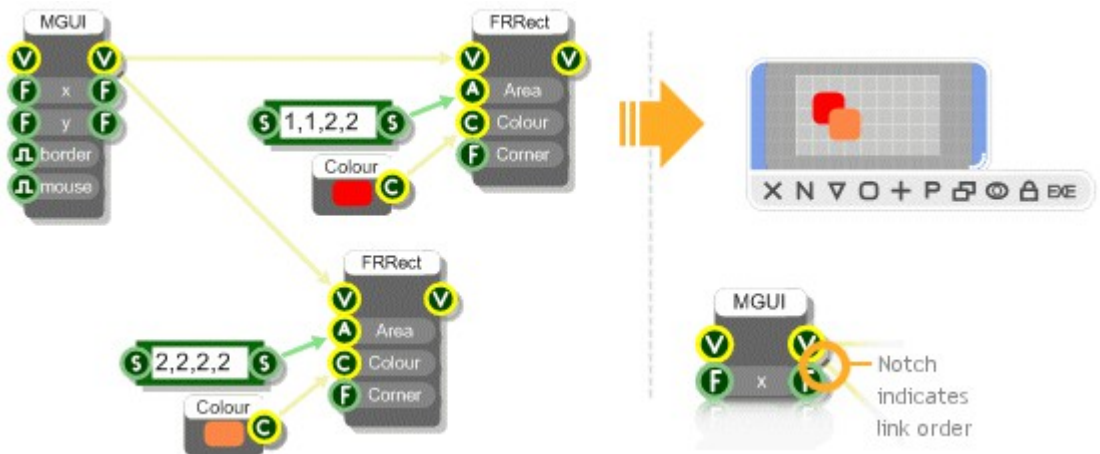
The example below shows how to draw a simple filled rectangle. We've used a little shortcut to specify the Area for the rectangle. This makes use of a string component to specify the x,y,width and height that define the Area. The colour is defined using the colour component.



## Drawing Order

Often you'll be drawing more than one element on a front panel. If two elements overlap then the one that is last in the link order will be displayed last and therefore over the top of the other element.

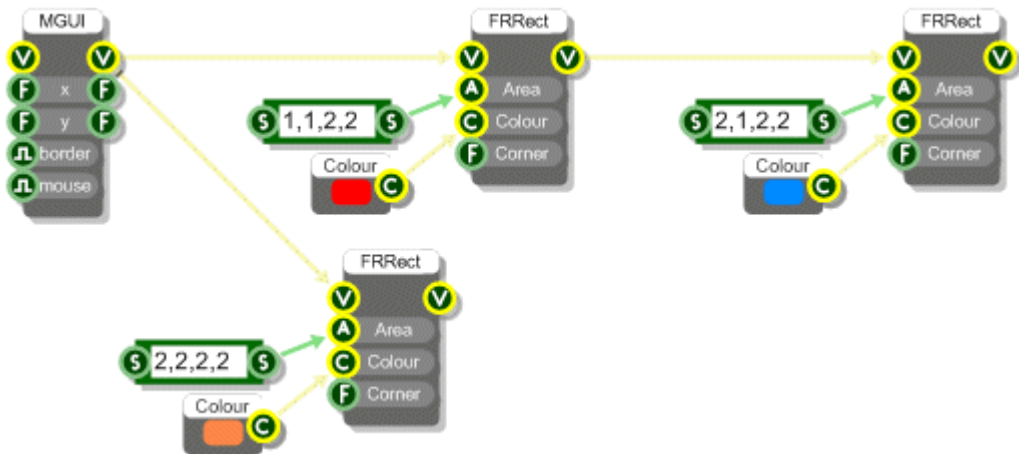
The example below shows this. The orange rectangle is on the second link from the MGUI (see Links for more on link order). The orange rectangle will therefore be drawn on top of the red one.



## Chaining GUI Components

Most GUI components have a view output connector. This allows other GUI components to be linked to them so chains of graphical elements can be created.

The example below shows how this is done. The effect on the link order is as follows: the link to the red rectangle is taken first followed by any links from its output connector. Then it's back to the next link from the MGUI etc. In this example the red rectangle would be drawn under the blue one which would then be under the orange one.



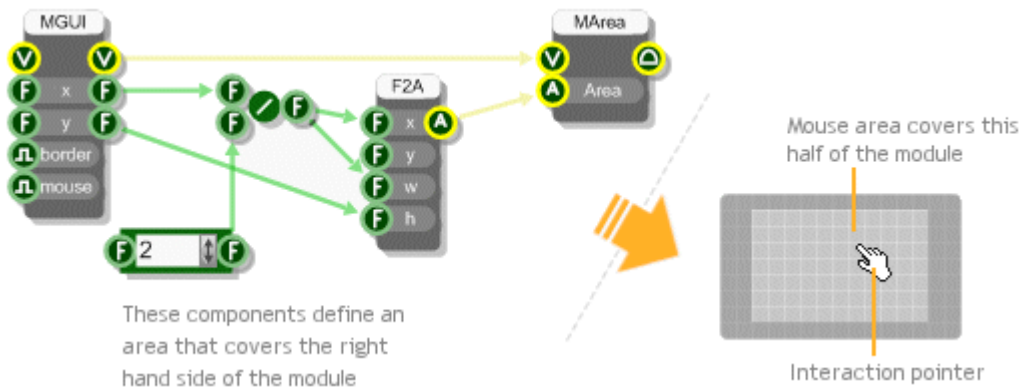
# Mouse Handling

This section describes how to handle mouse interaction using the Mouse components. With the introduction of the Ruby component you can now also do this using Ruby code. For more details see the Interaction section in the Ruby Component chapter.

## Mouse Area

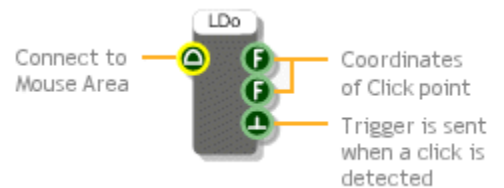
In order to receive mouse messages in a part of your front panel you must first define a mouse area. This is done using the Mouse Area component.

The example below shows how the right half of a module can be made to receive mouse messages. This is indicated by a change in cursor as the mouse pointer passes over the mouse area.



## Mouse Clicks

To trap mouse clicks on a mouse area you'll need a Mouse L-Button Down component. Link the Mouse output connector on the Mouse Area component to the input on the Mouse L-Button Down component.



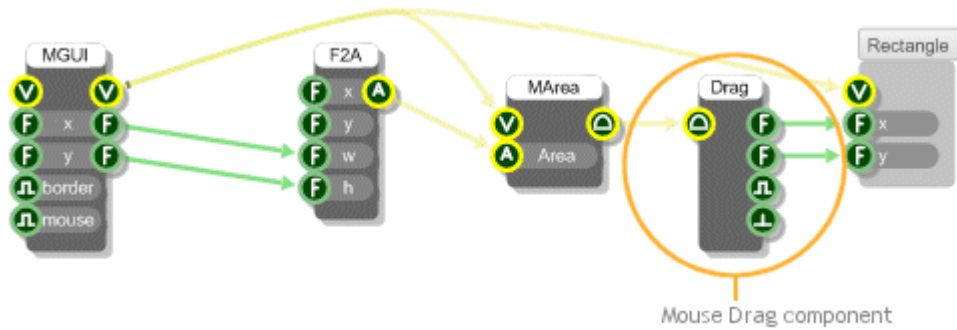
Whenever you click in the mouse area a trigger will be sent to the Trigger output on the Mouse L-Button Down. The coordinates of the click point (in grid squares) will be available from the two Float outputs.

## Mouse Dragging

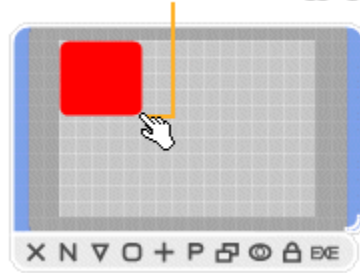
You can track the mouse position while the left mouse button is held down. This allows you to implement drag operations.

The component you need to do this is the Mouse Drag component. This takes Mouse messages at it's input and sends the coordinates to it's two Float outputs as you drag.

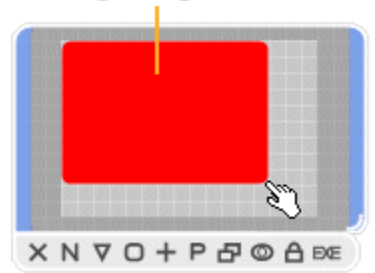
The example below shows how the Mouse Drag component can be used to change the size of a rectangle. We've created our own rectangle module to handle the drawing in this example. This makes the schematic a bit neater.



Click and hold then start dragging



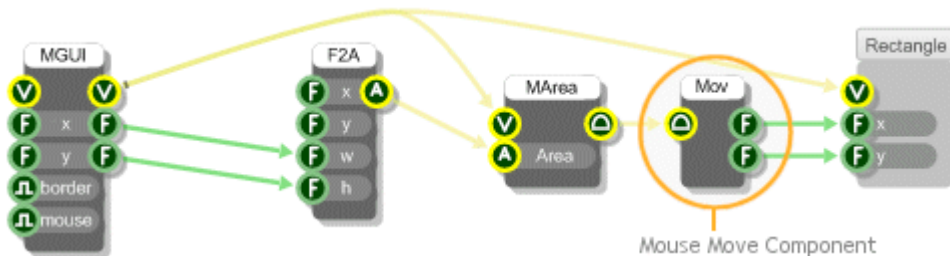
Rectangle changes to reflect mouse





## Mouse Moves

You can also track the mouse as it moves over a mouse area. This is done using the Mouse Move component. Before you can use this you need to enable mouse move messages on the appropriate MGUI component. Mouse move messages are disabled by default to reduce unnecessary performance overheads.



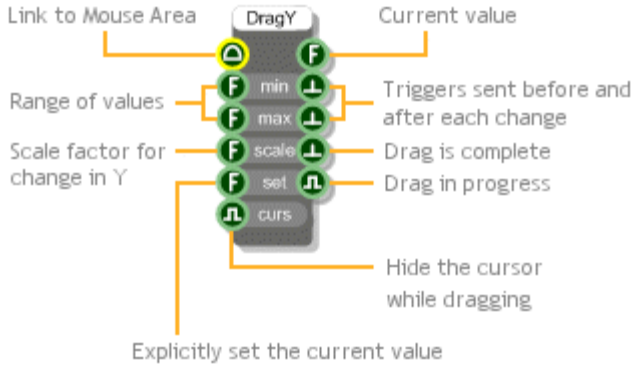
Try replacing the Mouse Drag component in the previous example with a Mouse Move component and you'll see how this works.

## Drag Accumulate

The Mouse Drag component is a bit too low-level for some tasks. If you want to create a slider or anything with moving parts that can be dragged around then it's much easier to use a Drag Accumulate component.

Drag Accumulate components manage much of the legwork of drag operations for you. There are three varieties: X, Y and XY.

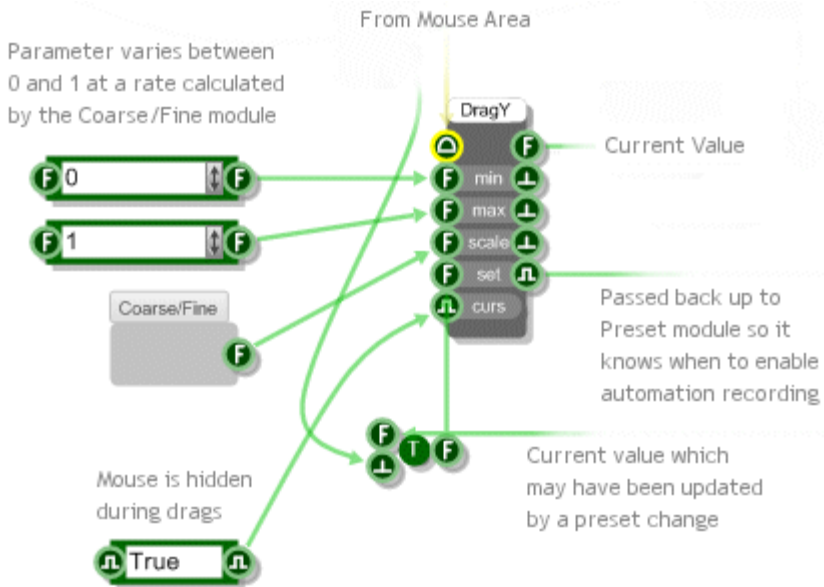
The Y Drag Accumulate manages a parameter that varies over a particular range. When you drag the mouse, the offset from the point where you clicked is maintained by the component. As you continue dragging the parameter is updated according to the Y position of the mouse and a scale value that you can specify.



If you have a look deep inside the Bitmap Knob module in the toolbox you'll see how the Y Drag Accumulate component is used. Move through the following modules:

Bitmap Knob\Knob\Interaction\Knob\Control\Moving Part\Knob Control

Inside the Knob Control module and you'll see the Y Drag Accumulate in the middle. The picture below explains what's going on.



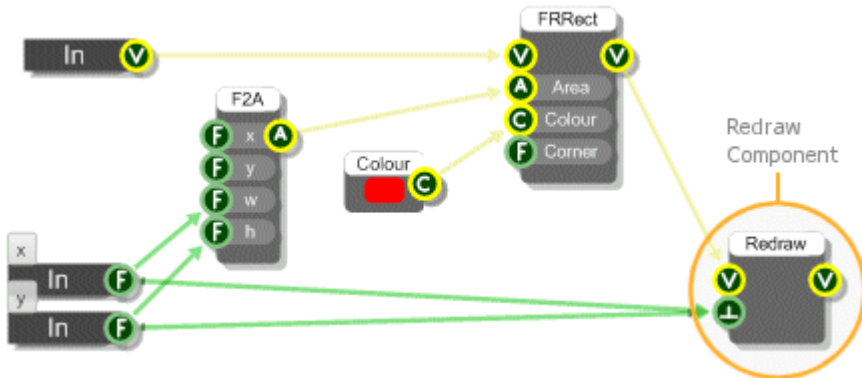
# Redrawing

## Redraw Control

If you change a property of a drawing element during a mouse operation like dragging, you'll often want the changes to be reflected immediately in the module front panel.

To allow for maximum flexibility FlowStone allows you to control when parts of the front panel are redrawn. In the rectangle dragging example we used the simplest kind of redraw - we just forced the whole panel to refresh. This was done using the Redraw component.

If you look inside the Rectangle module you'll see the Redraw component. When the component receives a trigger it sends a message back up through the View links to the first MGUI it finds. When the MGUI receives the message it redraws everything on it's front panel.

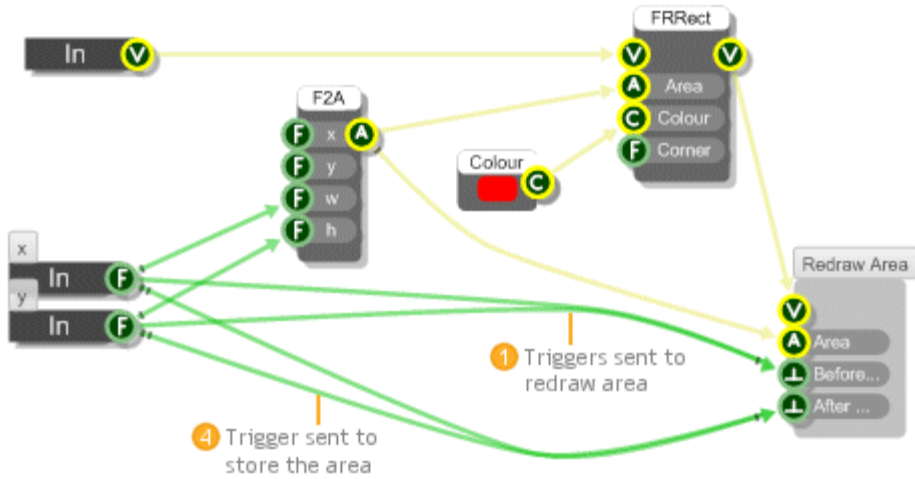


## Precision Redraws

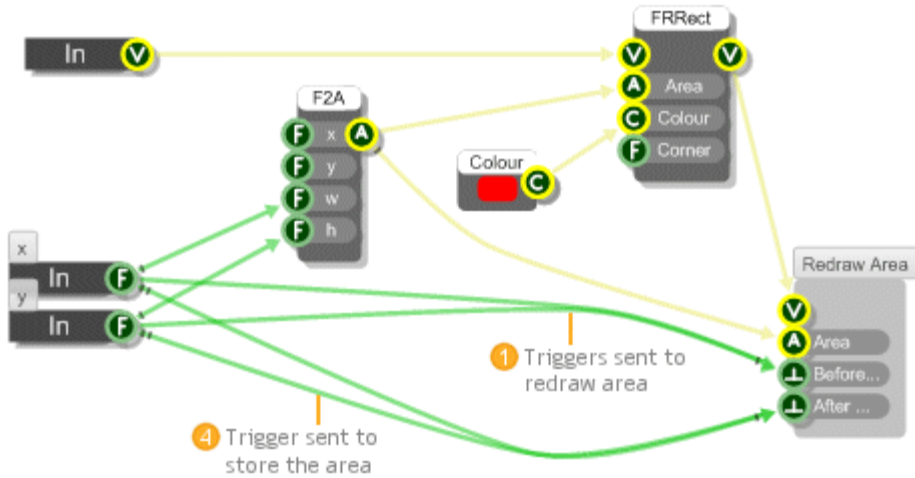
Redrawing the whole panel each time can be slow when the area is large (try resizing the module in the example above so that it's very big - you'll notice that dragging becomes sluggish).

Often only a small area of the panel is changing at any one time so it's much more efficient to redraw only the bit that has changed. For this purpose we have the Redraw Area component. This works in exactly the same way as the Redraw component except that it only redraws the area that you supply to it's Area input.

We've modified the drag rectangle example so it uses the Redraw Area component. Unfortunately it's not just a simple case of linking the rectangle area to the Redraw Area component. This is because the area before the last mouse move may need to be redrawn too. What we need is the combined area of the old and the new rectangles.



For this example we've created a module that handles the storage of the old area and its combination with the new area. This uses an Area Sample and Hold to keep the old Area for when it's needed.



8

# Ruby Component

A WORLD OF POSSIBILITIES

# Introduction

The Ruby component is by far the most flexible component in the FlowStone toolbox. It encapsulates the full Ruby language, it allows passing of data via an efficient time precise event system and it integrates fully with FlowStone triggered data types, mouse events and graphics.

The Ruby component can be used for anything from simple equations to complex class libraries, user interaction and graphics rendering. It really does open up a world of possibilities.

For the rest of this chapter we assume a basic knowledge of the Ruby language which is not covered in this guide. If you want a great reference we recommend the following site:

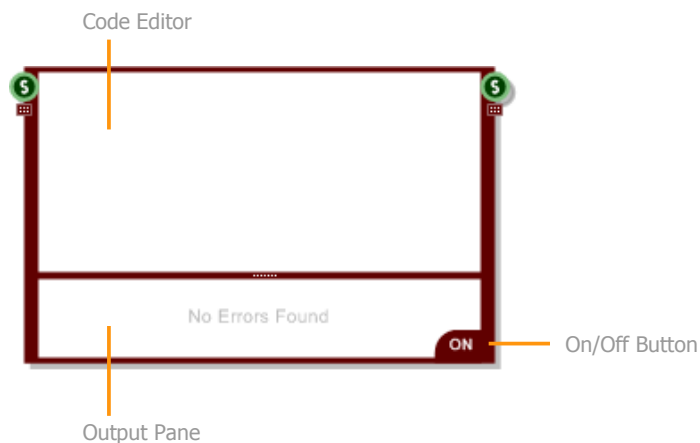
<http://www.ruby-doc.org/docs/ProgrammingRuby>

## Overview

The Ruby component is split into two halves. At the top is the Code Editor. Click here and type your Ruby code.

Below the editor is the Output Pane. This area shows any error messages, output from evaluated code and watched data values. You can change the relative size of the Editor and Output areas by dragging the separator bar.

The On/Off button can be used to prevent the Ruby code from being evaluated. This can be useful if you want to write code that might cause instability while it is being typed in (while loops are one such example).

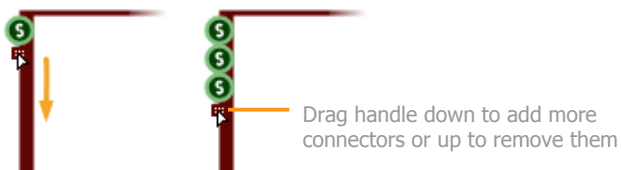


# Inputs and Outputs

The Ruby component can interface with most other triggered data types in FlowStone. By default the component has one String input and one String output.

## Adding or Removing

To change the number of inputs and outputs just click and drag the handles under the bottom most connectors.



## Changing Type





To change the type of a particular connector simply right-click on the connector and choose the type from the resulting menu. This is similar to the way you can set the type for a Module Input or Output.



## Inserting, Deleting and Moving

If you don't want to upset the configuration of connectors you already have then you can add or remove at a specific point.


These operations are performed by right-clicking on a connector. The right-click menu has a number of buttons at the bottom for deleting or moving the connector you've click on or for adding a connector below it.

-  - adds a new connector below the one you clicked on
-  - deletes the connector you clicked on
-  - moves the connector up one place in order
-  - moves the connector down one place in order

## Naming

It's useful to be able to name connectors, not only for readability (so it's clear what the data represents) but also because in the case of input connectors the label can be used within your Ruby code as a variable or as a reference to an input.

There are two ways to add connector labels. First you can right-click on the connector and then click the N button.

-  - opens up an in-place edit box where you can name or rename the connector

Type the name in the resulting edit box and hit the Return key.



Another way to name connectors is to first select the Ruby component then hold CTRL and hover the mouse near the connector whose label you wish to edit. An I-Beam cursor will appear together with a text box boundary. Click in the text box, type the name and hit return.

If you want to set multiple labels at once simply tab between the labels instead of hitting the Return key. If a connector already has a label you can use the same methods as above to edit it.



# Code Editor Basics

The code editor is where all the action happens. This is where you type in your Ruby code, have it process the input data and send any results to the output(s).

The editor has syntax colouring to make code more readable. It supports all the operations you'd expect including cut, copy and paste (CTRL+X, CTRL+C and CTRL+V), automatic scroll bars and mouse wheel scrolling.

A local undo is also implemented to allow you to undo and redo typing, deletions, pastes etc. The local undo applies to any changes you make during a particular session (Between clicking in the component and clicking away). After that you can still undo via the application's undo system, but this will only go back through changes made between edits.

You can page using the PGUP and PGDN keys. CTRL+HOME will go to the top and CTRL+END will go to the bottom.

You can Find search strings by using CTRL+F. After finding the first occurrence you can find again by pressing F3.

You can also Replace text by using CTRL+H. Highlight a selection of text first if you want to search within that as opposed to all the text in the component.

We have used the fact that Ruby is an extendable language to implement a number of FlowStone specific classes and keywords. We have also defined a collection of methods that you can implement in your code so that it can more tightly integrate with FlowStone.

The rest of this section assumes you know what Ruby is and have a basic understanding of what a Ruby class, method and variable is.

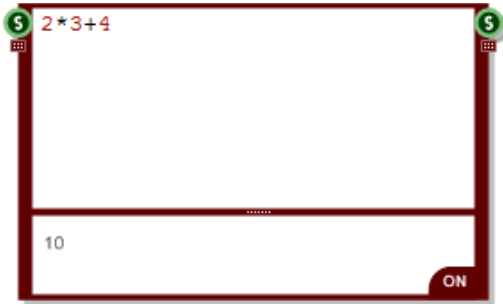
## The Output Pane

Before we get into the nitty gritty, a quick word about the Output Pane.

The code editor and output pane work together. Any errors or output from the code you type in the editor are displayed in the output pane.

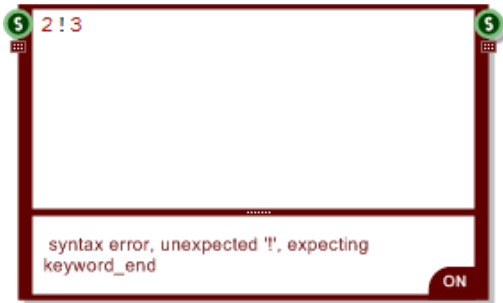
For example, you can type any valid mathematical expression into the Ruby component and the evaluated result is shown in the output pane.

CHAPTER 8



The example above shows 10 as the result of  $2*3+4$ .

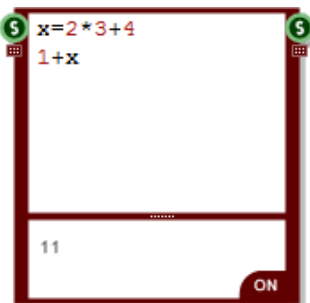
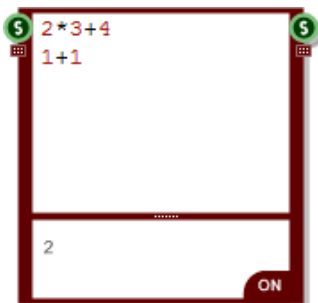
Type in an invalid expression or any syntax that causes an error and you'll get an error message instead. Note that only the first error encountered is displayed.



As you can see, the error message is displayed in red to differentiate it from evaluated output.

Note that the error messages come directly from the Ruby interpreter. They can sometimes seem unrelated to the error. However, the Ruby Component evaluates your code while you are typing so if an error does pop up you'll know exactly what change caused it.

One final thing worth mentioning is the output pane only shows the value of the very last evaluated expression. Take the examples below:



In the component on the left the first expression is evaluated, followed by the second expression. The second one is the last and so this gets sent to the output pane.

In the component on the right the same thing happens. However, in order to show that the first expression does indeed get evaluated we've assigned its result to a variable (x) and then used this in the second expression.

We now know enough about the Output Pane to proceed to the more exciting stuff.

## The RubyEdit Class

Each Ruby component is represented by an instance of the RubyEdit class. This is a class we have defined in FlowStone to represent the Ruby component.

We have defined methods and instance variables that allow you to communicate with the component. These provide the interface between your Ruby code and FlowStone.



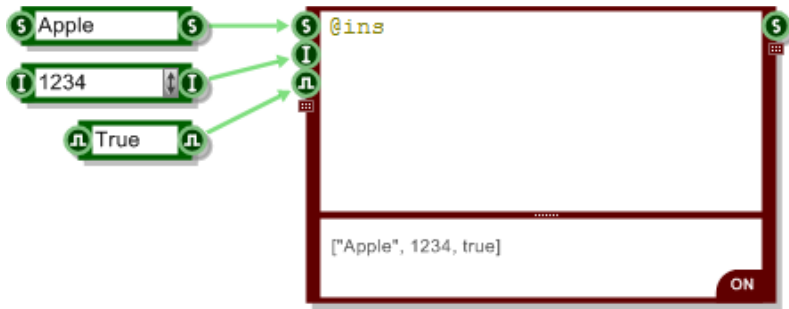
Typing **self** into the Ruby component will return the instance of the RubyEdit class that represents that component. In the above example you can see the instance representation in the output pane. You can also use the instance variable **@this** instead of **self**.

All code you type in a Ruby component is executed within the context of the RubyEdit object that represents it.

## Input Data

Data that arrives at a Ruby component is stored in an instance variable called **@ins**. This is a Ruby array and it stores the last value to arrive at each input.

You can look at this by simply typing it into the component. Note that instance variables are coloured gold in the code editor.

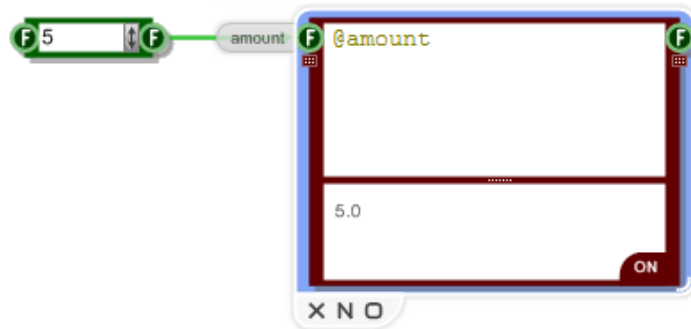


To access the value at any input use the Ruby Array element reference `[]`. Values are zero indexed so to get the value at the second input use `@ins[1]`.

It's quite common to have only one input so we've added another instance variable called `@in` that references this first input value directly.

### Input Labels

If you set a label for your input then this is automatically translated into an instance variable that you can use within your Ruby code.



Note that as the variable is an instance variable it must be preceded with the `@` symbol.

## Output Data

As well as receiving data from FlowStone you can of course send data out. To do this use the **output** method.



Note that **output** is not a keyword, it's a method of the RubyEdit class. Usually you would invoke a method on an object. However, in the case of the Ruby component all code is evaluated in the context of the RubyEdit instance that represents it.

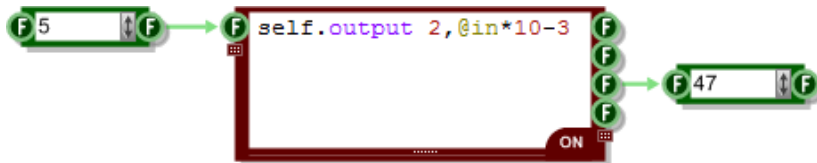
You could just as easily write **self.output** and you'd get the same result.

For this reason methods of the RubyEdit class are shown in purple so that you can distinguish them from Ruby keywords (which themselves show in blue) and inherited methods (which show in black).

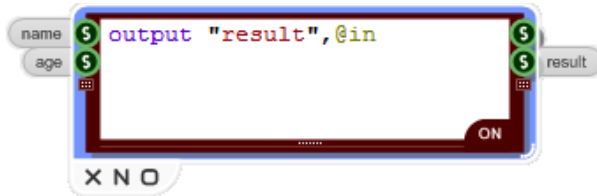
### Specifying an Output

In the previous example no output was specified and so the result is sent to the first output. If you want to specify a particular output then you add this as the first input to the **output** method.

The value can be the index of an output connector (starting from zero) as shown below:



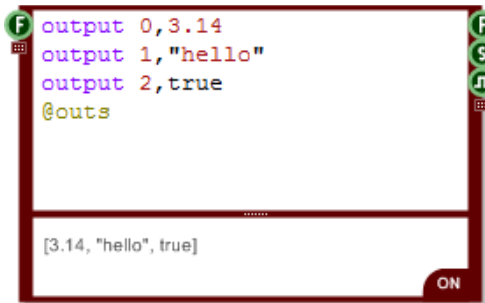
Or it can be the label of an output connector if you've specified one:



### Storage of Outputs

Just as the input data was stored in a Ruby Array we do the same for the output data. The very last value sent to an output is stored in an Array called `@outs`.

You can inspect the contents of this array in the same way as for the inputs array. This can be useful when debugging.



You can change the contents of the `@outs` array. This can be useful if you want to change a value at an output without sending an event or trigger. The value can then be read by other components that connect to the output.



This can be very useful if you have events data running at a high rate and you want to read it via triggered data at a lower rate. The example above shows a Tick25 being used to read the output at 25Hz regardless of the rate of change of the value inside the Ruby component.

# The event Method

As we mentioned earlier, the Ruby Component evaluates your code as you type. If your code contains expressions (as opposed to declarations or definitions) then these are executed and the last result is displayed in the Output pane.

Outside of the editor your code will also execute whenever data arrives at an input. This allows you to use the Ruby component for data processing.

This is all extremely useful but what happens if you want to respond to data received at one input differently from data received at another?

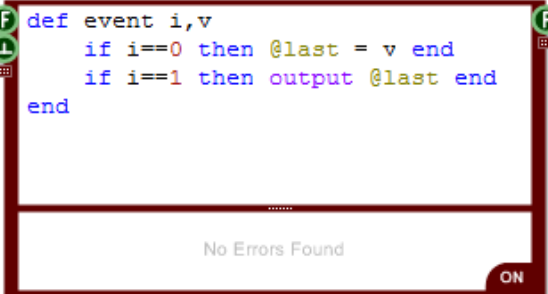
## Method Definition

For more advanced data handling you can define an **event** method. This is a special method which FlowStone looks for whenever it receives data at an input.

The **event** method can have up to 3 input parameters:

- **i** - references the input at which the data arrived
- **v** - value that arrived at the input
- **t** - time at which the data arrived (schematic time in seconds)

You can have 0,1,2 or all 3 input parameters but you must add them in the order. So for example, you can have no parameters or you can have **i** on its own or **i** and **v** or **i** and **v** and **t** but you can't have **v** on its own or **i** and **t** without **v**. More on **t** in the next section, we'll focus on **i** and **v** for now.



```
def event i,v
  if i==0 then @last = v end
  if i==1 then output @last end
end
```

No Errors Found

The example above shows how you would define an **event** method that would act as a 'sample and hold'. Data arriving at the first input is stored. Any trigger at the second input results in the last value being sent out.

If you don't supply a **v** input and just have a connector reference then you won't be passed any data. The **event** method will still be called when data arrives at any of the inputs. You'll know which input got triggered you just won't know what data arrived.

Note that you can name the input parameters whatever you like, you don't have to use **i,v** and **t**. However, the method name must always be **event** as that's what FlowStone will be looking for.

## Connector Referencing

You might think that the connector reference parameter passed to the event method is an integer value but it is in fact an instance of the **RubyEditConnector** class.

Objects of this class can be treated as integer indexes when used in a comparison situation. However, under the hood the class not only encapsulates the connector index but also any connector label that you may have assigned.

We have added overrides of the standard comparison operators to the **RubyEditConnector** class which means that as well as comparing with integer values in the event method you can also compare directly with a string.

The previous example showed how it can be used to compare with an integer. Comparing with a string (if you have a label for the connector) is just the same:

```

def event i,v
  if i=="name" then output 0,"name" end
  if i=="age" then
    output "result","age"
  end
end

```

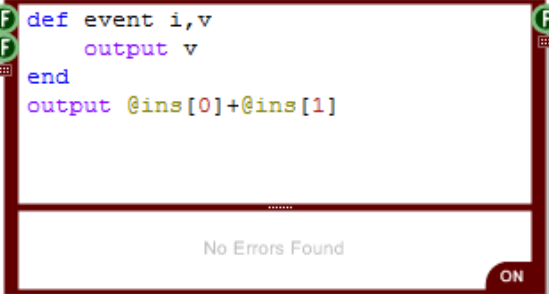
Comparison will work in case statements too. You can also use any of the inequality operators. Mathematical operations can also be performed on the **RubyEditConnector** object and return an integer result.

If you need you can access the individual index or name of a **RubyEditConnector** object using the **index** and **name** methods.



## Effect on Code Execution

If you supply an **event** method then any code you define outside of the method will no longer execute when an input changes. The example below shows an **event** method which simply outputs the most recent value received.



```
def event i,v
  output v
end
output @ins[0]+@ins[1]
```

No Errors Found

Prior to adding this method the remaining code would have executed whenever any of the inputs changed thus resulting in the sum of the inputs being sent to the output. However, with the **event** method in place FlowStone executes this method instead and ignores any other code.

# Scheduling Events

You may recall from the Data Types and Signal Flow section that, unlike triggered data which is sent immediately, event data can be scheduled to be sent at some time in the future.

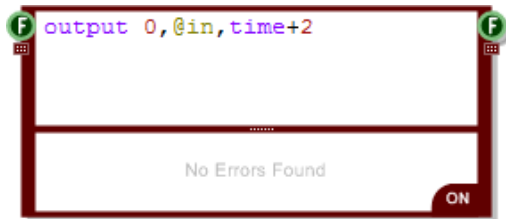
To do this a schematic has a clock. The clock is an elapsed time, in real world seconds, which begins when the schematic is loaded or created. The clock does not represent a time of day, it is simply a counter which continually increases.

## Scheduling an Event

To schedule an event you need to provide a time stamp when you send data to an output. This needs to be supplied as a third input parameter when you use the **output** method.

The time stamp will almost always be an offset from the current time. You can get the time in two different ways depending on where you make the call from.

From any location you can use the **time** method. If you're in the event method then if you define it with input reference, value and time inputs then you will have the time passed to you via the time input parameter.



The event will be sent to the events queue and will only be sent out through the designated output when the clock time for the event time stamp is reached.

The example below shows how to schedule an event from inside an event method using the time input parameter.

```

F def event i,v,t
  output 0,@in,t+2
end

```

No Errors Found

ON

## Sending to an Input

You can send events back to the same Ruby component by sending them to an input. This is useful if you want to create a repeating event or perform a recursive operation.

To do this use the **input** method. This works in exactly the same way as the **output** method but of course the connector reference refers to the input you want to send the event to instead.

```

F input 0,@in+1,time+1
  output @in

```

No Errors Found

ON

→ F 45

The above example creates a counter that moves in one second steps and continues without end.

Note that, just like for the output method, you can use a connector label instead of an index to identify where you want to send a value to.

## Scheduling Methods

You can also schedule method calls using the **scheduleMethod** method.

**scheduleMethod** methodName, arg1, arg2, ... , time

Where methodName is a string specifying the name of the method, arg1, arg2 etc. is a list of argument values to pass to the method (you don't have to supply any) and time is the time that you want the method to be called (use the **time** method to get the time now in seconds and offset from that)

## Clearing Events

Once you send an event it goes onto the events queue and waits until its time comes to be executed.

If you're scheduling events that happen some time in the future sometimes you might want to stop them from happening. You can do this using the **clearEvents** method. This will remove all pending events for that particular Ruby component from the events queue.

## Clock Accuracy

The schematic clock runs at 100 Hz. Unlike the Tick components which are not time precise due to their use of Windows timers, the Events system uses a different timer which is much more accurate so each 10 millisecond tick should occur precisely on time.

If you have any of the DirectSound or ASIO primitives in your schematic and these are switched on then the clock will automatically switch to run in sync with the audio processing. You can then schedule events to occur with sample precise timing within any audio frame.

# Ruby Values

Earlier in this guide we talked about the Ruby Value connector type. This is used to pass data from one Ruby component to another.

It's a very simple concept but as you'll see, it's also incredibly powerful as it gives you the flexibility to define and use your own data types.

We'll start with a quick reminder of what a Ruby Value is.

## The Ruby Value Type

In Ruby all data is considered to be an object. Numbers are objects, strings are objects, arrays are objects – everything is an object. Each object is an instance of some class. For example, an integer object is an instance of the FixNum class and an array is an instance of the Array class.

Some classes are subclasses of a parent class. FixNum for example is a subclass of the Integer class. Those superclasses can themselves be subclasses of another class. A key aspect of Ruby is that no matter how many superclasses there are they all end up at one common base class called Object. So all data is in fact an instance of the Object class and as such we can consider everything as having the same common base type.

When referring to ruby data in this common way we call it a VALUE. As such, when you pass Ruby objects between Ruby components you pass them through the Ruby Value connector.



Value - can refer to any Ruby object

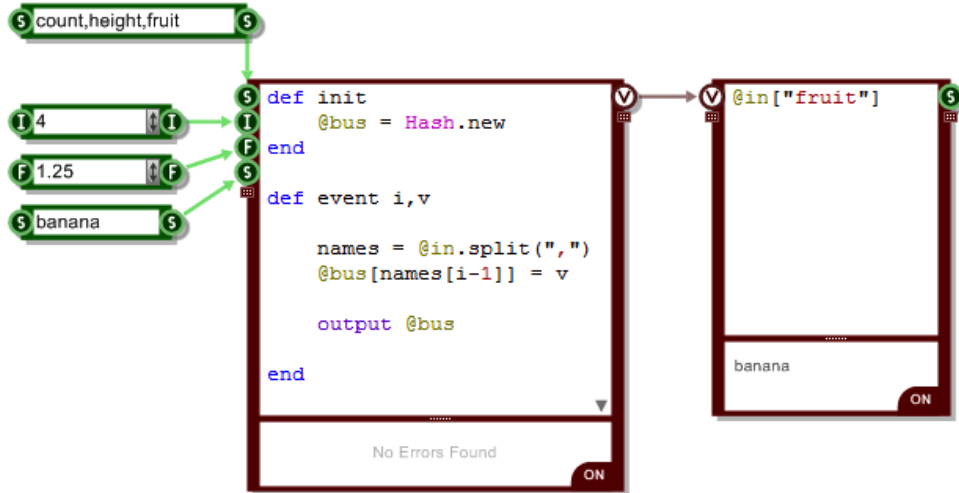
## Passing Ruby Values

Ruby values can be passed from one Ruby component to another. Data is transferred using the Event system we talked about earlier.

Because everything in Ruby is an object you can pass anything you like between components. This opens up many possibilities.

For example, FlowStone has Bus components which are used to pass collections of data around via one single connector. The Ruby Value connector can act like a bus by passing a Ruby hash (a collection of data and string pairs).

The example below shows how a Bus can be created using the Ruby component.



Of course you can be even more clever than that. In the previous example we were passing a Ruby Hash. However, we can pass anything we like. So we could create our own class with its own data and pass that.

This next example shows a RobotArm class designed to model a multiple link robot arm. This is getting quite advanced now but you can see that the options for extending the types of data passed around are almost limitless.

```

class RobotArm
  def initialize
    @base = { :w=>8, :h=>3, :offset=>2, :angle=>0, :min=>-90, :max=>90}
    @links = []
    @links.push ({:len=>8, :w=>3.5, :angle=>50, :min=>-90, :max=>90})
    @links.push ({:len=>6, :w=>2, :angle=>-40, :min=>-90, :max=>90})
    @links.push ({:len=>4, :w=>2.5, :angle=>-50, :min=>-90, :max=>90})
    @links.push ({:len=>8, :w=>2, :angle=>-50, :min=>-90, :max=>90})
    @links.push ({:len=>4, :w=>1.8, :angle=>-50, :min=>-90, :max=>90})
  end

  attr_accessor :base, :links
end

def init
  @arm = RobotArm.new
end

def event i,v
  @arm.links[i][:angle] = v
  output 0, @arm
end

def save
  @arm
end

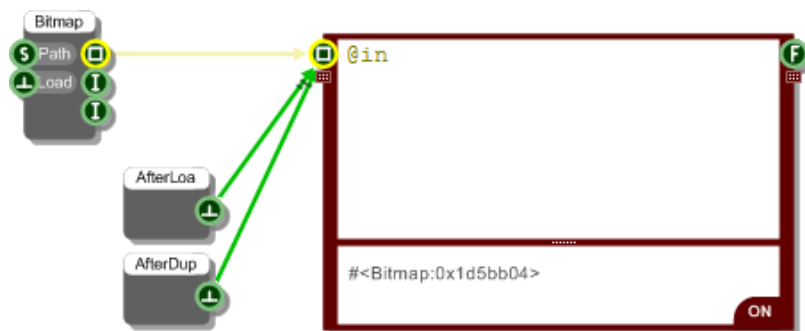
def load v
  @arm = v
end

```

## Persistence

The Ruby component saves its state. If you save a schematic with a Ruby component in it then when you load it back the **@ins** and **@outs** arrays are restored to exactly how they were before. This ensures that the component maintains its state at all times.

The only exception to this is bitmaps. Because bitmaps can be very large these are not automatically saved as part of the input and output arrays. If you want to restore a bitmap after loading use the After Load component to trigger a refresh of any input bitmap. In addition you may want to add an After Duplicate component too so that the bitmap refreshes when you copy and paste.



### User State Management

If you have your own data that you want to save with a Ruby component then you can implement the **saveState** and **loadState** methods:

```
def loadState v
end

def saveState
end
```

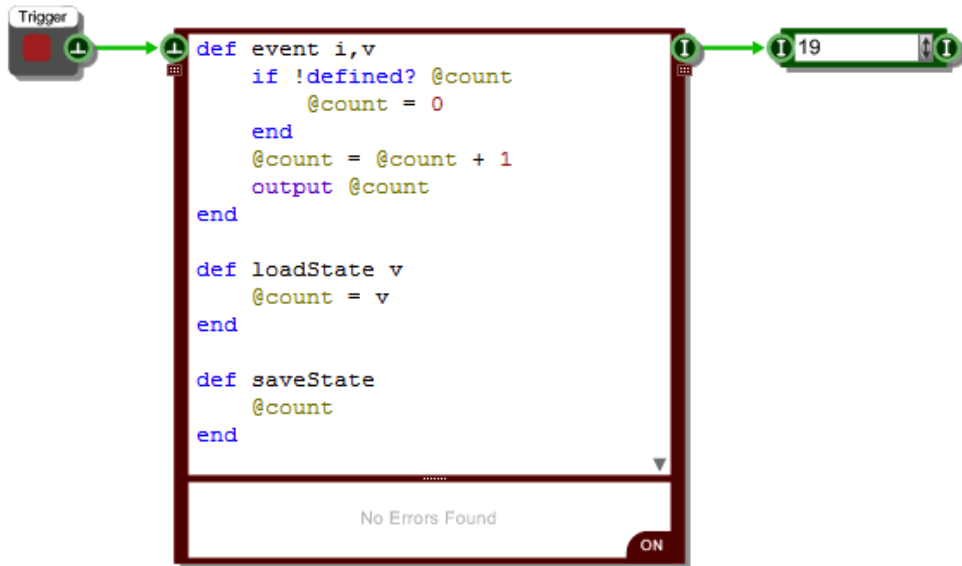
The way this works is really simple. In the **saveState** method you return a Ruby object that encapsulates all the data that you want to save. It could be a string or an array say if you have lots of data or it could just be a simple value.



When you save your schematic the `saveState` method is called by FlowStone and the state of your Rub component is saved as well.

When you open your schematic again FlowStone calls the `loadState` method and passes it the data you returned in the `saveState` method. All you need to do is use this data to restore your component state back to how it was.

Here's a little example of how this works. It shows a counter that retains its current value when saved:



If you have multiple values you want to save and restore then the easiest way to deal with these is to put them in an Array and return that in `saveState`. Let's say you have variables `@x`, `@y`, `@name`. Here's some example code that would save and restore these:

```

def saveState
  [@x,@y,@name]
end

def loadState v
  @x, @y, @name = v
end

```

In `loadState` we use Rubys syntax for assigning variables to consecutive elements in an array.

`@x, @y, @name = v` is equivalent to: `@x = v[0]; @y = v[1]; @name = v[2];`

When using this method always make sure you maintain the same order for the variables in your `loadState` and `saveState` methods or you might get some strange results.

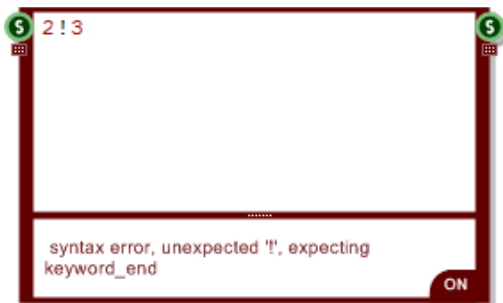
# Debugging

FlowStone is designed to make programming easier. However, it can never be foolproof and so at some point you are bound to find yourself facing bugs which have you scratching your head.

This aim of this section is to provide you with information, hints and tips that will hopefully make the debugging process much less of a headache.

## Error Reporting

The first thing you need to know when you have an error is where it is. As we discussed way back near the start of this whole chapter, the Ruby component evaluates your code while you are typing. If an error occurs it is displayed immediately in the output pane.



Hopefully this will help you eliminate most problems immediately when they occur.

There are some situations where errors are not reported straight away. Errors inside any methods you define will only show up when the method is called. There are some exceptions to this but most of the time the error will only show itself on executing the method.

When this happens you will still get an error message. However, unlike when you're typing you may have moved to another part of your schematic and not be able to see the component when the error appears.

To help with this, FlowStone highlights any module that contains a Ruby component that has an error.



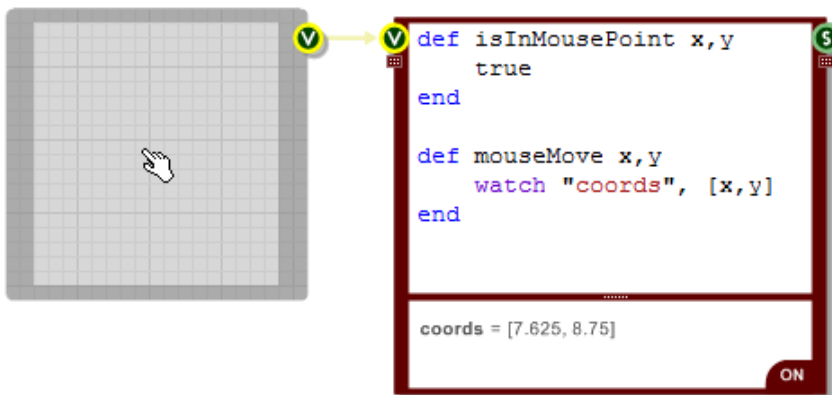
This highlighting propagates right up to the top level so no matter where you are in your schematic, you should see errors show up on the navigator and you can drill down through them to find the cause.

## The watch Method

If you're debugging your code it's often very useful to be able to look at the values of variables as your code runs. You'll recall from earlier in the chapter that the value of the last evaluated expression is shown in the output pane. This is great but what if you want to see a value that is calculated earlier or inside a method?

To help with this we have the **watch** method. This method takes two inputs. The first is a label to identify the watched data in the output pane. The second input parameter is the value you want to watch. The label can be a string or an integer value.

The example below shows how the mouse coordinates can be watched as the cursor moves across the View (not that this requires mouse move messages to be enabled on the MGUI inside the module – see the Interaction section for more on this).



If you just want to quickly see one value in a Ruby Edit then you can omit the string label input. This will show with the label 'Quick Watch' in the output pane. Note that quick watch will only show the last value sent to it so if you want to watch more than one either put all your watches in an array ( eg. `watch [x,y,name,pos]` ) or use a label for each watch separately.

# Drawing

As well as handling data the Ruby component can also be used to handle mouse events and draw to a front panel. This section covers drawing and the next handles mouse interaction.

To enable your Ruby component to draw to a front panel you need to add a View input connector and then link this to a View output connector from an MGUI (either directly or by a series of links) or from a Wireless Input that connects to a front panel in a module above.



Once you have this set up you can start responding to draw requests from FlowStone.

## The draw Method

To respond to redraw requests from FlowStone you'll need to define a **draw** method. FlowStone looks for this whenever drawing messages arrive at a View connector on a Ruby component.

```

V def draw i,v
  # Drawing code goes here
end
S

```

No Errors Found

ON

The method can have an optional input connector reference parameter (if you're handling multiple View connectors on the same Ruby component) followed by a mandatory View parameter.

The View parameter is an instance of the View class. This is a Ruby class that references the front panel and allows you to draw to it. Think of it as the drawing surface.

## Drawing Classes and Methods

Drawing is performed by calling methods of the View object. There are also a number of Classes which represent drawing objects.

All the methods and classes you'll use are wrappers onto Microsoft's Gdi+ graphics library. There are some minor variations but in general they map directly onto the methods and classes of Gdi+. There are many resources online dedicated to this so if you need any further help or examples you should be able to look these up and translate them directly into the FlowStone equivalents.

### Drawing Units

Before we begin we should quickly talk about the units of measurement we use for drawing. Because all drawing in FlowStone is scalable we work in grid step units and not in pixels (see the Coordinates section in the chapter on Advanced GUI Editing). All coordinates, measurements and sizes are in grid steps.

If you ever need to know the size of a grid square you can call the gridStep method on the View object at any time.

## Pens, Brushes & Colors

Before you can draw anything you'll need either a Pen, a Brush or a Color. These are all represented by Ruby classes. Pens are used for drawing lines and Brushes are used for filling areas. The Color class is used to define Pens and Brushes and as an input in other drawing methods.

### Color

You can't get anywhere without a Color so let's start here. Color objects are defined by three primary colour components (red, green and blue) plus an optional transparency which is called Alpha. Each component is an integer value in the range 0-255.

To create a Color object:

```
myColor = Color.new a,r,g,b
```

Where a is the alpha, r is the red component, g is the green component and b is the blue component.

Here are some examples of creating Color objects.

```
c = Color.new 255,255,0,0 # an opaque red
c = Color.new 128,0,0,255 # a half transparent blue
c = Color.new 0,255,0 # an opaque green (no transparency)
c = Color.new 64 # a dark grey (a single value is a grayscale)
```

## Pens

At their core, Pens are defined by a color and thickness (or width). They can also have a dash style, end caps like arrows or other shapes and they can render their joints in a number of different ways. These are more advanced features that we'll look at later.

To create a Pen object:

```
myPen = Pen.new color, thickness
```

The color parameter is a Color object that you need to create and pass in. Thickness is the line width (again in grid steps, so this is usually a number less than 1 for thinner lines).

If you need it, you can get the thickness or width of a pen object by calling the **getWidth** method. You can also set the width using the **setWidth** method.

The colour can be set by calling the **setColor** method and passing a Color object.

## Brushes

Brushes can be solid colours, textures, or gradients. The simplest brush is the solid brush. This is represented by the Brush class.

To create a Brush object:

```
myBrush = Brush.new color
```

Once again, the color parameter is a Color object that you need to create and pass in. As with pens, the colour can be set by calling the **setColor** method and passing a Color object.

## Basic Shapes

You can draw a number of different basic shapes using methods of the View class. They all take a drawing instrument (either a Pen for outlines or a Brush for fills) and then a number of other parameters to define position and size.

The parameters are usually points, collections of points or rectangles. These are represented by Ruby arrays. A point is represented by a two element array with the first element being the x coordinate and the second being the y coordinate.

A rectangle is represented by a four element array. The first two elements are the x and y coordinates of the top-left corner followed by the width and the height.

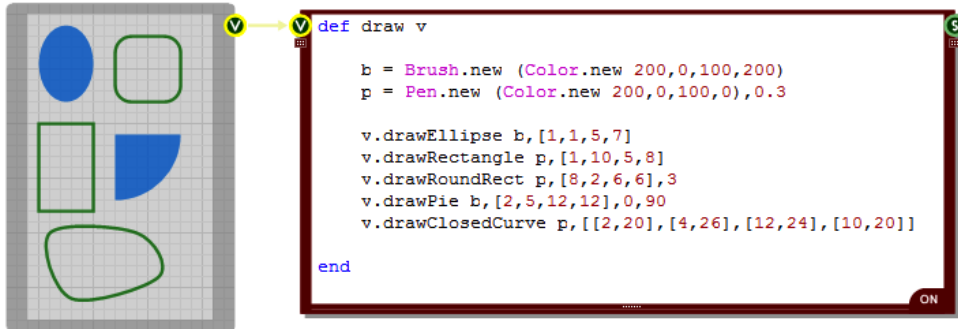
A collection of points is represented by an array of two element point arrays.

There are 6 shape methods:

**drawRectangle** instrument, rect

**drawRoundRect** instrument, rect, corner  
**drawEllipse** instrument, rect  
**drawPie** instrument, rect, startAngle, sweepAngle  
**drawPolygon** instrument, points  
**drawClosedCurve** instrument, points

Here are some examples of drawing shapes:



The drawClosedCurve method can take an additional input:

**drawClosedCurve** instrument, points, tension

The tension input determines the curvature. A value of zero will produce straight lines, a value of 0.5 will produce curves at the standard curvature and increasing tensions will produce more curvy lines.

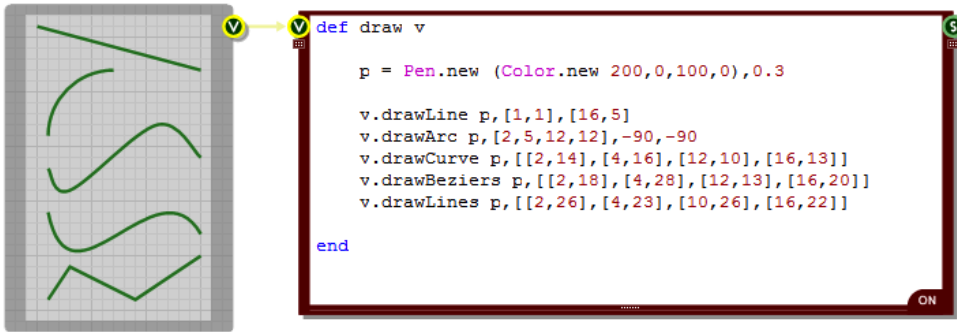
## Lines and Curves

The View class has a number of different line drawing methods. They all take a pen as their first input. The remaining parameters depend on the type of line. As with the shapes, Ruby arrays are used to specify points, collections of points and rectangles.

There are 5 line drawing methods:

**drawLine** instrument, point1, point2  
**drawArc** instrument, rect, startAngle, sweepAngle  
**drawCurve** instrument, points  
**drawBeziers** instrument, points  
**drawLines** instrument, points

Here are some examples of line drawing in action:



The drawCurve method can take two additional forms:

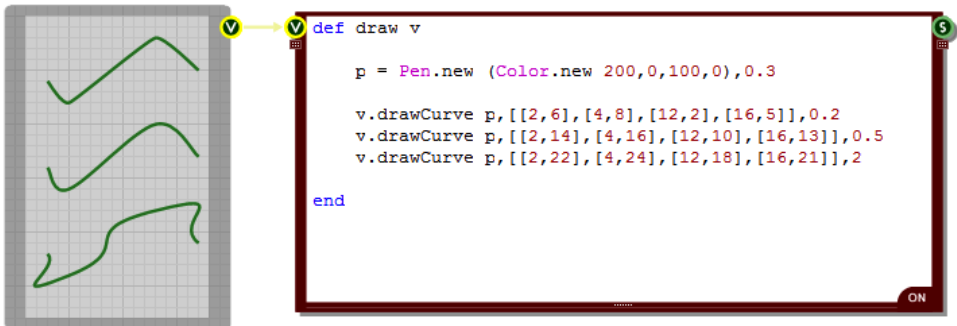
**drawCurve** instrument, points, tension

**drawCurve** instrument, points, offset, segments, tension

The tension input determines the curvature. A value of zero will produce straight lines, a value of 0.5 will produce curves at the standard curvature and increasing tensions will produce more curvy lines.

The offset value determines the point at which drawing will begin and the segments input determines how many segments of the curve are drawn.

The image below shows how the curvature changes for tensions above and below 0.5:





## Graphics Paths

A graphics path is a combination of shapes and lines which are grouped together, in sequence to form one single object. Paths are represented by the **GraphicsPath** class.

To create an instance of a GraphicsPath is easy:

```
path = GraphicsPath.new
```

After that you call methods on the GraphicsPath object to add elements to the path. There is one method for each of the line and shape methods in the View class. They take the same parameters except for the drawing instrument input.

There are 5 shape methods and 5 line methods as follows:

```
addRectangle rect
addEllipse rect
addPie rect, startAngle, sweepAngle
addPolygon points
addClosedCurve points

addLine point1, point2
addArc rect, startAngle, sweepAngle
addCurve points
addBeziers points
addLines points
```

Note that when you add line elements, consecutive lines will automatically get joined at their end and start points. You can then call the closeFigure method to make the end of the last line join with the start of the first to form a closed shape.

The **addCurve** and **addClosedCurve** methods can also take the same optional input parameters that the **drawCurve** and **drawClosedCurve** methods take. See the previous two sections for more information.

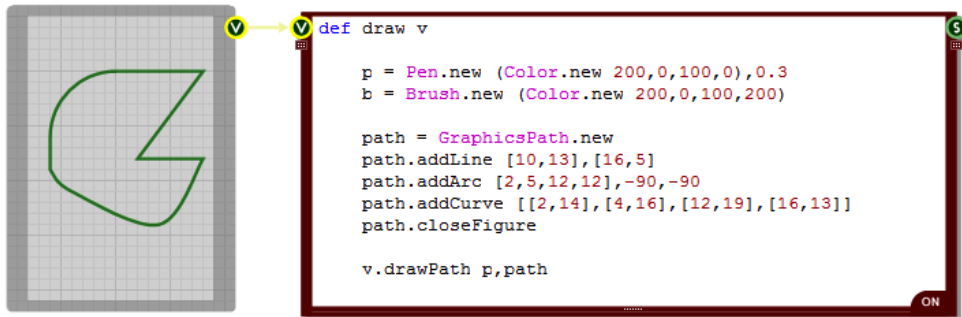
There is one additional add method of the GraphicsPath class called addPath. This allows you to append another path to the path. The method has two inputs: the path to be added and a true/false value to indicate whether the path is to connect into the existing figure (true) or be a new figure in the path (false).

```
addPath path, connect
```

Once you have created a GraphicsPath object you can draw it using the View class **drawPath** method. This takes a path and a drawing instrument (pen or brush) as inputs:

```
drawPath instrument, path
```

The example below shows how you might use a graphics path in practice:



### Other GraphicsPath Methods

You can get the bounding rectangle of a GraphicsPath object by calling the **getBounds** method. This will return a Ruby array of the form [x,y,w,h] giving you the smallest rectangle that contains the complete path.

To widen the path call the **widen** method:

**widen** pen, {optional View object}

The view object input is optional and serves only to provide scaling information. The pen input provides a pen to widen the path by. Widening is done by adding an additional outline to the path as if the path had been drawn in the pen.

You can check if a point is inside the GraphicsPath using the **isVisible** method:

**isVisible** point, {optional View object}

The view object input is the same as for the widen method. The point input is a two dimensional Ruby array [x,y] for the point you want to test. The return value is either true or false.

You can also check if a point is on the outline of a GraphicsPath using the **isOutlineVisible** method

**isOutlineVisible** point, pen, {optional View object}

The first two inputs are the same as for the isVisible method. The pen input defines the pen that the path would be drawn with. The point is then tested to see if it would lie on the drawn path. Obviously the wider the pen, the more likely this is. The return value is either true or false.

## Text

You can draw text to a View using the **drawString** method. However, before we talk about this you're going to need to know about two other drawing classes.

### The Font Class

The Font class defines the typeface you'll use for drawing your text. You need to supply the name of the typeface (this must be a font that is installed on your system), the size of font (in grid squares) and the style.

The style can be any combination of "normal", "bold", "italic", "underline" and "strikeout". For example, "bolditalic" or "strikeout-underline". You can have spaces in the strings or use hyphens or other symbols – as long as the key words are present then FlowStone will pick them up. You can also use an integer in the range 0-15 to specify style combinations as a bitmask. The bits are 1 = bold, 2= italic, 4 = underline and 8 = strikeout. So by adding these together  $1+4+8 = 13$  is the same as "bold-underline-strikeout".

Here's an example of how to create a font:

```
font = Font.new "Arial", 1.2, "normal"
```

### The StringFormat Class

The StringFormat class defines how text is placed relative to its location or bounding rectangle. There are 3 attributes to consider. Alignment determines how the text aligns horizontally. There are 3 options: near (left), center (middle) or far (right).

Line Alignment determines how the text aligns vertically. There are 3 options here too: near (top), center (middle) or far (bottom).

Finally Flags allows you to specify combinations of more detailed options. The flags input is an integer that should be a bitwise combination of the options you want. There are quite a number of options so to avoid cluttering things here we've listed them in full at the end of the next section.

Here's an example on how you might create a StringFormat object:

```
sf = StringFormat.new
sf.setAlignment "center"
sf.setLineAlignment "far"
sf.setFlags 4128
```

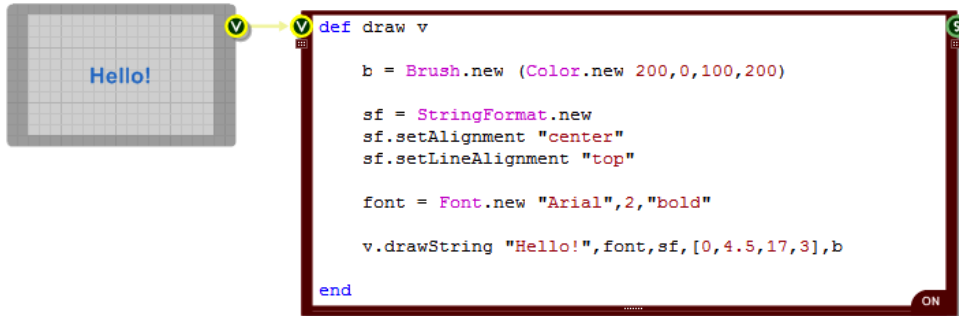
### Drawing the Text

Now let's return to the **drawString** method. This is defined as follows:

```
drawString text, font, stringFormat, location, brush
```

The location can be a point (an array of two values) or a rectangle ( [x,y,width,height]). If the location is a point then all drawing will be referenced to that point. If the location is a rectangle then the text will be drawn inside that rectangle.

Here's an example:



The StringFormat object can be replaced by nil in the drawString method call should you wish to go with the default alignment of top-left.

### Measuring Text

Sometimes you need to be able to measure how much screen space a block of rendered text will occupy. For this purpose there is the **measureString** method:

**measureString** text, font, stringFormat, location

The input parameters are the same as for the drawString method. However, if the location defines a rectangle then this acts as a maximum bounding box for the text the idea being that once measured the actual bounding box will be smaller than this one.

The drawString method returns a Ruby Hash containing the following items:

- “**bounds**” - the bounding rectangle that would be occupied by text [x,y,w,h]
- “**width**” - the width of the bounds rectangle
- “**height**” - the height of the bounds rectangle
- “**chars**” - the number of characters that would be drawn
- “**lines**” - the number of rows of text that would be drawn

A Ruby Hash is just like an array, except that you access elements by a reference object instead of an integer index (although you can use an index too if you wish). The Hash returned by **measureString** uses strings so for example, if the hash variable was called 'measure' then to get the width you would use the following expression:

```
measure["width"]
```

## String Format Flags

In the previous section on Text we talked about StringFormat flags. This section describes these in full. This information is taken directly from the Microsoft Developer notes on GDI+.

### **Flag = 1**

Specifies that reading order is right to left. For horizontal text, characters are read from right to left. For vertical text, columns are read from right to left. By default, horizontal or vertical text is read from left to right.

### **Flag = 2**

Specifies that individual lines of text are drawn vertically on the display device. By default, lines of text are horizontal, each new line below the previous line.

### **Flag = 4**

Specifies that parts of characters are allowed to overhang the string's layout rectangle. By default, characters are first aligned inside the rectangle's boundaries, then any characters which still overhang the boundaries are repositioned to avoid any overhang and thereby avoid affecting pixels outside the layout rectangle. An italic, lowercase letter F ( f ) is an example of a character that may have overhanging parts. Setting this flag ensures that the character aligns visually with the lines above and below but may cause parts of characters, which lie outside the layout rectangle, to be clipped or painted.

### **Flag = 32**

Specifies that Unicode layout control characters are displayed with a representative character.

### **Flag = 1024**

Specifies that an alternate font is used for characters that are not supported in the requested font. By default, any missing characters are displayed with the "fonts missing" character, usually an open square.

### **Flag = 2048**

Specifies that the space at the end of each line is included in a string measurement. By default, the boundary rectangle returned by the Graphics::MeasureString method excludes the space at the end of each line. Set this flag to include that space in the measurement.

### **Flag = 4096**

Specifies that the wrapping of text to the next line is disabled. NoWrap is implied when a point of origin is used instead of a layout rectangle. When drawing text within a rectangle, by default, text is broken at the last word boundary that is inside the rectangle's boundary and wrapped to the next line.

### **Flag = 8192**

Specifies that only entire lines are laid out in the layout rectangle. By default, layout continues until the end of the text or until no more lines are visible as a result of clipping, whichever comes first. The default settings allow the last line to be partially obscured by a layout rectangle that is not a whole

multiple of the line height. To ensure that only whole lines are seen, set this flag and be careful to provide a layout rectangle at least as tall as the height of one line.

**Flag = 16384**

Specifies that characters overhanging the layout rectangle and text extending outside the layout rectangle are allowed to show. By default, all overhanging characters and text that extends outside the layout rectangle are clipped. Any trailing spaces (spaces that are at the end of a line) that extend outside the layout rectangle are clipped. Therefore, the setting of this flag will have an effect on a string measurement if trailing spaces are being included in the measurement. If clipping is enabled, trailing spaces that extend outside the layout rectangle are not included in the measurement. If clipping is disabled, all trailing spaces are included in the measurement, regardless of whether they are outside the layout rectangle.

Multiple flags set can produce combined effects:

When both 2 and 1 are set, individual lines of text are drawn vertically. The first line starts at the right edge of the layout rectangle; the second line of text is to the left of the first line, and so on.

When 2 is set and 1 is not set, individual lines of text are drawn vertically. The first line starts at the left edge of the layout rectangle; the second line of text is to the right of the first line.

When 1 is set and 2 is not set, the individual lines of text are horizontal and the reading order is from right to left. This setting does not change the order in which characters are displayed, it simply specifies the order in which characters can be read.

The 2 and 1 flags can affect string alignment.

## Bitmaps

If you pass a bitmap to a Ruby component via a Bitmap connector then you can use this in the View class **drawBitmap** and **drawBitmapSection** methods to display it (or a section of it) on the View.

These methods are defined as follows:

**drawBitmap** bitmap, position [, alpha, rotation, origin]

**drawBitmapSection** bitmap, source, position [, alpha, rotation, origin]

The 'bitmap' parameter is the bitmap you want to display. In the case of **drawBitmapSection** the 'source' parameter is a 4 element array representing the rectangular section of the bitmap that you want to display. The rectangle array is defined by [x,y,w,h] where x,y define the top-left and w,h define the size of the section in pixels.

The 'position' parameter can either be a two element array representing the top-left corner of where you want the bitmap to be displayed or it can be a 4 element array representing the rectangle that you want to draw the bitmap into. The rectangle array is defined by [x,y,w,h] where x,y define the top-left and w,h define the size in grid squares.

If you supply a point for the position then the bitmap draws at full size. If you supply a rectangle the bitmap (or section of bitmap) will be stretched or reduced in size so as to fit exactly in the rectangle.

The 'alpha' parameter is optional. This defines the transparency level for the drawn bitmap. It is an integer value in the range 0-255 with 255 being fully opaque and 0 being fully transparent. If you omit this parameter a value of 255 is assumed.

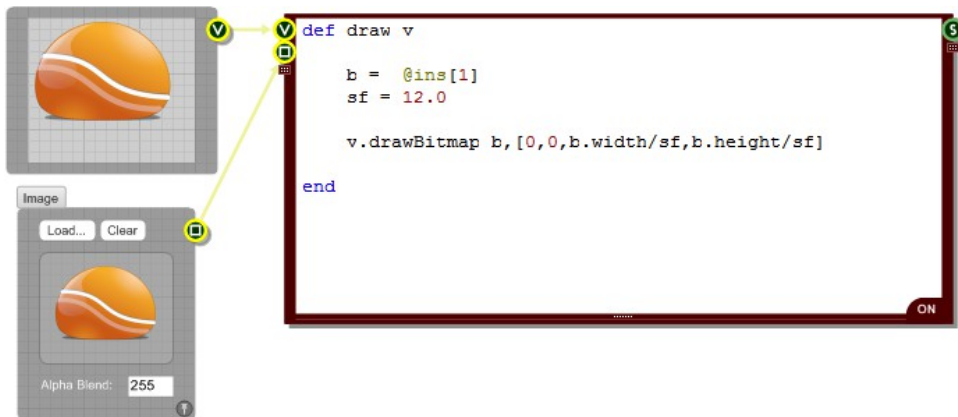
The 'rotation' parameter is also optional and allows you to rotate the bitmap through an angle, clockwise in degree. The origin of the rotation is assumed to be the centre point of the bitmap unless you supply an 'origin' parameter, in which case this point is used instead. The origin parameter is a two element array [x,y] where x and y are the coordinates of the rotation origin in grid squares.

The 'bitmap' parameter is an instance of the **Bitmap** class. There are four methods of this class for determining the bitmap size:

**width**  
**height**  
**widthPixels**  
**heightPixels**

The width and height methods give you the size of the bitmap in grid squares at the current default zoom level. The widthPixels and heightPixels methods give you the exact size of the bitmap in pixels.

Here's an example of a bitmap being drawn into a rectangle scaled down 12 times:



### Bitmap Rendering Options

If you are drawing a bitmap into a rectangle that is a different size then you can choose how to render that bitmap by setting the interpolation mode. This is done using the following View method:

**setInterpolationMode** mode

This takes either an integer (0-7) or one of the following strings:

"low", "high", "bilinear", "bicubic", "nearest", "hqbilinear", "hqbicubic"

You can also find out the current interpolation mode by using the following method. This will return one of the 8 strings shown above:

**getInterpolationMode** mode

## Redraw

If values changes in another method which affects what should be displayed by the draw method then you need to be able to request a redraw. This is very easy to do. All you do is call the **redraw** method.

If you have more than one View connected to your Ruby component then you should also provide the input connector reference of the View you want to redraw as an input parameter to the redraw method.

If only part of the display needs to be redrawn you can pass a rectangle after the connector reference So that's:

**redraw** connectorReference, redrawArea { both optional }

Where redrawArea is a Ruby array of the form [x,y,w,h].

## Clipping

Sometimes you need to be able to draw only within a portion of a View. This is where clipping comes into play. The View class has two methods for setting the clipping:

**setClip** region

**combineClip** region, combineMode

Both methods take a region as input. This can be a four element array defining a rectangle, the same format as the rectangles we used for drawing shapes. It can also be a collection of points defining a polygon. This is represented by an array of two element arrays, each of which defines an (x,y) position. You can also use a graphics path object to define the region.

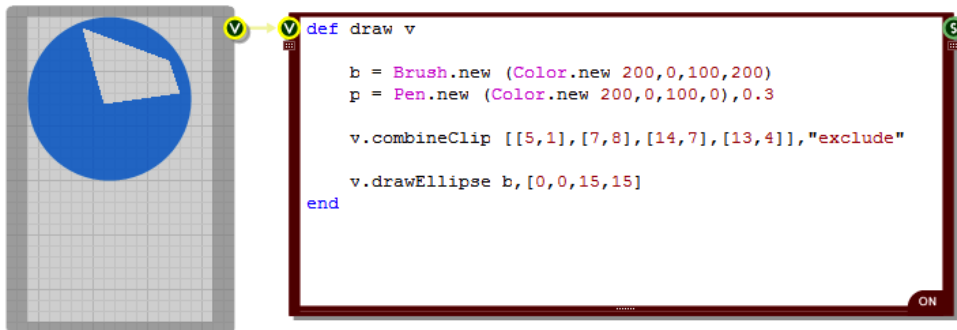
The **setClip** method sets the clipping region to the input region. This means that drawing will only occur within that region – anything outside is 'clipped' out.



The **combineClip** method combines the region with the current clipping region. The `combineMode` input is a string that specifies how the regions are to be combined. The options are:

- “**replace**” - use the new region
- “**intersect**” - use the intersection of both regions
- “**union**” - use both regions combined
- “**xor**” - use the union minus the intersection
- “**exclude**” - use the old region minus the new one
- “**complement**” - use the new region one minus the old one

Here's an example of how to use clipping:



There are another two methods relating to clipping:

**resetClip**  
**getClip**

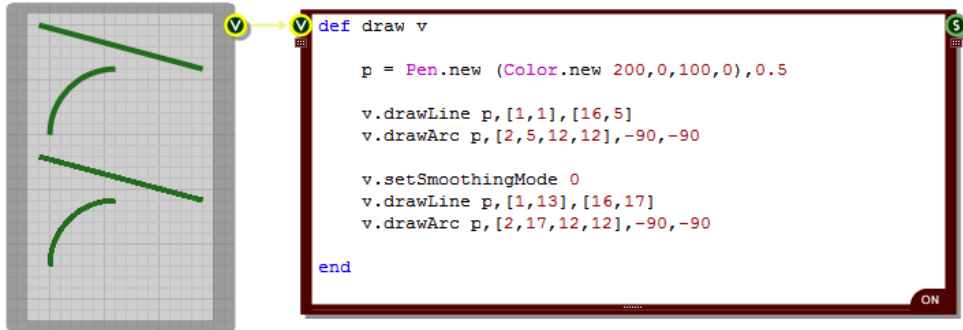
To reset the clipping region just call **resetClip** or you can also pass `nil` as the region for the `setClip` method.

The **getClip** method will return an array representing the bounding rectangle of the current clip region in the format `[x,y,width,height]`.

## Smoothing

By default all graphics are smoothed. The otherwise sharp jagged edges around graphics objects are antialiased to make them appear smooth.

If you're drawing straight lines you may want to turn this off as a side effect is that it does introduce a little blurring. The **setSmoothingMode** method of the `View` class allows you to control when smoothing is applied. You can pass `0` & `1` or “on” & “off”.



There is a similar method for text smoothing called **setTextRenderingHint**. This method has more input options:

- |                            |  |
|----------------------------|--|
| "off"                      | - no smoothing   |
| "singleBitPerPixelGridFit" | - use glyph bitmaps for each character with hinting                |
| "singleBitPerPixel"        | - use glyph bitmaps for each character with no hinting             |
| "antiAliasGridFit"         | - use antialiased glyph bitmaps for each character with hinting    |
| "antiAlias"                | - use antialiased glyph bitmaps for each character with no hinting |
| "clearTypeGridFit"         | - use ClearType glyph bitmaps for each character with hinting      |

## View Properties

There are a few more methods of the View class that we haven't mentioned yet. As well as the drawing methods we've already talked about there are four methods that give you some important properties of the View. These are as follows:

**gridStep**  
**defaultGridStep**  
**width**  
**height**

The **gridStep** method gives you the number of pixels per grid square for a view object. The **defaultGridStep** method gives you the default grid step ie. the number of pixels per grid square when the zoom level is set to normal. If you don't change the default grid step (under Zoom Level on the Schematic options dialog) then this will be 8 pixels per grid square.

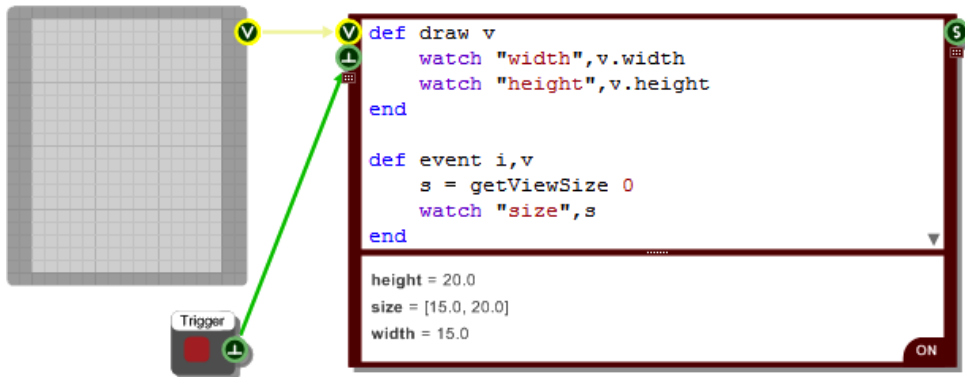
These two methods are not often used but they can be useful in special circumstance eg. for scaling bitmaps as you may want a bitmap to appear at normal size when at the default zoom level but scale accordingly when zoomed.

The **width** and **height** methods are much more commonly used. If you want your graphics to change based on the size of the front panel you're drawing to then you'll need to know the width and height. These are of course supplied in grid square units.

Sometimes you need to be able to get the view size when you're not in the Draw method. What do you do if you don't have access to a View object? Well we have a special method of the RubyEdit class that you can use:

**getViewSize** connectorReference

You need to pass the index of the View connector that you want to query (or you can leave it blank and zero will be assumed). The method returns a two element array containing the width and height. In the example below we've pressed the trigger button and the event method has been called triggering the call to **getViewSize** and the result is then displayed in the watch list.



If you don't have a View connected then the call to **getViewSize** will return nil. This can happen when you have no link connected or during loading or init so if you intend to use **getViewSize** in loading or init methods it's best to check whether the result is nil before acting on it.

As well as a size, a view can also have a position if it's displayed on a parent front panel. You can find out this position using the **getViewPos** method:

**getViewPos** connectorReference

Once again you need to pass the index of the View connector that you want to query (or you can leave it blank and zero will be assumed). The method returns a two element array containing the X and Y coordinates of the View in its parent front panel.

## Changing The View

So far we've been working with a View that has a fixed size and position. But what if you wanted to change that? What if you wanted to expand the view, move its position in the parent front panel or even hide it?

Well, you can do that using the **setViewSize** and **setShowInParent** methods.

The **setViewSize** method takes an array of the form [x,y,width,height] (you can also supply an initial parameter to define the index of the input connector you want to connect through).

The x,y part defines the location of the top-left corner of the view in its parent panel and the width and height parameters define the size.

The **setShowInParent** method takes a boolean parameter which specifies whether the view is to be shown in the parent panel or not (once again, you can also supply an initial parameter to define the index of the input connector you want to connect through).

# Advanced Brushes

The Drawing section covered how to create a solid brush and use it to fill basic shapes or draw text. In this section we'll look at more advanced brush options.

## Linear Gradients

You can create filled areas that flow smoothly between colours using the **LinearGradientBrush** class. This is a variation on the standard Brush class and can be used as an instrument for drawing in all the shape drawing methods just like the standard Brush.

In its simplest form a LinearGradientBrush allows you to define a gradient between two colours. You create an instance of the class as follows:

```
b = LinearGradientBrush.new boundRect, color1, color2, angle, scaleTrans
```

*boundRect*

This is a rectangle ( once again defined as a ruby array [x,y,width,height] ) that defines the boundary of the gradient fill. The gradient will fill this area so you need to make sure that you create a bounding rectangle that fits with whatever shapes you intend to fill.

*color1, color2*

These are the end colours for the gradient and should be instances of the Color class. If you don't supply an angle parameter then color1 will be at the left-hand edge of the rectangle and color2 will be at the right-hand edge.

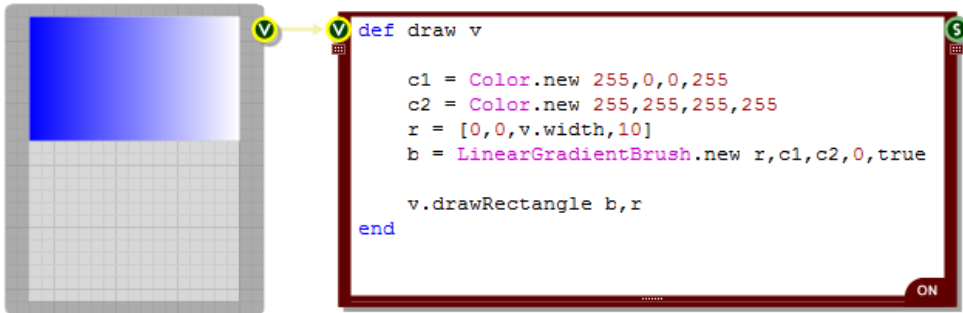
*angle*

This is angle of rotation for the gradient in degrees in the clockwise direction. If the angle is 90 degrees for example, color1 will be at the top of the rectangle and the gradient will run downwards to color2.

*scaleTrans*

This defines whether the angle is affected by the transform associated with the brush. The parameter is a boolean flag so it should be set to **true** or **false**.

Here's a simple example of a LinearGradientBrush:



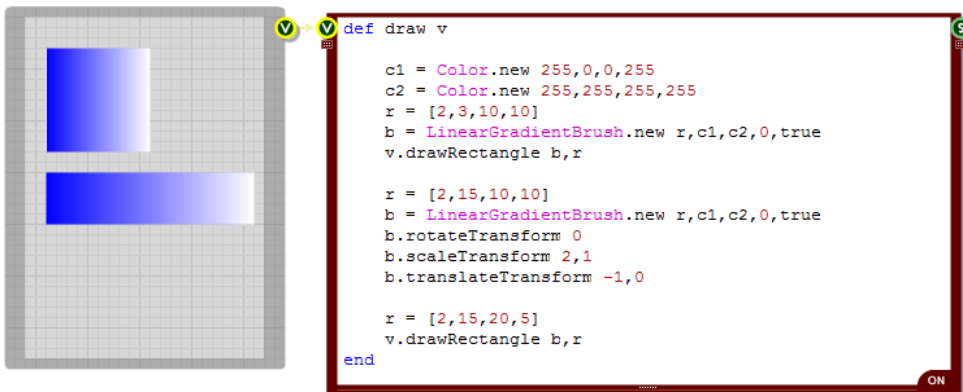
### Transformations

You can rotate, translate or scale the brush after it has been created. To do this you use the following methods:

- rotateTransform** angle
- translateTransform** x, y
- scaleTransform** sf-x, sf-y
- resetTransform**

The rotation method takes an angle in Degrees. The translate method takes x and y offsets. The scale method takes x and y scale factors – these are floats with 1.0 meaning no change. The resetTransform method does exactly what it says, it resets any applied transforms.

The example below shows a gradient that has been stretched and translated.



## Blending

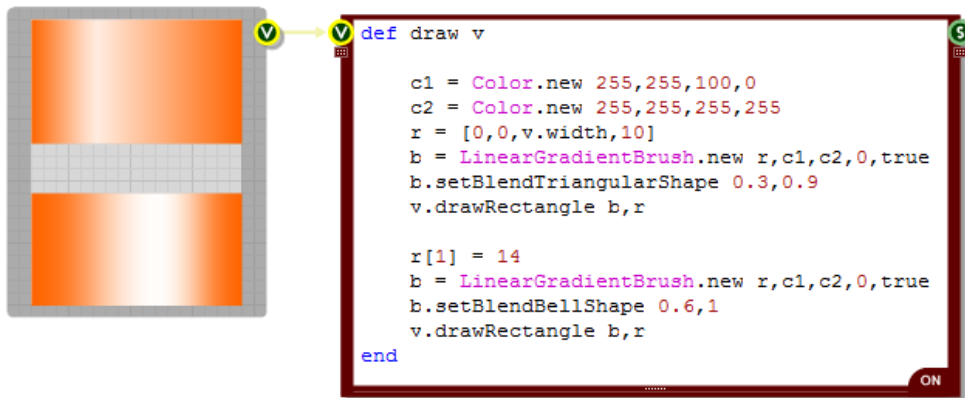
You don't just have to have a straight gradient. Using the **setBlendTriangularShape** and the **setBlendBellShape** methods you can move the end colour to some in between point and blend out on either side. The bell option uses a non-linear blending curve.

**setBlendTriangularShape** focus, scale

**setBlendBellShape** focus, scale

Both methods take two inputs. The first is the **focus**. This is a float between 0 and 1 which defines the relative distance of the end point colour from the start point. A value of 0.5 would put it bang in the middle for example. The second parameter is the **scale**. This is also a float between 0 and 1 and it defines the intensity of the colour at the focus point.

Here's an example of these in action:

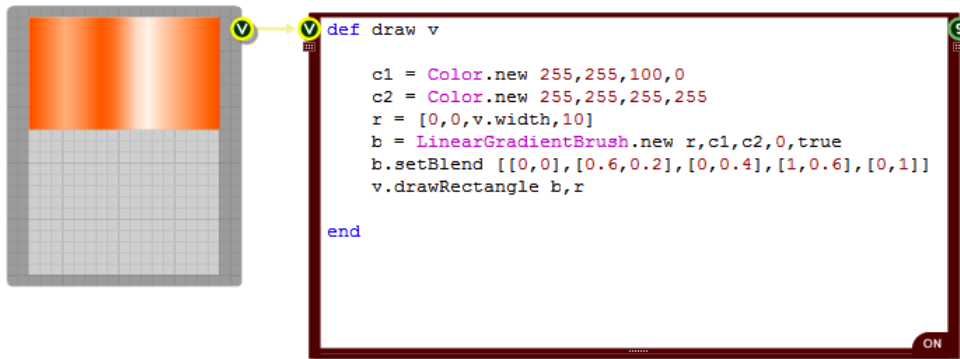


You take this a step further and set your own blend points. This is done using the **setBlend** method.

**setBlend** blendArray

The method takes an array of two element arrays. Each two element array defines first a blend factor then a position along the gradient. Both values are floats in the range 0 to 1.

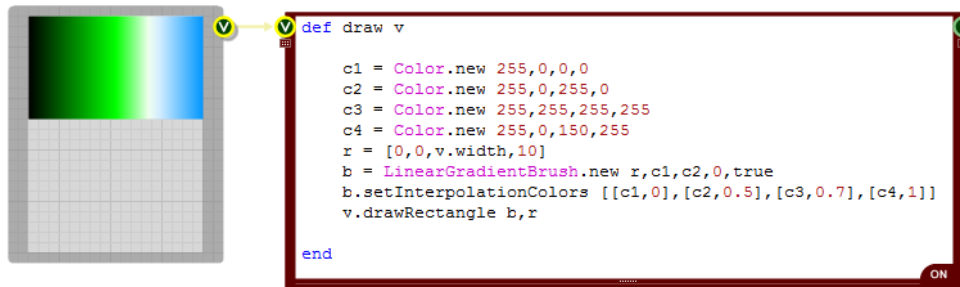
This is all best illustrated with an example. The picture below shows a gradient with three blend points in between the end points which themselves are set to zero (ie. color1):



You can also introduce a whole range of colours into the gradient using the **setInterpolationColors** method. This is very similar to the setBlend method except that instead of a blend factor you have a Color object paired with the position along the gradient.

**setInterpolationColors** colourArray

The example below shows 4 colours merging together at various points:



## Wrapping

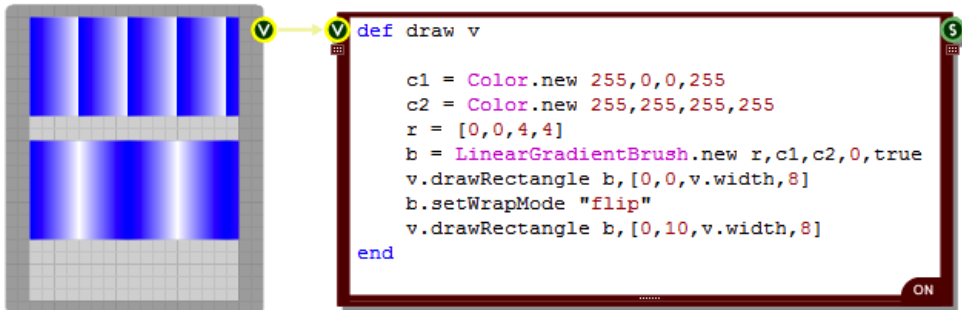
The LinearGradientBrush has a defined bounding rectangle. This makes the brush into a tile. The tile is then used repeatedly to pave any area that is painted with the brush.

Depending on how you define your gradient you may want to flip the tiles as they are placed so that you don't get hard edges between them. You can do this by setting the wrap mode to 'flip' using the setWrapMode method.

**setWrapMode** mode



The default mode is to tile without flipping but by passing the string "Flip" or the number 1 you can set this to flipped. Here's an example of how the wrap mode affects the overall result. You can see that in the bottom rectangle each tile is flipped so that the gradient runs smoothly from one tile into another.



## Path Gradients

Path gradients provide another way of producing smooth transitional fills between different colours. Unlike the LinearGradientBrush they create a radial gradient from a central colour out towards a set of surrounding colours.

The surrounding area is defined by a path and colours can be distributed along the path. A single central colour is also defined and the colour gradient is calculated by blending outwards from the central point.

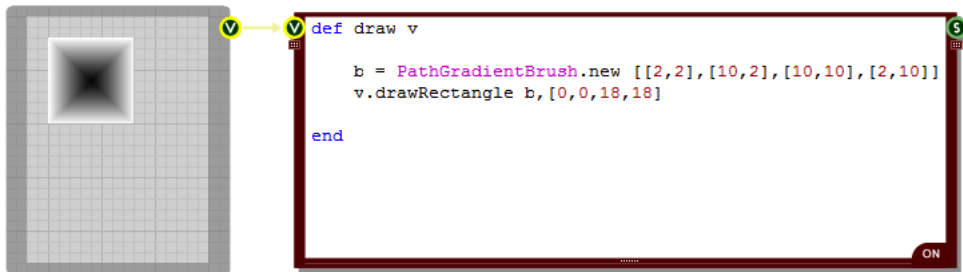
In its simplest form a PathGradientBrush allows you to define a gradient between two colours. You create an instance of the class as follows:

```
b = PathGradientBrush.new path
```

There is a single input parameter, the path. The path can either be an array of points (represented as an array of two element arrays containing the x and y coordinates of each point) or it can be a GraphicsPath object.

The default colours are Black at the centre and White on the path.

Here's an example using a square path and the default colours:



### Changing Colours

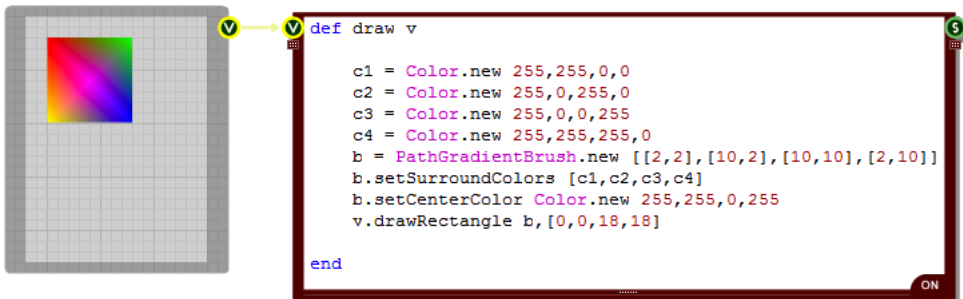
There are two methods of the PathGradientBrush class that allow you to change the colours.

**setCenterColor** colour

**setSurroundColours** colourArray

The **setCenterColor** method takes a single colour object. The **setSurroundColours** method takes an array of colours. By default these colours are allocated in turn to each point in the path. If there are fewer colours than path points and so we run out of colours when allocating them to points on the path, the last colour in the array is used on all subsequent points.

The example below shows what we get when changing the centre colour to pink and the outer colours to red, green, blue and yellow.



## Wrapping

In a similar way to the LinearGradientBrush a PathGradientBrush has a bounding rectangle which contains the whole path. This makes the brush into a tile. The tile can then be used repeatedly to pave any area that is painted with the brush.

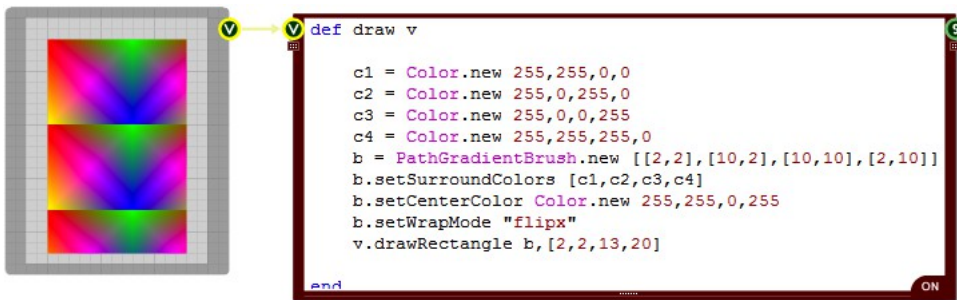
Unlike the LinearGradientBrush, the default for the PathGradientBrush is to not tile or clamp. You can change this by setting the wrap mode using the setWrapMode method.

### setWrapMode mode

The various mode options allow you to flip the tiles as they are placed so that the tiles blend into one another. Here are the options for the mode input value:

0 or "tile"	Normal tiling is performed
1 or "flipx"	Tiling is performed with alternate tiles flipped in the x direction
2 or "flipy"	Tiling is performed with alternate tiles flipped in the y direction
3 or "flipxy"	Tiling is performed with alternate tiles flipped in the x and y directions
4 or "clamp" or "notile"	No tiling is performed – this is the default option

The example below shows the "flipx" wrap mode in action. You can see how the tiles in the x direction are alternately flipped but the tiles in the vertical direction are not.



## Blending

The PathGradientBrush supports blending just like the LinearGradientBrush. Using the **setBlendTriangularShape** and the **setBlendBellShape** methods you can move the centre colour to some in between point and blend out on either side. The bell option uses a non-linear blending curve.

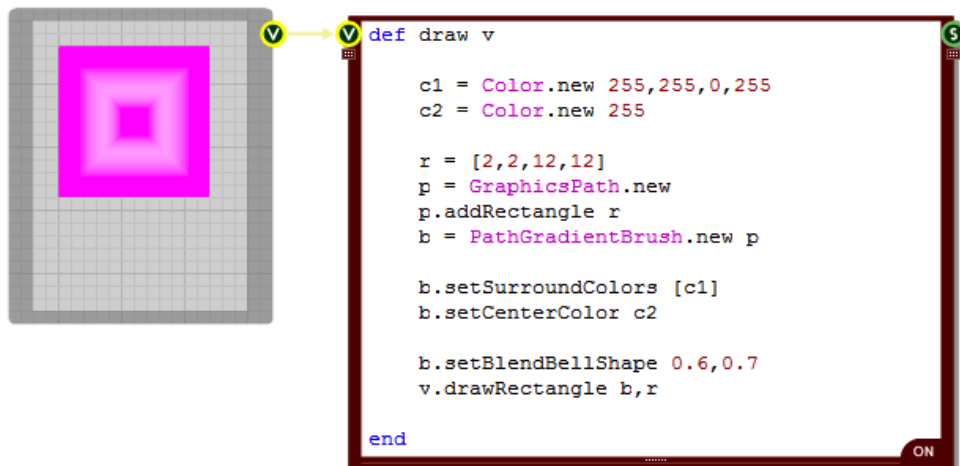
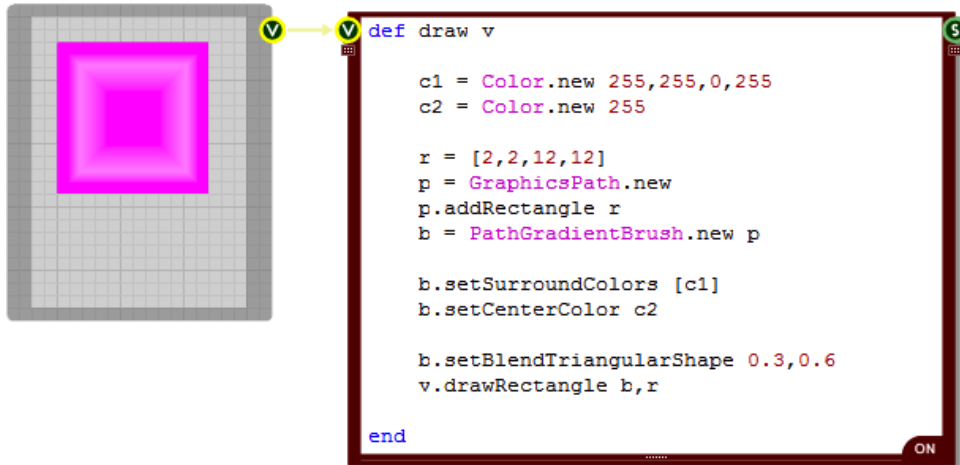
**setBlendTriangularShape** focus, scale

**setBlendBellShape** focus, scale

Both methods take two inputs. The first is the **focus**. This is a float between 0 and 1 which defines the relative distance of the end point colour from the start point. A value of 0.5 would put it bang in the

middle for example. The second parameter is the **scale**. This is also a float between 0 and 1 and it defines the intensity of the colour at the focus point.

Here's an example of these in action. We've used a GraphicsPath instead of an array of points this time too just to show how that works when defining the brush.

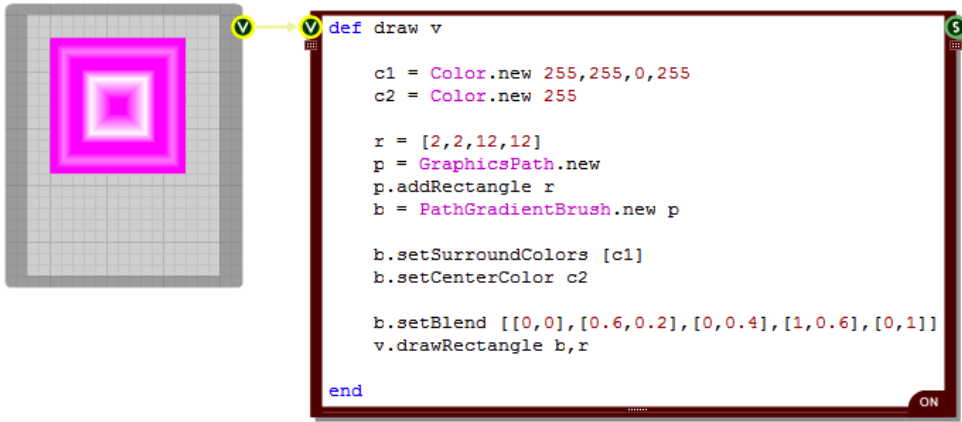


You take this a step further and set your own blend points. This is done using the **setBlend** method.

**setBlend** blendArray

The method takes an array of two element arrays. Each two element array defines first a blend factor then a position along the gradient. Both values are floats in the range 0 to 1.

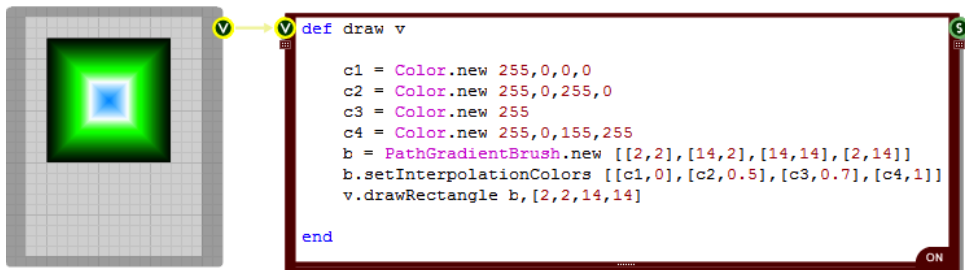
This is all best illustrated with an example. The picture below shows a gradient with three blend points in between the end points which themselves are set to zero (ie. color1):



You can also introduce a whole range of colours into the gradient using the **setInterpolationColors** method. This is very similar to the setBlend method except that instead of a blend factor you have a Color object paired with the position along the gradient.

**setInterpolationColors** colourArray

The example below shows 4 colours merging together at various points:

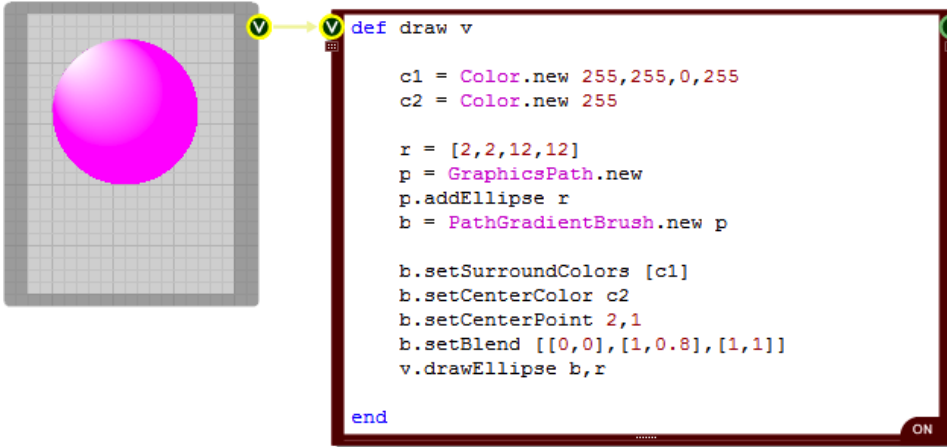


### Centre Point

By default the centre point is calculated for you based on the path. However, you can set the position of the centre point yourself using the **setCenterPoint** method.

**setCenterPoint** x, y

All you need to do is pass the x and y coordinates of the point you want to have as the new centre. The example below shows a displaced centre point being used to create a 3D effect by creating the illusion of reflected light from a ball.



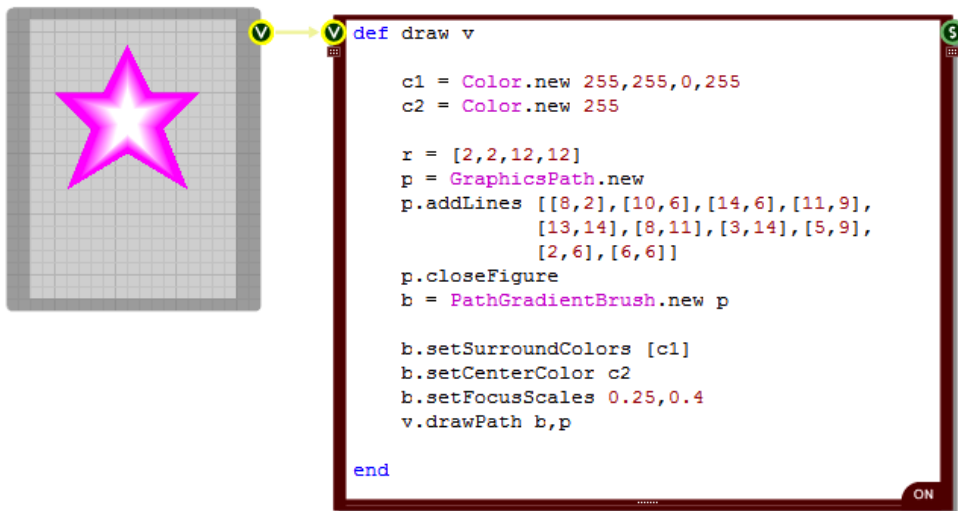
## Focus Scales

By default the centre colour is focused at the centre point. However, you can change the focus size around this point by using the `setFocusScales` method.

**setFocusScales** scaleX, scaleY

The method takes two scale factors. These are applied to the path that defines the brush and the resulting shape defines the area that is filled with the centre colour. Scale factors of 1.0 will make the whole shape the focus colour so most of the time the scale factors you use will be less than 1.0.

Here's an example that shows how this works. Here we have increased the size of the focus point and stretched it a bit more vertically than horizontally.



## Transformations

You can rotate, translate or scale a PathGradientBrush after it has been created. To do this you use the following methods:

**rotateTransform** angle

**translateTransform** x, y

**scaleTransform** sf-x, sf-y

**resetTransform**

The rotation method takes an angle in Degrees. The translate method takes x and y offsets. The scale method takes x and y scale factors – these are floats with 1.0 meaning no change. The resetTransform method does exactly what it says, it resets any applied transforms.

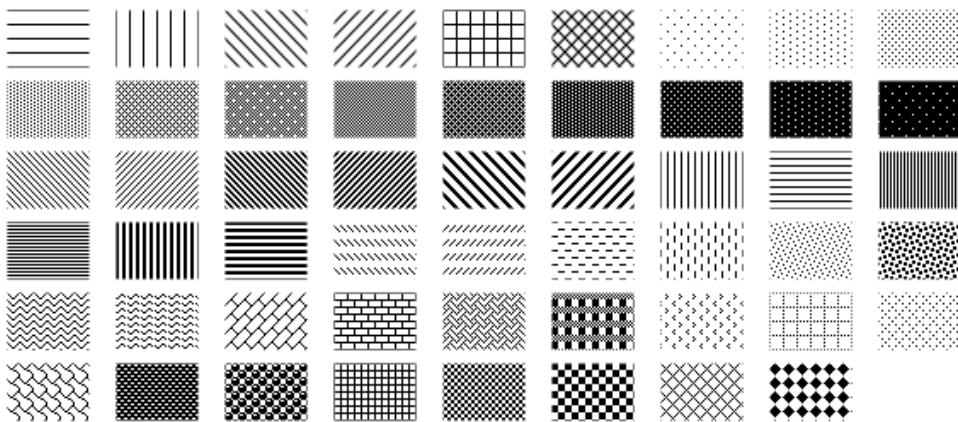
## Hatch Brushes

A HatchBrush is used for filling an area with a hatched or stipple effect. It is defined by a hatch style, a foreground colour and an optional background colour. You can create an instance as follows:

```
b = HatchBrush.new style, foregroundCol, backgroundCol
```

If you don't supply a background colour then the background will be left transparent and all you will get is the hatching. For the style you have 53 different options. This can be specified as an integer (in the range 0-52) or a text string.

The hatch styles are shown below (in increasing order left to right, top to bottom):



The hatch style strings are as follows:

"HORIZONTAL"

"VERTICAL"

"FORWARDDIAGONAL"

"BACKWARDDIAGONAL"

"CROSS"

"DIAGONALCROSS"

"05PERCENT"

"10PERCENT"

"20PERCENT"

"25PERCENT"

"30PERCENT"

"40PERCENT"

"50PERCENT"

"60PERCENT"

"70PERCENT"

"75PERCENT"

"80PERCENT"

"90PERCENT"

"LIGHTDOWNWARDDIAGONAL"

"LIGHTUPWARDDIAGONAL"

"DARKDOWNWARDDIAGONAL"

"DARKUPWARDDIAGONAL"

"WIDEDOWNWARDDIAGONAL"

"WIDEUPWARDDIAGONAL"

"LIGHTVERTICAL"

"LIGHTHORIZONTAL"

"NARROWVERTICAL"

"NARROWHORIZONTAL"

"DARKVERTICAL"

"DARKHORIZONTAL"

"DASHEDDOWNWARDDIAGONAL"

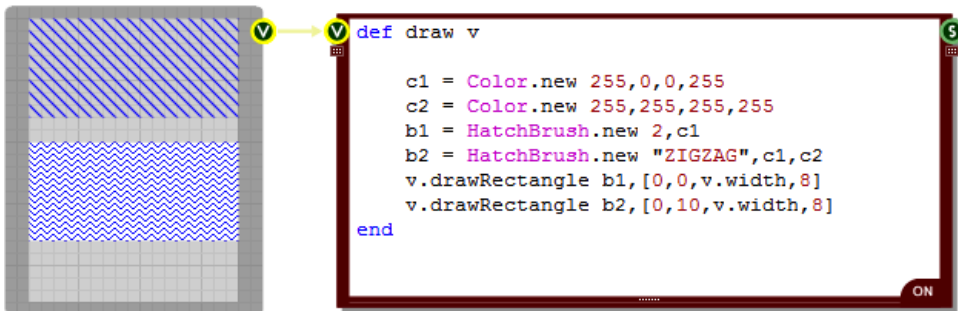
"DASHEDUPWARDDIAGONAL"



"DASHEDHORIZONTAL"  
 "DASHEDVERTICAL"  
 "SMALLCONFETTI"  
 "LARGECONFETTI"  
 "ZIGZAG"  
 "WAVE"  
 "DIAGONALBRICK"  
 "HORIZONTALBRICK"  
 "WEAVE"  
 "PLAID"  
 "DIVOT"

"DOTTEDGRID"  
 "DOTTEDIAMOND"  
 "SHINGLE"  
 "TRELLIS"  
 "SPHERE"  
 "SMALLGRID"  
 "SMALLCHECKERBOARD"  
 "LARGECHECKERBOARD"  
 "OUTLINEDIAMOND"  
 "SOLIDIAMOND"

The following example shows how to create and use a HatchBrush. The top rectangle uses a brush created using just a foreground colour. You can see the background underneath as only the hatching is applied. The bottom rectangle has been defined with a white background colour. We've also used the two different ways of specifying the hatch style in each rectangle – by index or by name.



## Texture Brushes

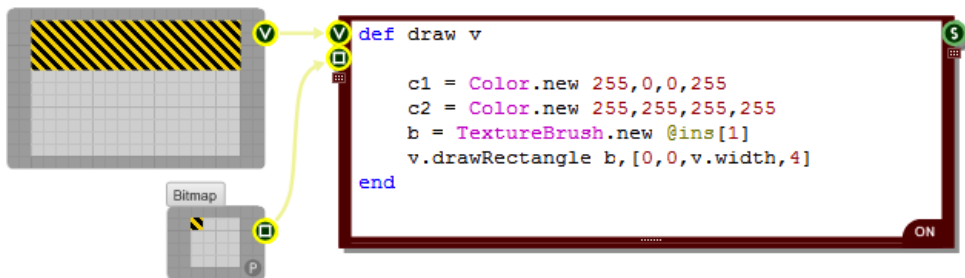
A TextureBrush is defined by a bitmap. When you fill an area with a TextureBrush the bitmap is used repeatedly to tile the area you want to fill.

You can create an instance as follows:

```
b = TextureBrush.new bitmap
```

The bitmap input parameter needs to be a Bitmap object. You need to pass a Bitmap object into your Ruby component so that you can use it in your TextureBrush.

The example below shows how a TextureBrush is created and used.



### Wrapping

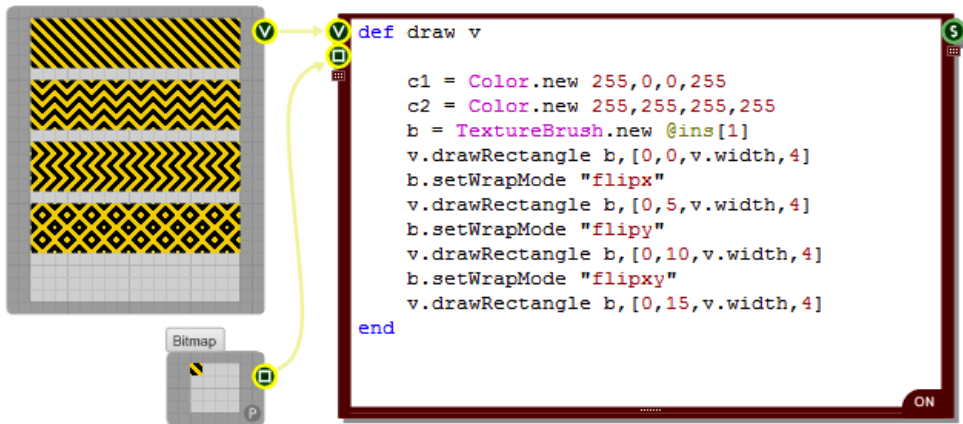
When filling an area TextureBrush tiles are placed next to each other. However, you could choose to flip the tiles alternately as you pave vertically or horizontally to create a more fluid pattern. To do this you need to use the `setWrapMode` method.

**setWrapMode** mode

Here are the options for the mode input value:

- |               |  |
|---------------|--|
| 0 or "tile"   | Normal tiling is performed – this is the default option                    |
| 1 or "flipx"  | Tiling is performed with alternate tiles flipped in the x direction        |
| 2 or "flipy"  | Tiling is performed with alternate tiles flipped in the y direction        |
| 3 or "flipxy" | Tiling is performed with alternate tiles flipped in the x and y directions |

The example below shows the four different wrap modes in action.



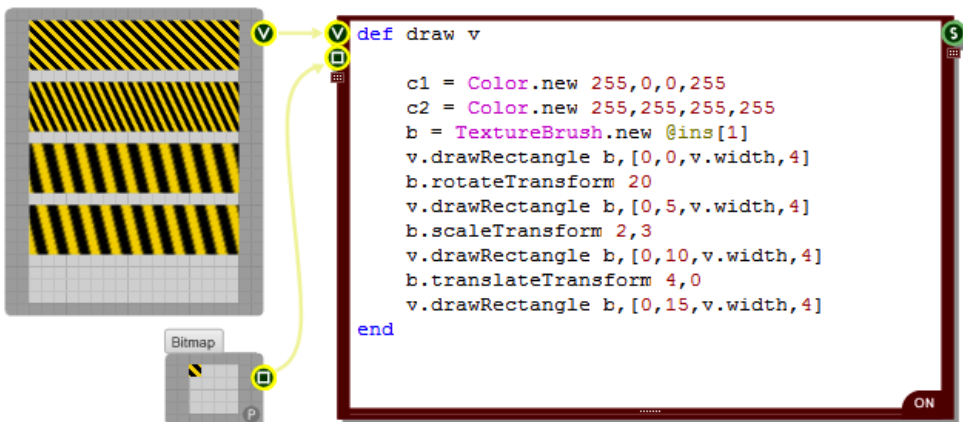
## Transformations

You can rotate, translate or scale a TextureBrush after it has been created. To do this you use the following methods:

- rotateTransform** angle
- translateTransform** x, y
- scaleTransform** sf-x, sf-y
- resetTransform**

The rotation method takes an angle in Degrees. The translate method takes x and y offsets. The scale method takes x and y scale factors – these are floats with 1.0 meaning no change. The resetTransform method does exactly what it says, it resets any applied transforms.

Here's an example showing how these all work:



# Advanced Pens

The Drawing section we covered the very basics of Pens. In this section we'll look at Pens in more detail and show you some of the more advanced things that you can do with them.

## Pen Alignment

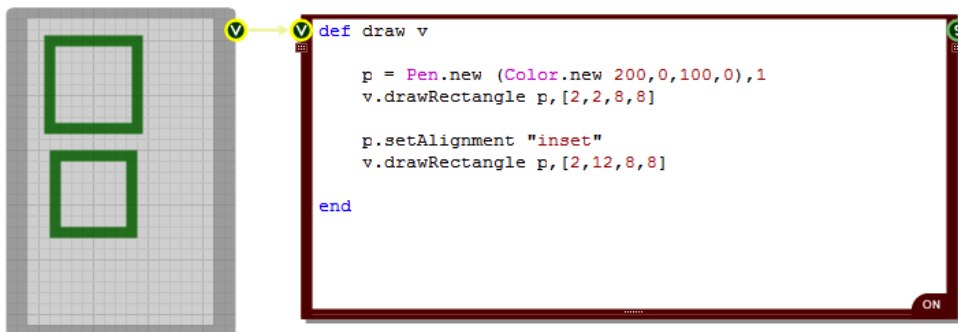
By default lines drawn with a pen are centred. This means that if you have a thicker pen then part of the pen will fall on either side of the line. Sometimes this is not the behaviour you want so you can change it using the **setAlignment** method for the Pen.

**setAlignment** alignmentType

There are just 2 options for the alignment type:

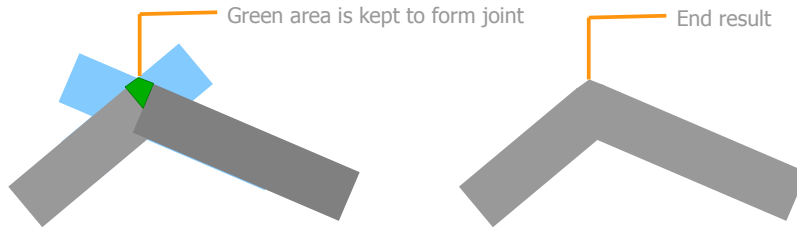
- |               |  |
|---------------|--|
| 0 or "center" | The pen is centred on the line                             |
| 1 or "inset"  | The pen is inset so its outer edge always follows the line |

The example below shows how the two types work in practice. The two rectangles are exactly the same size. However, the top rectangle is drawn with the alignment centred. You can see how this makes the rectangle appear larger.



## Line Joins

When you draw any shape that contains straight lines the joints between the lines are mitred together by default. A mitre joint is created between two lines by extending both lines artificially until they cross each other and then cutting off the protruding edges.



In the diagram above the blue lines show the artificial extensions. The green area is kept to form the joint and the blue protruding parts are removed. The end result is shown on the right.

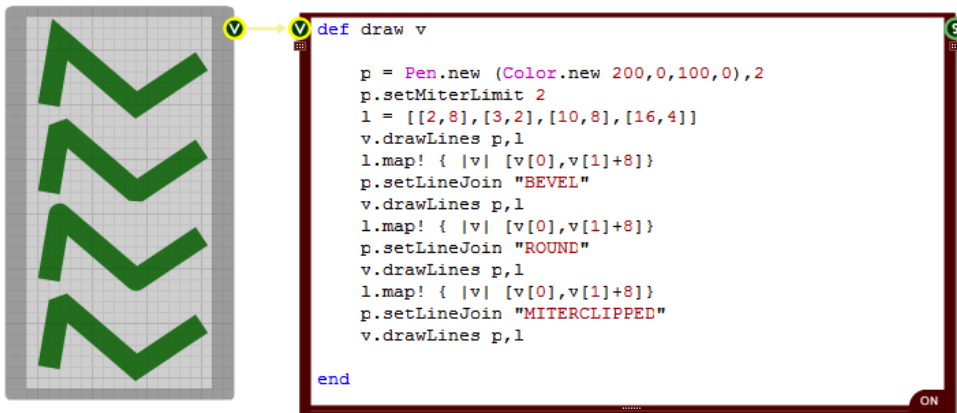
You can change the way in which lines are joined using the `setLineJoin` method of the `Pen` object.

`setLineJoin` type

The method takes a single input. This can be either an index or a string:

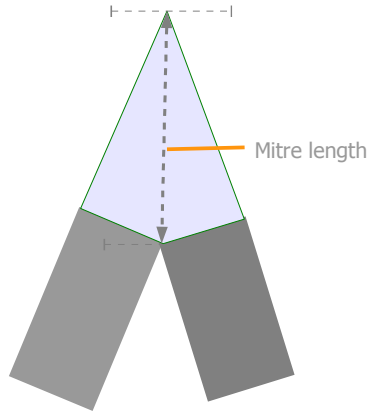
- |                     |  |
|---------------------|--|
| 0 or "mitre"        | A standard mitre joint – this is the default option                      |
| 1 or "bevel"        | The joint is flattened off instead of creating a point at the outer edge |
| 2 or "round"        | A curved edge is used to join the lines                                  |
| 3 or "mitreclipped" | A mitre joint is used if under the mitre limit otherwise a bevel is used |

The best way to understand these is to see them in action:



The bottom line shows the "mitreclipped" option. Notice that the first joint is beveled but the others are mitred. This is because for the first joint a mitre would extend beyond the mitre limit.

The mitre limit is the maximum allowed limit of mitre length to pen width. The mitre length is measured as shown in the diagram below.



The default mitre limit for a pen is 10.0 but you can change this using the **setMitreLimit** method.

**setMitreLimit** limit

If the line join is set to the default "mitre" option then any mitre joins above the mitre limit will be clipped at the mitre limit. However, if the line join is set to "mitreclipped" then any mitre joins above the mitre limit will be converted to bevels.

## Dashes

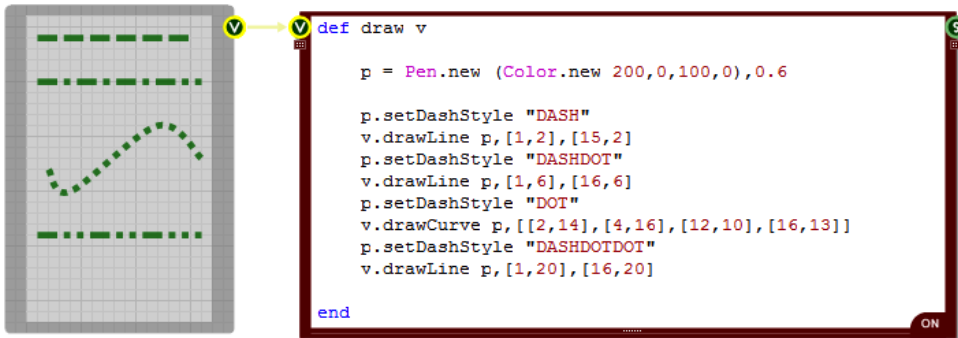
Pens can be set to draw broken rather than solid lines. You can do this using the **setDashStyle** method of the **Pen** object.

**setDashStyle** type

The method takes a single input. This can be either an index or a string:

0 or "none"	No dash is applied, instead you get the default solid line
1 or "dash"	A dashed line
2 or "dot"	A dotted line
3 or "dashdot"	A line comprised of alternating dashes and dots
4 or "dashdotdot"	A line comprised of alternating dashes and double dots

The example below shows a selection of these styles.



### Dash Caps

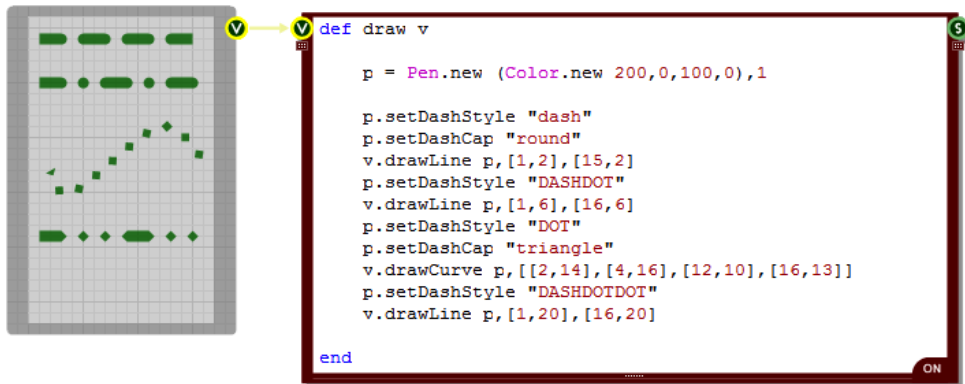
You can set the way that ends of the dashes and dots are drawn by using the **setDashCaps** method.

**setDashCaps** type

There are only 3 options for the type:

0, 1 or "flat"	The ends are flat, dots are squares – this is the default option
2 or "round"	The ends are rounded, dots are circles
2 or "triangle"	The ends are pointed, dots are diamonds

Here's an example showing how the round and triangle options work:

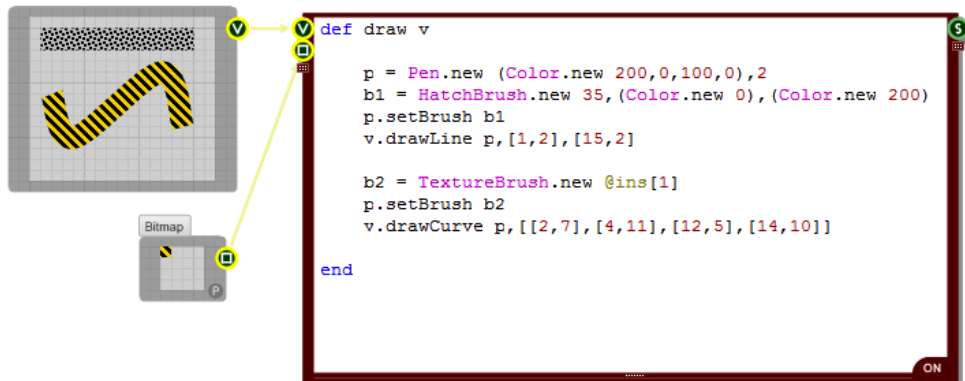


## Brushed Lines

We've seen how it's possible to fill a shape using a brush. Well you can also use a brush to fill a line drawn with a pen. To do this use the **setBrush** method of the Pen object.

**setBrush** brush

The input is a brush object. You can use any kind of brush, gradient, hatched, texture, path or solid. The example below shows two lines being drawn with hatch and texture brushes.





## Line Caps

Line caps define how the end points of a line are drawn. By default lines are squared off at the ends but you may want a rounded or triangular end to your lines and line caps allow you to do this.

You can also have a larger shape at the start or end of a line. This is called an Anchor. The most familiar line cap anchor is the arrow head but there are several more standard ones and you can also make your own.

Line caps are properties of **Pen** objects. To set the line caps for a pen there are two Pen class methods, one for setting the cap at the start of the line and one for setting the one at the end:

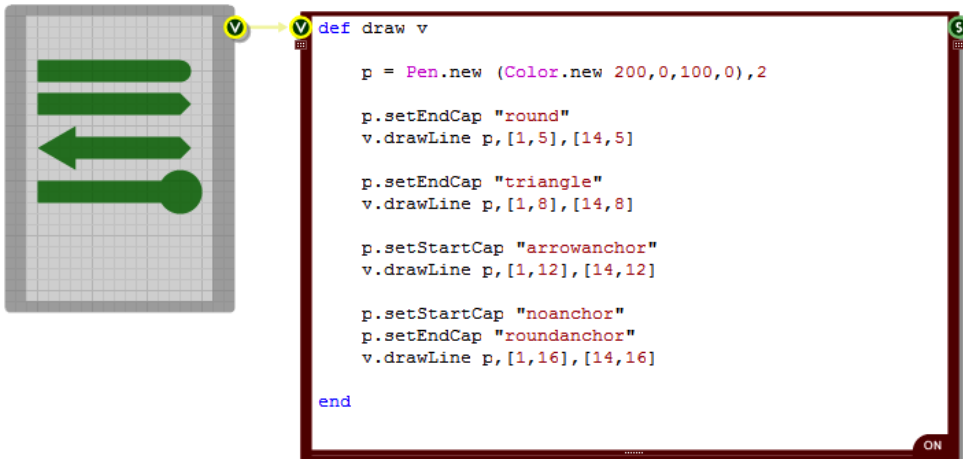
**setStartCap** type

**setEndCap** type

The type parameter can be any of the following strings:

“square”, “round”, “triangle”, “noanchor”, “squareanchor”, “roundanchor”,  
“diamondanchor”, “arrowanchor”

Here is an example that shows how to use line caps:



## Scaling

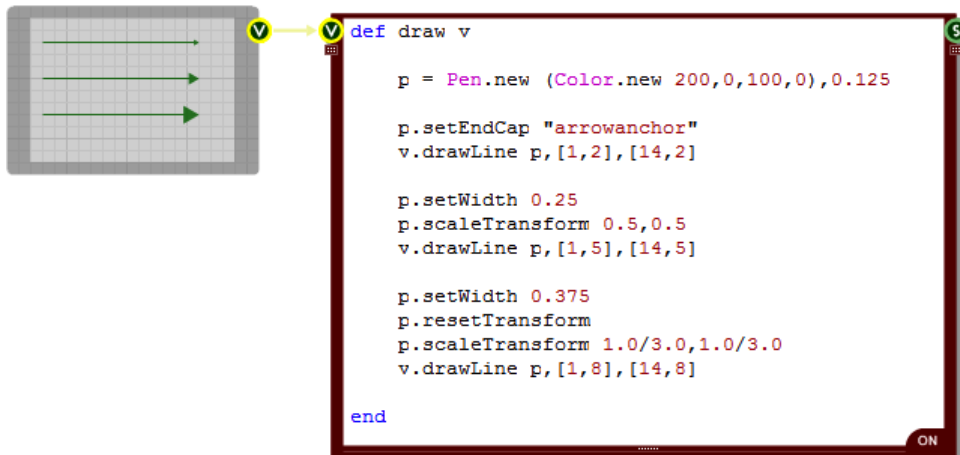
Line caps are scaled automatically in proportion to the line thickness. This scaling can make it difficult to achieve the style you're looking for. For example, if you create a single pixel thickness line with an arrow end cap the arrow will be just 2 pixels wide.

You can get round this by creating a thicker pen and then scaling it down using the **scaleTransform** method. The end cap is inversely affected by the scaleTransform and so if you find a scale factor that maintains the .

**scaleTransform** sf-x, sf-y  
**resetTransform**

The scaleTransform method takes x and y scale factors – these are floats with 1.0 meaning no change. There is also a resetTransform method which is useful if you're reusing the same pen and resets any applied transforms.

The example below shows a single pixel width line with an arrow end cap. Below it are two identical lines but with larger arrows using the technique we just described.



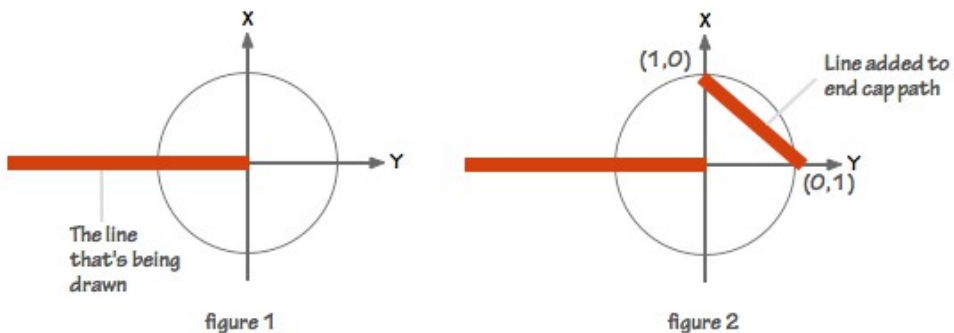
## Custom Line Caps

If you want to create your own custom line caps then you can do this using the **CustomLineCap** class. You create an instance of this class then use it to set the start or end cap for the pen.

Before you can do any of this you will need to create a **GraphicsPath** object. This will define the shape of the line cap.

You create a graphics path in the same way as usual. The units are still grid squares, however the coordinate system is relative to the line. This is necessary because when the line changes direction you want the end cap to change its orientation accordingly.

The diagram below shows how the coordinate system works. Figure 1 shows the axes relative to an arbitrary line. Figure 2 shows a single line having been added to the graphics path for the custom end cap. The line starts at (1,0) and ends at (0,1). If we added another line from (0,1) to (-1,0) and closed the path then we'd have a path to make our own custom arrow head line cap.

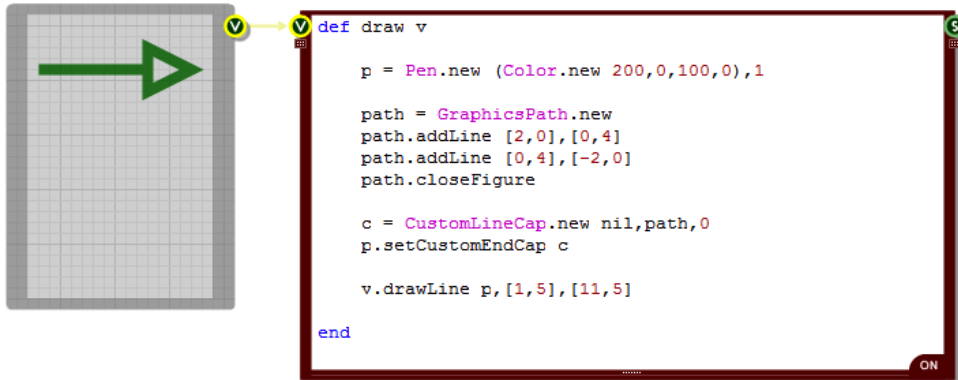


So now we go back to the **CustomLineCap** class. We can create an instance of this class in one of two ways. Either we have an end cap where the path shows as an outline or we have a path which is filled (with the pen color). Here's how you handle both cases:

```
c = CustomLineCap.new nil,path,0 # line cap outline
c = CustomLineCap.new path,nil,0 # line cap filled
```

The third parameter is the offset of the cap from the line in grid squares. A positive offset will move the cap away from the line and a negative one will pull it closer.

Now that you have a **CustomLineCap** object you can use the **setCustomStartCap** and **setCustomEndCap** methods of the **Pen** class to assign the cap to the **Pen**. The example below shows how this all works.



### Scaling

The custom line cap scales proportionally to the pen size. The default is for this to scale 1:1. You can change this by using the **setWidthScale** method of the CustomLineCap class. Setting this to 2 for example, would make the cap be double the size it would be normally at whatever pen width. The width of the pen used to stroke the path would also be double the pen width.

### Join and Caps for the Path

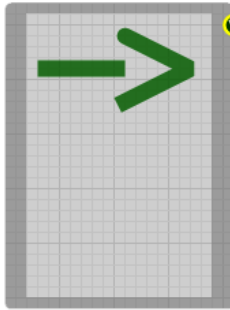
Going into even more detail, you can even set the end caps for the custom line cap itself. Use the **setStrokeCaps** method of CustomLineCap. This takes two parameters, a cap for the start of the path and a cap for the end of the path. These are strings and can be one of the following:

“square”, “round”, “triangle”

The join style can also be set by using the **setStrokeJoin** method. This takes a single string to define the join style and can be one of:

“bevel”, “round”, “miterclipped”

The example below shows these methods being used. Note that the end cap and join styles only apply to stroked paths. If you use a filled path for your custom end cap then these are ignored.



```
def draw v

  p = Pen.new (Color.new 200,0,100,0),1.5

  path = GraphicsPath.new
  path.addLine [2,0],[0,4]
  path.addLine [0,4],[-2,0]

  c = CustomLineCap.new nil,path,0
  c.setStrokeCaps "round","square"
  c.setStrokeJoin "bevel"
  p.setCustomEndCap c

  v.drawLine p,[1,5],[9,5]

end
```



ON

# Interaction

In the previous section we saw how you can draw to a front panel using a Ruby Component. In this section we'll talk about handling mouse events from a front panel.

You'll need a View input set up in exactly the same way as for the drawing. Alternatively, if you just want to handle mouse event input you can create a Mouse connector instead of the View. The way you handle events within the Ruby component is exactly the same in both cases.

## Overview

You will recall from the previous section that for drawing FlowStone looked for a **draw** method in the Ruby component. Mouse events are handled in a very similar way. For each type of event FlowStone looks for a particular method in your Ruby component. If it finds it then the details of the event are passed to that method so that it can respond.

There are ten methods that FlowStone looks for. They all take the same set of input parameters. There is an optional first parameter which is the input connector reference (an integer or label string). You only need this if you have multiple Views connected to your Ruby component. In the vast majority of cases this will not be so and so you don't need to worry about the connector reference parameter.

The main parameters are the x and y coordinates. These are both Floats and are in grid square units.

For example, to define the method to handle left mouse clicks:

```
def mouseLDown x,y
end
```

If you had more than one View or Mouse connector then you can add the connector reference so that you can differentiate between arriving events:

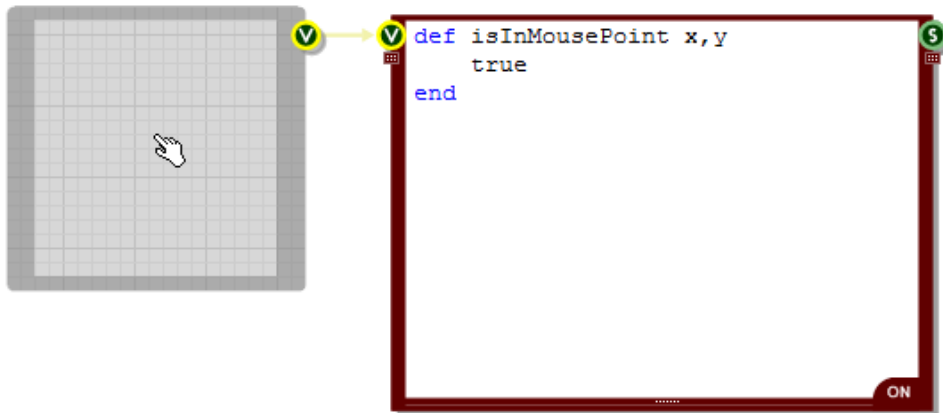
```
def mouseLDown i,x,y
end
```

We'll now look at all 10 methods in detail.

## Handling Mouse Events

The most important method you need to define is **isInMousePoint** because this is what tells FlowStone that you accept mouse events. Without this you'll get no mouse events whatsoever.

The method needs to return either true or false to tell FlowStone whether it handles mouse events or not (you can also return 1 or 0).



The example above shows how the mouse cursor changes to the default 'hand' to indicate that interaction is possible.

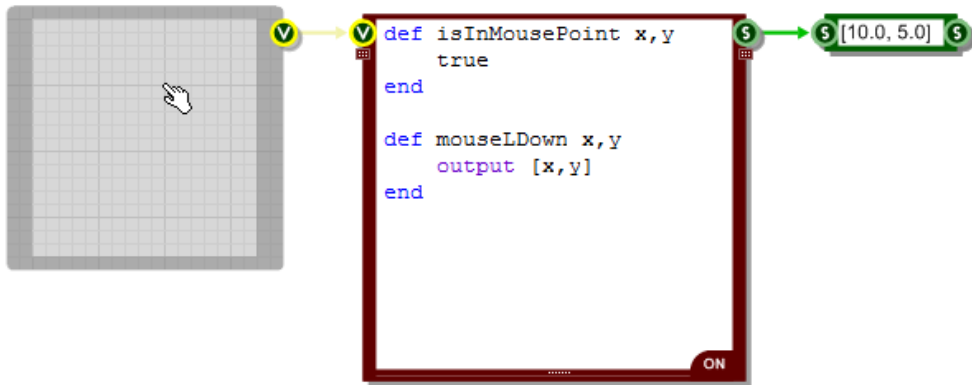
In this case we've elected to handle mouse events regardless of where the mouse pointer is in the View. However, you could use the x and y position to return a different result depending on whether the mouse is in a particular area or location.

## Mouse Clicks

Once you have your `isInMousePoint` method defined you'll receive an event whenever a mouse click occurs in the View (providing `isInMousePoint` returns true for the particular point).

There are 6 methods for responding to standard mouse clicks, 3 for the left mouse button and 3 for the right. These are as follows:

- mouseLDown** - left mouse button clicked
- mouseLUp** - left mouse button released
- mouseLDouble** - left mouse button double-clicked
- mouseRDown** - left mouse button clicked
- mouseRUp** - left mouse button released
- mouseRDouble** - left mouse button double-clicked



Here we defined a left mouse button click method. We made it so that it outputs an array with the coordinates of the click point. You can see that the mouse pointer is at 10 grid squares across and 5 down.

All the other mouse click methods work in the same way.



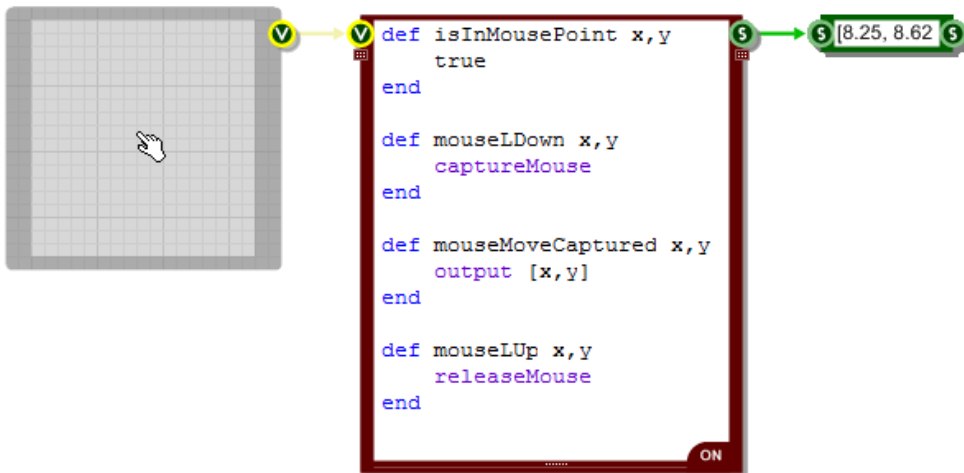
## Mouse Capture & Dragging

If you want to handle drag operations, where the user clicks, holds then moves the cursor, you will need to capture the mouse. To do this you call the **captureMouse** method. This should be done in response to a mouse down event.

Now, so long as the mouse button remains down FlowStone will look for a method called **mouseMoveCaptured** in your Ruby component. If it exists it will be called each time you move the mouse.

When the mouse button is released you'll get a call to the appropriate mouse up method. From this method you will need to call the **releaseMouse** method.

All this is much better illustrated in an example:



Here we've extended the previous example to send the coordinates of the mouse pointer only while you drag the mouse.

## Key Modifiers

Sometimes you want to define different responses to clicks and drags based on whether a particular key modifier has been applied. Most commonly you'll want to check whether the SHIFT, CTRL or ALT keys are pressed at the time of or during interaction.

To do this we have the **isKeyPressed** method:

```
isKeyPressed keyName
```

The method has one input, the `keyName`. This can be either a string representing the key or an integer representing the virtual key code.

So to check the 'q' key, use "q" or the virtual key code of 81 as the key name. You can find a list of virtual key codes here:

[http://msdn.microsoft.com/en-us/library/aa243025\(v=vs.60\).aspx](http://msdn.microsoft.com/en-us/library/aa243025(v=vs.60).aspx)

The method will return true or false depending on whether the key is currently pressed or not.

### Special Keys

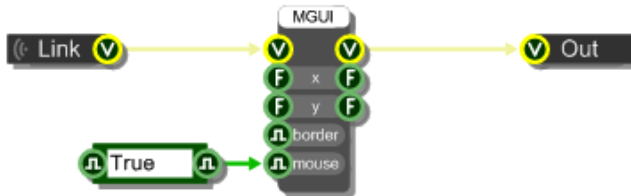
For some special, frequently used keys we have setup shortcut strings to save you having to look up the virtual key codes:

Key	Key Name String to Use (case insensitive)
Shift	"Shift"
Control	"Control" or "CTRL"
Alt	"Alt"
Space	"Space"
Return / Enter	"Return" or "Enter"
Arrow Key Up	"Up" or "Arrow Up"
Arrow Key Down	"Down" or "Arrow Down"
Arrow Key Left	"Left" or "Arrow Left"
Arrow Key Right	"Right" or "Arrow Right"

## Mouse Move

If you want to capture mouse movements when no mouse buttons are pressed then you first need to enable this on the MGUI component you're connecting to. By default mouse move messages are suppressed in order to improve performance.

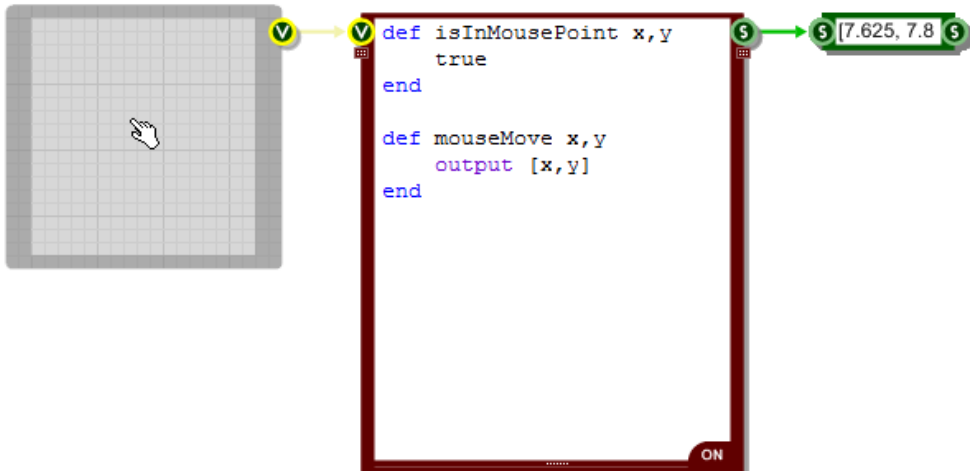
To enable mouse moves you need to go to the MGUI component, connect a Boolean component to its Mouse input and set it to True (as shown below).



The example above shows what we have inside the module that we've been using in the examples we've shown so far.















Once you have enabled mouse moves FlowStone will look for a **mouseMove** method in your Ruby component. Every time the mouse moves in your View the mouseMove method will be called.

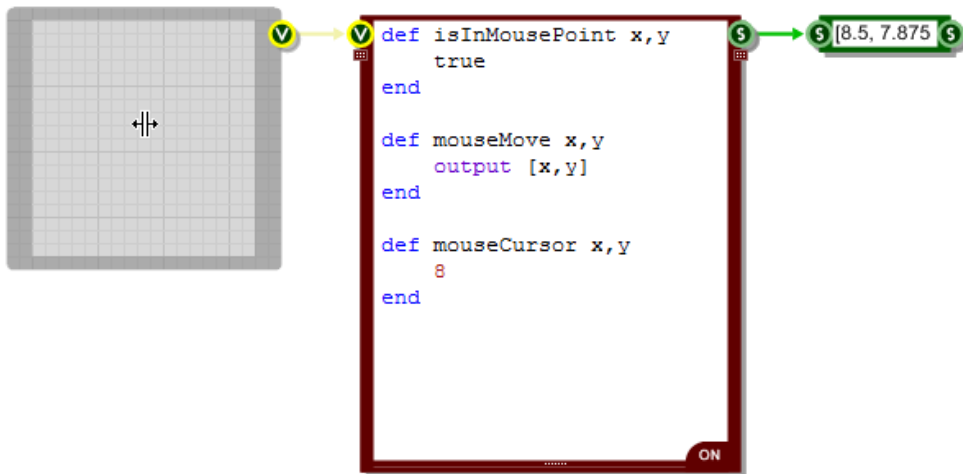
The example below shows the coordinates being sent out each time the mouse moves over the View.



## Mouse Cursor

There are other cursors you can use other than the default one. To set the cursor you need to create a **mouseCursor** method. FlowStone expects a return value from this method which tells it which cursor you want to use. The return value can be an integer or a string:

0 or "default"	
1 or "ibeam"	
2 or "pointer"	
3 or "handpointer"	
4 or "move"	
5 or "pointeradd"	
6 or "pointerdel"	
7 or "smallpointer"	
8 or "resizeew"	
9 or "resizens"	
10 or "handopen"	
11 or "handclosed"	
12 or "crosshair"	
13 or "deny"	



Sometimes it's useful to be able to hide the mouse cursor. A typical example would be when dragging knob or slider controls. You can do this by using the **showCursor** method:

```
showCursor show
```

Here, *show* can be either *true* (to show) or *false* (to hide).

This method maps onto the Windows system API function for controlling the cursor. This function doesn't operate a simple on and off state but instead uses a counter. The counter is incremented when a value of *true* is passed and decremented when a value of *false* is passed. This means that if you make two calls passing a value of *true* you would have to make 3 calls passing a value of *false* in order to hide the cursor.

This is useful when using **showCursor** in nested methods.

# Controls and Dialogs

There are a few user interface elements that we use Windows controls and dialogs for. We have primitives for these but when you're programming an interface using Ruby components it's far easier to be able to access them from within your code.

This section describes these elements and how to use them.

## In Place Edit Controls

Edit controls are pretty much essential for gathering precise numerical or text input. You can create an in place edit control on-the-fly whenever you need to get information by using the `createEdit` method:

```
createEdit input, id, area, [startText [,font [,textColour
[,backColour [,multiline]]]]]
```

The inputs to the method are as follows:

<b>input</b>	- reference to the input View connector (name or index)
<b>id</b>	- an id that you can use on the callback so you know which edit is reporting
<b>area</b>	- four element array [x,y,w,h] to define position and size
<b>startText</b>	- the text that will show in the edit to start with [OPTIONAL]
<b>font</b>	- the font to use (a font object) [OPTIONAL]
<b>textColour</b>	- the colour of the text [OPTIONAL]
<b>backColour</b>	- the colour of the background rectangle for the edit control [OPTIONAL]
<b>multiline</b>	- whether the control should be a multiline edit (true or false) [OPTIONAL]

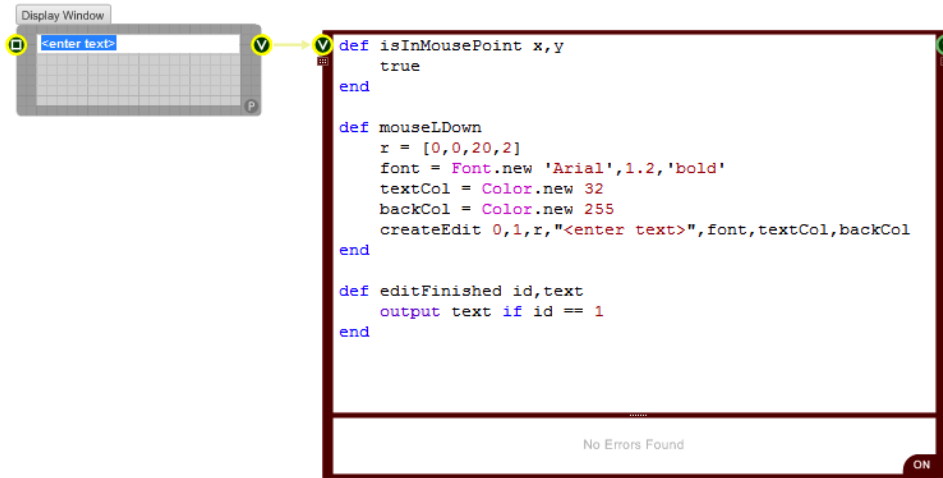
Once the edit control is created it will report back to the Ruby component. You can capture the reported information by implementing one or both of the following methods:

```
def editFinished id,text
end

def editChanged id,text
end
```

The inputs to these methods are the same – the id of the edit control that is reporting (the one you supplied when you created it) and the text that is currently in the control. The only difference between the two is that the `editChanged` method is called with every keystroke whereas the `editFinished` method is only called when the edit control is closed (when the user presses tab or clicks away).

Here's an example of an in place edit control being used to capture input. In this example we've clicked on the UI so the `mouseLDown` method has already been called and created the edit control.



## Drop Lists

For selecting from a discrete list of options we have the drop list control. You can create one using the `createDropList` method:

```
createDropList input, id, pos, items, [checked [,style [,disabled]]]
```

The inputs to the method are as follows:

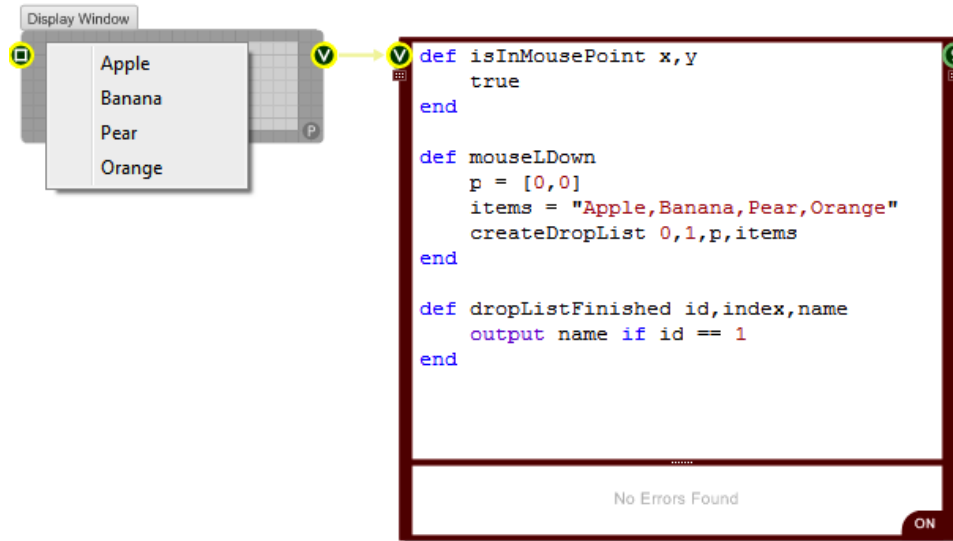
- |                 |  |
|-----------------|--|
| <b>input</b>    | - reference to the input View connector (name or index)                      |
| <b>id</b>       | - an id that you can use on the callback so you know which list is reporting |
| <b>pos</b>      | - top left corner of the drop list   |
| <b>items</b>    | - comma separated string of item names                                       |
| <b>checked</b>  | - index of the item you want to show as checked [OPTIONAL]                   |
| <b>style</b>    | - "scroll", "autocol" or number of items per column [OPTIONAL]               |
| <b>disabled</b> | - the index of the item you want to show as disabled [OPTIONAL]              |

Once the drop list control is created it will report back to the Ruby component once a selection has been made. You can capture the reported information by implementing the following method:

```
def dropListFinished id, index, name
end
```

The `id` input is the id of the drop list that is reporting (the one you supplied when you created it). The `index` is the index of the item that was selected and `name` is the name of the item as displayed in the list.

Here's an example of a drop list control in action. In this example we've clicked on the UI so the `mouseLDown` method has already been called and created the control.



## Message Boxes

It's useful to be able to provide warnings when performing unrecoverable actions or to ask a user for confirmation. This is where message boxes come in. You can create a message box using the `messageBox` method:

```
messageBox id, text [,title [,type ]]
```

The inputs to the method are as follows:

- id** - an id that you can use on the callback so you know which one is reporting
- text** - the text to display in the message box
- title** - the text for the title bar of the message box [OPTIONAL]
- type** - defines how the message box looks and behaves [OPTIONAL]

The type is a combination of strings in any order with any separators you like. The options are as follows:

**Button arrangements:** okcancel, retrycancel, yesnocancel, yesno, abortretryignore

**Icons:** exclamation, warning, information, asterisk, question, stop, error, hand

**Default buttons:** defbutton1, defbutton2, defbutton3, defbutton4

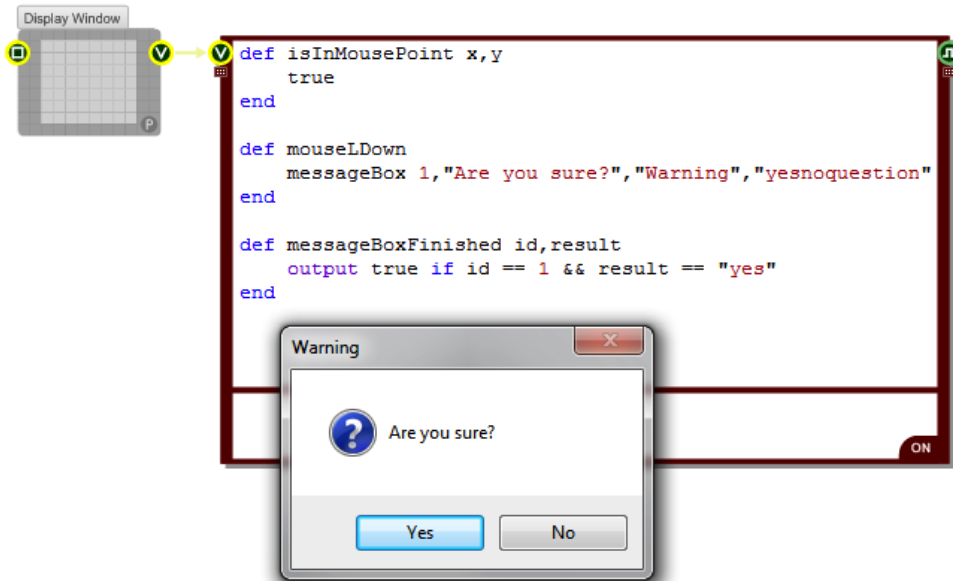


Once the message box has closed it will report back to the Ruby component. You can capture the reported information by implementing the following method:

```
def messageBoxFinished id, result
end
```

The id input is the id of the message box that is reporting (the one you supplied when you created it). The result input is the button that was pressed eg "yes", "ok", "cancel", "retry" etc.

Here's an example of a message box in action. In this example we've clicked on the UI so the mouseLDown method has already been called and created the message box.



# Sounds

FlowStone has a comprehensive signal processing engine for handling sound. Using this you can do just about anything you want from an audio perspective.

However, sometimes all you want to do is play a sound. So to save you having to bother with the audio engine we have provided a simple way to play a sound from within your Ruby code.

You do this using the **playSound** method.

## Playing

This method takes a path to the sound file that you want to play. For example:

```
playSound "C:/Windows/Media/chimes.wav"
```

This will play the wave file `chimes.wav` once from beginning to end.

If you don't provide a path then FlowStone will look in the folder where the schematic is currently located. You can use relative paths to this too, for example: `"Sounds\explosion.wav"`.

By default the method plays the sound asynchronously. This means that your program continues executing as the sound plays.

## Waiting for Completion

If you want, you can override the asynchronous behaviour and have execution wait for the sound to complete. To do this you add a second input parameter and pass the string `"sync"` (meaning synchronous). For example:

```
playSound "C:/Windows/Media/chimes.wav","sync"
```

This will play the wave file `chimes.wav` once from beginning to end as before but the next instruction will not execute until the sound has finished playing.

As a result this is only suitable for very short sounds. If you try and play a sound that has a long duration it could lock up the Ruby thread and cause errors.

## Looping

A sound can be made to loop continuously. You can do this by passing "loop" as the second parameter to the playSound method. For example:

```
playSound "C:/Windows/Media/chimes.wav", "loop"
```

## Stopping

At any time you can interrupt and stop the currently playing sound by passing nil to the playSound method. For example:

```
playSound nil
```

## Utility Methods

Currently we only have one utility method that doesn't fit into any of the categories above and that's the **schematicLocation** method.

You can use this method to give you the path to where the current schematic is located.

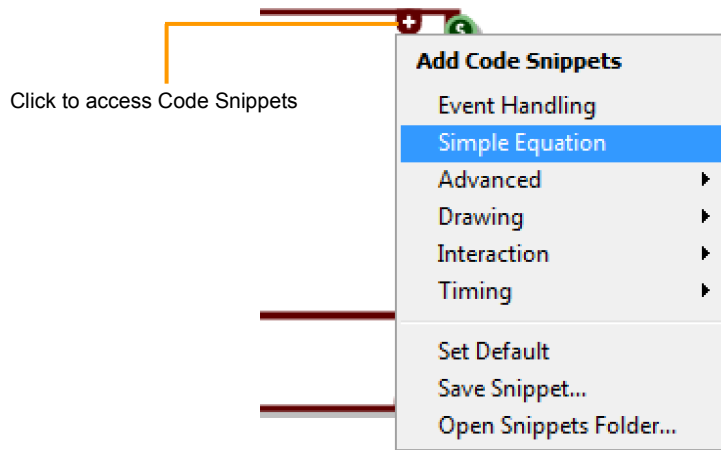
In exported exes or plugins it will give you the location of the exe or plugin.

# Code Snippets

In order to make your Ruby coding quicker we have a neat little feature called Code Snippets. This allows you to store snippets of code that you use frequently and recall them in seconds whenever you need them.

## Adding Code Snippets

To access your code snippets click on the code snippets button as shown in the picture below.



A drop down menu appears showing all your snippets. Some are grouped under sub menus. When you select an option in this menu the appropriate snippet is instantly inserted at the end of your code.

## Default Code

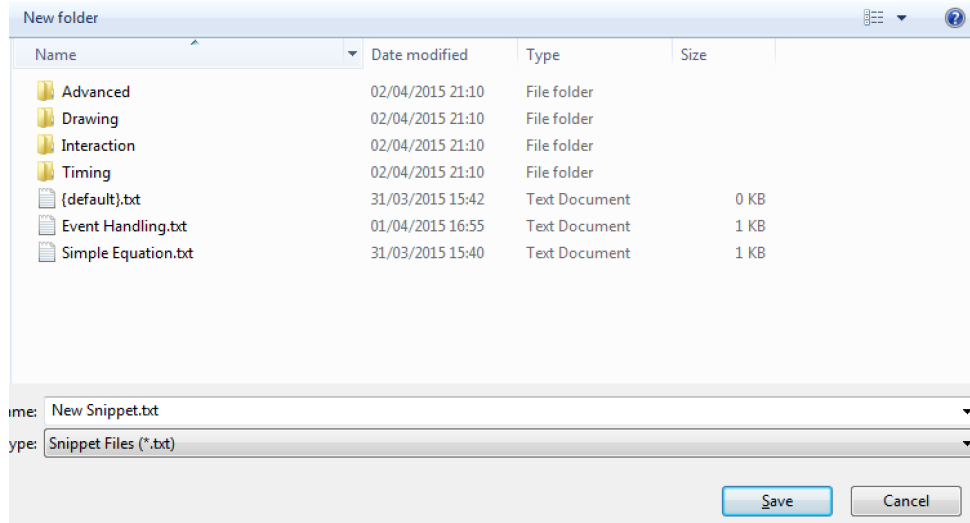
If you want you can have a code snippet appear by default whenever you create a new Ruby component. You might want an event method stub to appear for example.

To do this, just select the Set Default option from the Code Snippets menu and whatever code you have in your Ruby component will now appear whenever you create a new one.

## Saving Snippets

You can create your own snippets really easily. Just write some code you want to save in a Ruby component, then select Save Snippet from the Snippets Menu.

The standard Save dialog will appear.



Snippets are saved as text files in a special Snippets folder. Simply choose a name for your snippet file and click Save. If you want to put it in a sub folder you can click the 'New folder' button.

## Organising Snippets

As we just mentioned, snippets are simply text files stored in a special Snippets folder. You can access this folder by selecting Open Snippets Folder from the Code Snippets menu. From there you can organise and edit your snippets as you please.

Note that there is one snippet called {default}.txt. This is the one that appears by default in every newly created Ruby component. It does not appear in the menu and must remain at the top level in the Snippets folder.

# External DLLs

You can call functions in external DLLs from inside your Ruby code. This is done by making use of the Ruby Win32API extension.

All you do is create an instance of the Win32API class that has all the information about the function call you want to make. You then execute the Win32API **call** method to execute the function. All very straightforward.

Let's take a look in detail about how this is all achieved.

## The Win32API Extension

The key to providing access to external DLLs is the Win32API extension. This is installed in your FlowStone folder in Program Files under \ruby\libraries\win32.

The files are separate from the software but FlowStone knows where to look for them on your PC. If you are only ever working on the same machine then you don't have to worry about any of this.

However, if you create an exported exe that uses the extension and want to pass this to someone else then either they will need to have FlowStone installed on their system or you will have to distribute the \ruby\libraries\win32 folder (including the ruby and libraries parent folders).

The ruby parent folder should then be inside the same folder and at the same level as your exported exe as shown in the picture below:

The first thing you need to do is tell Ruby you want to use the Win32API extension. This is done by calling the Kernel method **require**:

```
require "Win32API"
```

## Creating a Function Object

Before you can call a function in an external DLL you need to create an instance of the Win32API class. The resulting object must have all the information it needs about the target function before it can be used to make the call.

To create a Win32API object:

## CHAPTER 8

```
callObject = Win32API.new( library, functionName, inTypes, outType )
```

The input parameters (shown in italics above) are as follows:

*library*

This is the name of the dll that contains the function you want to call. You don't need to add the .dll file extension so you can leave that off if you wish. If you don't specify a path then the software will look for the dll in the following places:

- the folder from which the application executed
- the current folder
- the Windows system folder (e.g. c:\windows\system or system32)
- the Windows folder
- the folders that are listed in the PATH environment variable

*functionName*

This is the name of the function you want to call. It must be exactly as it is written in the dll and is case sensitive.

*inTypes*

This defines the number of inputs to the function and their types. Each parameter is represented by a single character. You can use a Ruby array with each element as a character ( e.g. ['l','n','c'] ) or you can use a string ( e.g. "inc" ).

The types you can use are as follows:

<b>I or i</b>	- integer
<b>L or l</b>	- long
<b>P or p</b>	- pointer (including char* buffers)
<b>S or s</b>	- string (const char* only)
<b>V or v</b>	- void

For constant strings use 'S'. For string buffers that might be modified by the function call use 'P'.

*outType*

This defines the output type. This is a single character and can be any of those shown above including 'v' (upper or lowercase) to represent a void return value.

As an example we're going to use the GetCursorPos function in user32.dll so our Win32API object would be created as follows:

```
getCursorPos = Win32API.new("user32", "GetCursorPos", ['P'], 'V')
```



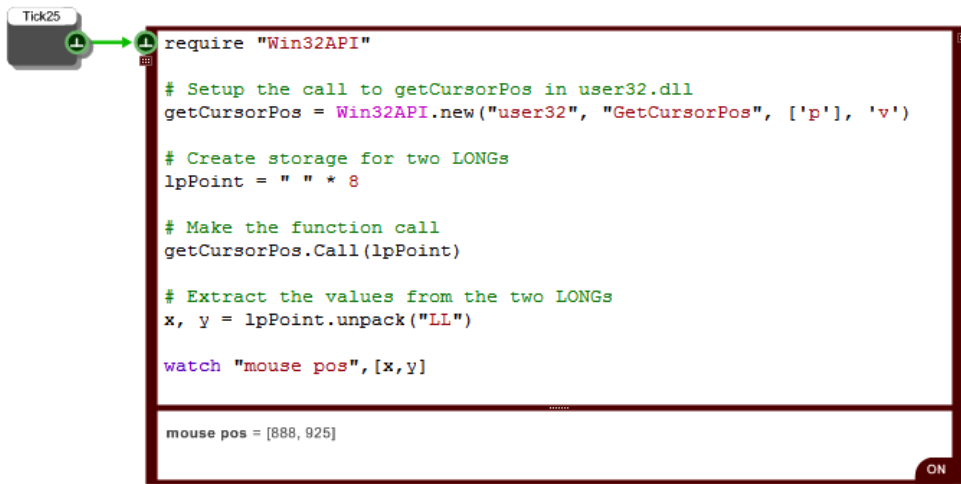
## Making the Call

Once you have your object you are ready to make your function call. To do this we use the Win32API method **call** and pass it a set of parameters that match the ones we supplied when we created the object.

If your function call modifies the input data passed to it then you need to allocate the memory for that data before passing it to the function.

In the example we've chosen the GetCursorPos function takes an LPPOINT which is 8 bytes. So we need to create a string with 8 characters to allocate this memory. We can then unpack this after the call has been made to extract the x and y coordinates we're after.

Here's how everything looks when it's put together:



```

require "Win32API"

# Setup the call to getCursorPos in user32.dll
getCursorPos = Win32API.new("user32", "GetCursorPos", ['p'], 'v')

# Create storage for two LONGs
lpPoint = " " * 8

# Make the function call
getCursorPos.Call(lpPoint)

# Extract the values from the two LONGs
x, y = lpPoint.unpack("LL")

watch "mouse pos", [x,y]

-----

mouse pos = [888, 925]

```

We added a Tick25 so that we can watch the coordinates change as we move the mouse.

Here's another example:



```

def event
  mbox = Win32API.new("user32", "MessageBox", ['I','p','p','i'], 'i')
  mbox.call(0, "Hello!", "Test Message", 0)
end

-----

No Errors Found

```

This will show a message box whenever the trigger button is pressed. The input parameters to the MessageBox call are parent window, message, title, type. Notice that the message and title are

passed as strings. You don't need to do any packing. We use 'p' in the input types because these strings are char\* values, not constants.

Here's one final example. This shows how to retrieve a string value from a function call.

```

$ require 'win32api'

buf = " " * 260
len = [buf.length].pack('L')
a = "Hope this is long enough"
getUser = Win32API.new('advapi32', 'GetUserName', 'PP', 'I', )
getUser.call(buf, len)

watch "buffer",buf
watch "size",len.unpack('L')[0]

.....

buffer = Fred
size = 5

```

We're using a call to GetUserName to return us the name of the current user. For this we have to create a buffer for the function to put the name into and also pass a variable to send the buffer size and receive the length of the string sent back.

To create a suitable buffer we just need a Ruby string of sufficient size. To set up the variable to send the buffer size we have to create a 32bit unsigned int. This we do using the pack method of the Ruby Array class. We put the number we want to initialise the int with in an array ([buf.length] ie. [260]). We call pack on the array passing 'L' meaning 32bit unsigned int. See the Ruby documentation for more on the Array Pack method.

You can see that the buffer returns the name of the user, "Fred" in this case.

The size of the returned string is placed in 'len'. This is a 32bit unsigned int remember so to convert it to a Ruby Int we need to unpack it. The result of calling unpack is an array with the string size as the first entry. We can see the size is 5, the four characters in 'Fred' and the string null terminator.

### Distribution

If you plan to distribute any exported exes or plugins that use an external DLL you must ensure that the DLL is distributed with it. You must also distribute the ruby\libraries\win32 folder that is located in your FlowStone install folder. You need to have a folder called ruby in the same folder as your exported file. Inside this folder should be a libraries folder and inside that you place the win32 folder and all its contents. Without these your export will not function correctly.

# MIDI

When you send MIDI data to a Ruby component it is represented by an instance of the MIDI class. Using Ruby you are then free to process this MIDI object in any way you like. You can create new MIDI objects and you can send objects out of the Ruby component to play a synth in your schematic or somewhere else.

## Reading MIDI Objects

The MIDI class is a very straightforward. A single instance of the class represents one individual MIDI message. FlowStone handles MIDI channel messages and MIDI system exclusive messages.

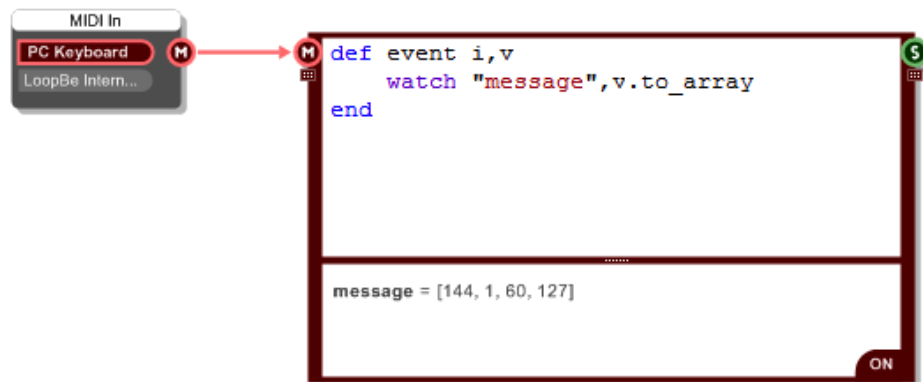
### MIDI Channel Messages

MIDI channel messages are defined by four properties:

- |         |   |
|---------|---|
| Status  | - the type of message eg. note on, program change etc.              |
| Channel | - the MIDI channel on which the message was transmitted (1-16)      |
| Data1   | - first piece of status dependent data associated with the message  |
| Data2   | - second piece of status dependent data associated with the message |

We won't go into the MIDI specification in any great detail here. If you're not familiar with it then there are plenty of good resources online.

To look access these properties for a MIDI object, call the **to\_array** method. This will give you a ruby array with the above properties as the elements, listed in order.




In the above example we're playing a middle C. You can see that the MIDI message shows 144 for the status (ie. NOTE ON) and 1 for the MIDI channel. For MIDI note on and off messages Data1 gives you the note number (60 = middle C) and Data2 gives you the velocity (127 is maximum velocity).

### MIDI System Exclusive

MIDI system exclusive messages (or sysex for short) are single chunks of data expressed in hexadecimal (Hex) form. A sysex hex string starts with F0 and is terminated by F7. The data in between can represent mostly anything that a synth manufacturer may want to send or receive from their MIDI device.

Again, we won't go into the details of sysex here, we'll just explain how you can manipulate it within a Ruby component.

When a MIDI object containing sysex arrives at a Ruby component you can look at it using the **to\_array** method. This will give you a Ruby array containing two entries. The first is the sysex hex data string and the second is the number of hex bytes in that string (as each byte is two characters this number will of course be half the length of the string).



```
def event i,v
  watch "message",v.to_array
end
```

.....

```
message = ["F043104C080B3C06F7", 9]
```

ON

The above example shows how to look at the data in a system exclusive message. You can see the array with the hex data as the first entry.

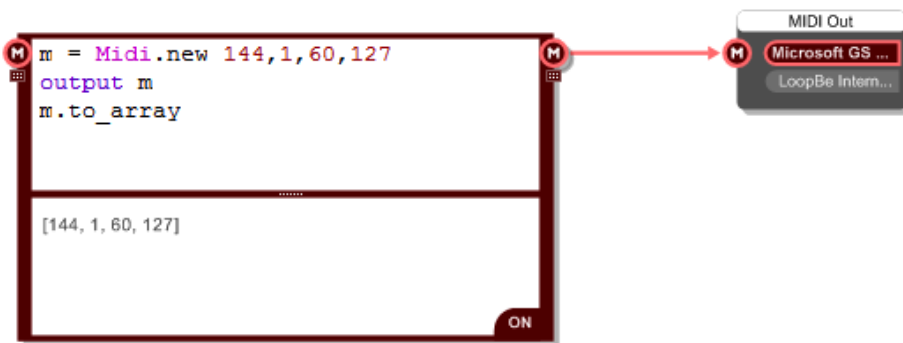
## Creating MIDI Objects

You can create your own MIDI messages using the MIDI class. This is done differently depending on whether you want a MIDI channel or system exclusive message.

### MIDI Channel Messages

To create a MIDI channel message you need to supply the four properties mentioned above as inputs to the **new** method.

The example below shows how to create a NOTE ON message for middle C and send it out.



Ideally you'd want to create a note on together with a note off shortly after, so that you get a note of a particular length. To do this you use the event scheduling capabilities of the Ruby component.

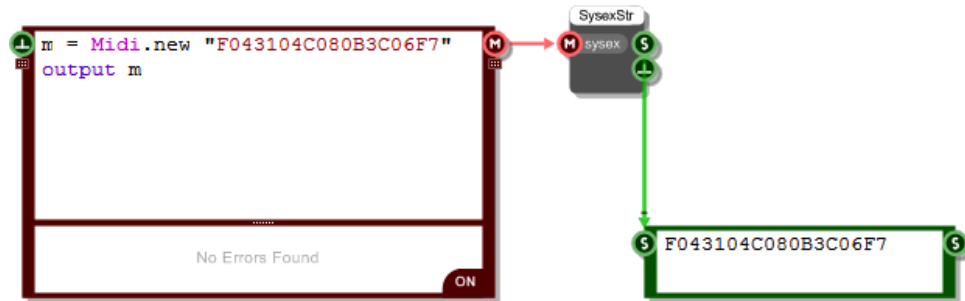


The example above will play a middle C at full velocity lasting a quarter of a second whenever you press the trigger button. We need to make use of the optional time input to the **event** method here. This gives us the current time and we schedule the 'off' MIDI event to be sent at time now + 0.25 seconds.

### MIDI System Exclusive

To create a MIDI sysex message you simply provide the sysex string to the **new** method.

The following example shows how you do this. You can see that we've sent the sysex message out and the Sysex To String component has deciphered it.



# Frames

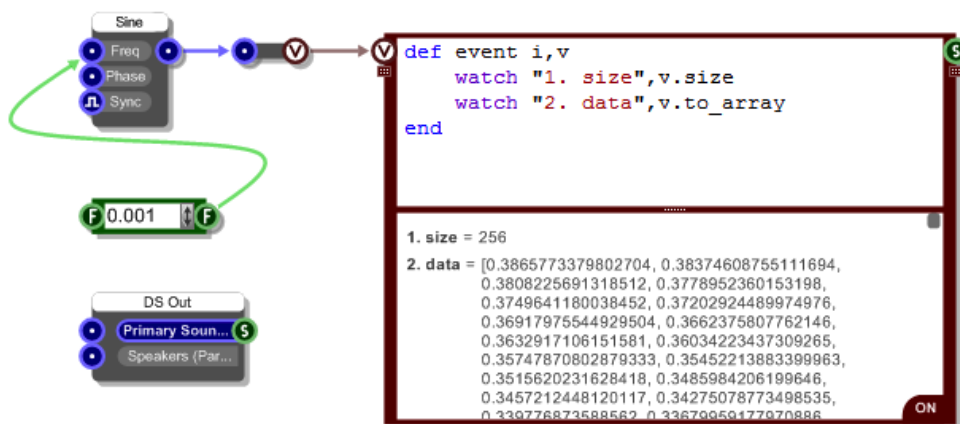
FlowStone processes high data rate signals such as audio using Streams. These allow you to perform one-sample-at-a-time processing at rates of 44.1Khz and beyond with minimal compromise in performance.

Frames allow you to process this data via the Ruby component without losing a single sample. Because of the high data rates involved you can't do this processing one sample at a time – the processing overhead would be way too much. What we can do is give you access to the data in batches of samples. We call these batches Frames.

Processing frames in Ruby can never compete with the speed of streams, so there is a performance cost. However, if squeezing everything out of your cpu is not that critical then, together with Ruby, Frames provide a very flexible way of analysing high data rate signals.

## Mono to Frame

The Mono To Frame component takes Mono stream data and delivers it one frame at a time in precise sync with the stream. The example below shows how you can use this.



The component produces instances of a Frame class. These are generated every time a frame of samples is requested from ASIO or Direct Sound. The size of the frame depends on the ASIO or Direct Sound setup you have.

## The Frame Class

The Frame class encapsulates a frame of samples. Frame objects can be easily manipulated inside a Ruby component.

### Reading Sample

There are two ways to access the individual samples. You can use the element reference operator ie. square brackets to access a particular sample:

```
myFrame[i] - gives you the sample at index i in myFrame
```

If you need access to the whole frame of samples you can convert it to an array:

```
myFrame.to_array - gives you the samples in a Ruby array
```

There's a little bit of an overhead in converting to a Ruby array. However, if you plan on manipulating all the samples then it's probably more efficient to convert to a Ruby array and do the processing on the array than it is to access each sample individually.

### Writing Samples

In the same way that you can read a single sample you can also write to a single sample:

```
myFrame[i] = 3.0 - assigns 3.0 to the sample at index i in myFrame
```

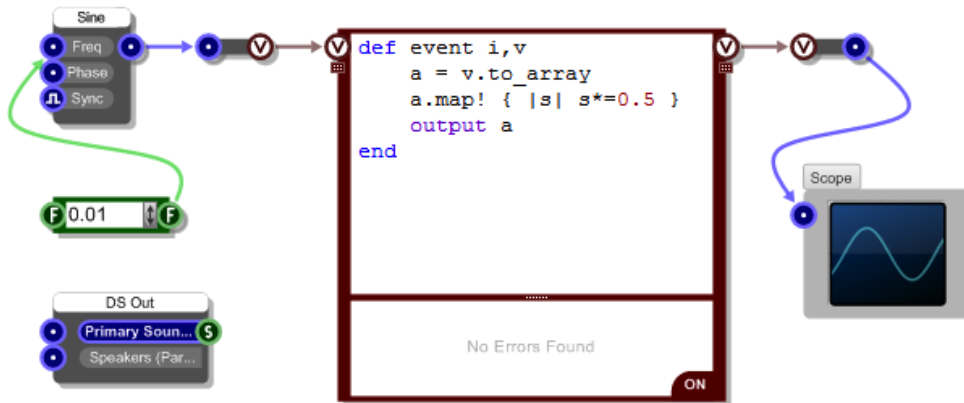
If you want to assign an array of Ruby Floats to a Frame then you'll need to create a new Frame object. Here's how you do it:

```
f = Frame.new smpls - creates a new Frame from the array called smpls
```



## Frame To Mono

The Frame To Mono component takes Frames and passes them back into the Mono stream. Using this component in conjunction with the Mono To Frame component allows you to apply Ruby processing within a Mono section.



In the example above we're taking a sine wave signal and halving the amplitude before passing it back into the Mono stream. The Frame is converted to a Ruby array and then each element in the array is multiplied by 0.5. The resulting array is passed out.

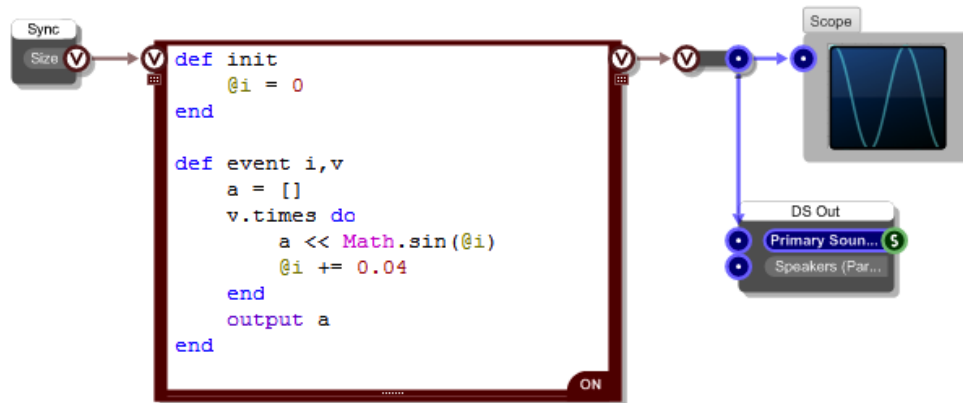
Note that we didn't pass out a new Frame object. This is because the Mono To Frame component can take a Ruby array as input too.

We could easily have created a Frame object too by replacing the "output a" by "output (Frame.new a)".

## Frame Sync

If you just want to generate a signal then you don't need the Mono To Ruby component, you just need to generate Frames from your Ruby component and pass them to the Frame to Mono.

However, what you do need is a way to sync the Frame output with the Mono stream and also a way of finding out the required frame size. This is where the Frame Sync component comes into play.



The example here shows a generated sine wave. The Frame Sync component sends the required Frame size to the Ruby component at the time when a Frame needs to be generated. The Ruby component then creates an array of samples, in this case based on a sine wave and sends this out immediately.

## Processing Frames in a DLL

You can process Frames in an external DLL. This is useful if you need maximum performance or if you have existing algorithms written in another language that you want to make use of.

To show you how this is done we've created a function, written in C, for a simple delay effect and put this in an external DLL. The code for this function is as follows:

```
// Applies a delay effect to a frame
// fsz   - size of frame
// fptr  - pointer to frame
// bptr  - pointer to circular buffer (held externally)
// idx   - index of circular buffer (held externally)
// delay - length of delay in samples
// fb    - feedback amount (0-1)
// dry   - dry mix in amount (0-1)
// returns the current index or -1 if failed
FRAMES_API int delayFrame( int fsz, int fptr, int bptr, int idx,
                          int delay, float* pfb, float* pdry )
{
    if( fsz > 0 && fptr != 0 && bptr != 0 && delay > 0 )
    {
        float* pData = (float*)fptr;
        float* pBuff = (float*)bptr;

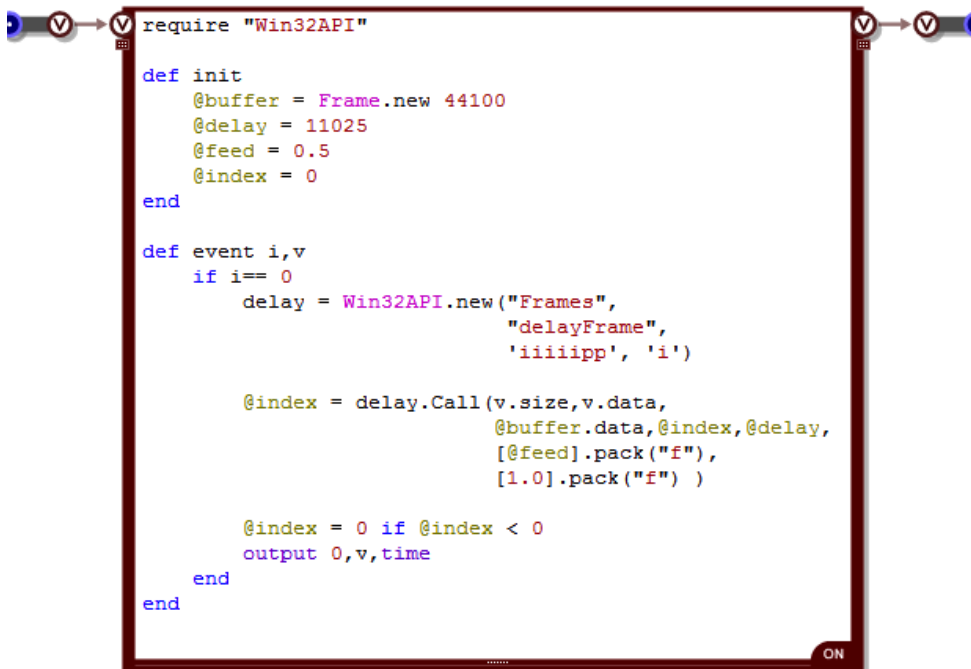
        float curr;

        float fb = *pfb;
        float dry = *pdry;

        for( int i=0; i<fsz; i++ )
        {
            curr = pData[i];
            pData[i] = pBuff[idx] + curr*dry;

            pBuff[idx] = pBuff[idx]*fb + curr;
            idx++;
            if( idx >= delay ) idx = 0;
        }
        return idx;
    }
    return -1;
}
```

To call this from FlowStone we need to use a Ruby component and the Win32API class. See the earlier section on External DLLs for more information about using Win32API.



```

require "Win32API"

def init
  @buffer = Frame.new 44100
  @delay = 11025
  @feed = 0.5
  @index = 0
end

def event i,v
  if i== 0
    delay = Win32API.new("Frames",
                        "delayFrame",
                        'iiiiipp', 'i')

    @index = delay.Call(v.size,v.data,
                       @buffer.data,@index,@delay,
                       [@feed].pack("f"),
                       [1.0].pack("f") )

    @index = 0 if @index < 0
    output 0,v,time
  end
end
end

```

We use a circular buffer for the delay and this together with the current buffer index needs to be managed on the FlowStone side. For the buffer we use a Frame object.

You can see from the C code of the external DLL that the function is called delayFrame and it takes 5 ints followed by two pointer inputs.

In the Call method we pass ints directly. So the size of the frame, the current index and the delay amount are all passed without any need for conversion. The other two int values are pointers to the Frame data for the current frame and the circular buffer. These are accessed via the **data** method of the Frame class. This method gives you a Ruby int value which is the memory address of the raw sample data. By doing this we don't need to convert between the underlying sample data and Ruby which is much more efficient.

The remaining two inputs are Float values. Floats are not handles explicitly by Win32API so you need to pack them into 32 bit floats.

That's all there is to it. The call is made and the external function updates the data in the current Frame object and the circular buffer.

**NOTE:** if you want to create a standalone that uses a dll in this way then see the External DLLs section for details on what you have to include in your distribution.

# Ruby Timeout

When code in a Ruby component is executed, FlowStone waits until it completes before allowing the schematic to proceed. It is sometimes possible to inadvertently write code that performs an infinite loop or a very long calculation. To protect against this FlowStone has a maximum time that it allows a Ruby component before interrupting the calculation and switching the component off as a precaution.

This is called the Ruby Timeout. It defaults to 8000 milliseconds but you can change this if you have some Ruby code that might need longer to execute.

There are two ways to do this.

## Global Timeout

You can change the global timeout setting by selecting Advanced from the Options menu. You'll see the timeout value at the bottom of the dialog box.

### Ruby Timeout

Interrupt Ruby operations after:  milliseconds

## Local Timeout

If you want finer control you can set the timeout for an individual Ruby component. You can do this by using the **setTimeout** method:

```
setTimeout timeout
```

The timeout is in milliseconds. Be careful not to set it too low. A minimum of around 500ms is recommended but you have total control and this minimum isn't enforced.

To remove the local timeout and make the Ruby component use the global setting set the timeout to any value less than zero.

You can check the timeout for any Ruby component using the **timeout** method.

# Ruby Limitations

Our main aim for Ruby in FlowStone was to provide a scripting mechanism that binds tightly into our graphical programming environment and allows you to do things that were not possible before or were very difficult previously using graphical programming techniques.

Ruby integration brings a massive number of benefits but this is Ruby within FlowStone and as such there are some limitations when compared with using pure Ruby via text files and the command line.

The vast majority of people will be able to use Ruby in FlowStone without encountering any of these limitations. However, for some advanced users of Ruby there are certain aspects of the implementation in FlowStone that you should be aware of as they may restrict some of the things you might want to do. This section covers those limitations.

## Single Interpreter

The first thing to note is that there is a single interpreter for each instance of FlowStone. This means that if you open two schematics within the same exe they will share the same interpreter – the same Ruby airspace so to speak.

Individual Ruby components are always completely self contained so any instance variables you use there relate only to the Ruby component in which they are declared. However, if you declare global variables or classes in one schematic then they will be visible in the 'Ruby space' of the other schematic.

Most of the time this will not be an issue. It's just worth knowing in case you happen to run into any behaviours that may arise from this.

## Standard Ruby Libraries

The standard installation of Ruby comes with a number of libraries. We don't currently supply all of these with FlowStone. In fact we only supply one, win32 (used for connecting to external DLLs).

The main reason for this is that if you choose to export to a standalone exe or plugin, because the libraries are external to Ruby you would need to distribute the library files alongside your export.

We hope to be able to provide an automatic way of dealing with this in the future so that you don't need to think about dependencies on libraries – instead it would all be dealt with in the export process. This would allow us to be able to offer the full set of standard libraries.

## Declaration Order

Because all Ruby components share the same interpreter you can declare globally available classes and variables in one Ruby component and they will be available in all others. This is a very useful feature. However, you need to make sure that when you re-load your schematic, the declarations occur before they are needed. To do this the Ruby component that contains the declarations must be interpreted before any Ruby components that use those declared items.

When FlowStone loads a schematic it starts with the top-level schematic and then loads each component on that schematic in the order they were added. When it reaches a module it will load the schematic for that module and the process continues in the same way.

The earlier a Ruby component appears in this ordering the earlier it will be interpreted. So let's say that you have some class declarations in a Ruby component that are only used in a particular module. If you ensure that the Ruby component is the first component in the schematic for the module then all will be fine.

Unless you really plan ahead and add the Ruby component for your declarations first it's likely that you'll find yourself in a situation where you need to change the order of components so that it's the first one. You can do this by selecting all the other components, cutting them and pasting them back in.

## Declaration Persistence

The Ruby interpreter is running constantly. Because of this any declared classes, global or instance variables and constants will remain declared even if you delete the code for them. Currently the only way to flush all the declarations is to save your schematic and restart FlowStone.

## Gems

All seasoned Rubyists will know about Gems. These are packaged libraries of code that you can install and use to extend Ruby beyond its standard capabilities.

If you have a Gem you want to use with FlowStone then if it is implemented in pure Ruby you should have no problem using it. You will need to place the ruby files for the gem in a folder and then reference that folder when you write your 'require' statement. You could also put it in the FlowStone install folder under "ruby/libraries".

If your Gem is written partly or wholly in C then it needs to be (or have been) compiled using Visual Studio in order for it to work with FlowStone. If it has been compiled with mingw (as many Windows gems are) it is unlikely to work.

## CHAPTER 8

The same distribution issues apply to gems as for the standard libraries. If you decide to export your creation and it uses a gem then you need to distribute the gem alongside it. You must also make sure that the paths to the gem are all taken care of or it won't load.

### **Gems and VST Plugins**

Currently VST plugins generated by FlowStone cannot make use of Ruby extensions or Gems.



# Ruby DLL

Ruby functionality is provided by an external DLL. This dll is called msvcr90-ruby191.dll. In order to use this with FlowStone it has to be compiled using Microsoft Visual Studio 2008.

To comply with the Ruby licence agreement we provide details here of how to build it.

## Changes

We have made no changes to the Ruby source that change the way that the Ruby interpreter functions. However, we have added a small section of code into eval.c that we use to reference one of our own objects.

The code is:

```
void* g_pInterpreter = 0;
void* ruby_interpreterPtr(void) { return g_pInterpreter; }
void ruby_setInterpreterPtr(void* ptr) { g_pInterpreter = ptr; }
```

This should be inserted into eval.c just after:

```
#include "eval_jump.c"
```

This change does not affect the way Ruby works or add to it in any way. It merely allows us to store an object within the scope of the dll.

## Building the DLL

You must have a version of Microsoft Visual Studio 2008 installed before you can proceed.

1. Download the ruby source code from here:  
<http://mirrors.ibiblio.org/ruby/1.9/ruby-1.9.3-p0.zip>
2. Unzip to a folder eg. [C:\ruby-src](#)
3. Open the Visual Studio/C++ Command prompt as Administrator
4. Run vcvars32.bat
5. Move into C:\ruby-src\ruby-1.9.3-p0
6. In the command prompt execute the following instructions:

```
win32\configure.bat
```

```
nmake
```

```
nmake test
```

```
nmake install
```

The msvcr90-ruby191.dll binary will be located in the C:\ruby-src\ruby-1.9.3-p0 folder.

9

# DSP Code Component

THE ULTIMATE LOW-LEVEL DSP TOOL

# DSP Coding

For most tasks, FlowStone's graphical programming approach meets your needs nicely. However, when it comes to programming DSP a more algorithmic approach is required. For this purpose we have the Code Component.

Using the code component you can do any kind of processing you like. The component uses a very small set of commands to allow you to translate DSP algorithms into very simple code. The code is then dynamically compiled into binary that runs extremely fast for maximum performance.

## The DSP Code Component

You'll find the DSP code component in the Code group of the toolbox. The component provides an area into which you can type your code. Click on the component to go into edit mode. You'll stay in edit mode until you click or tab away.

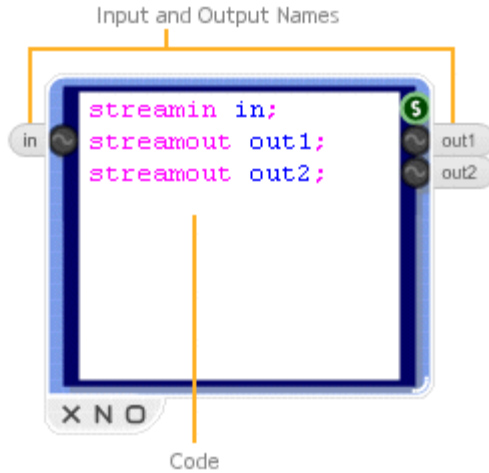
## Inputs and Outputs

In order to link the code into a schematic you'll need some inputs and outputs. The code component is used for audio signal processing so it only allows you to create Poly, Mono, Stream or Stream Boolean inputs and outputs. There are eight commands for adding inputs and outputs:

```
polyin, polyout, monoin, monoout, streamboolin, streamboolout, streamin,  
streamout, polyintin, polyintout, memin
```

You need to make sure that you give each input and output a name and that each line of code is terminated with a semicolon. Once the correct syntax has been applied, the input or output will appear automatically when you click or tab away.

Stream boolean inputs/outputs allow you to process or receive boolean mask data. See the sections on Expressions and Conditional Statements for more information on masks.



The input/output names become variables that you can use in subsequent code. The variables will take whatever values come in or go out when you link the component to other parts of a schematic. If nothing is linked to an input, the variable will assume a value of zero.

One final point worth noting is that it's a good idea to use streamin and streamout instead of the poly and mono alternatives as your code can then be used in both poly and mono sections.

## Syntax Colouring

The editor colours the code to indicate correct syntax. Input/Output commands, operators and brackets are purple, variables are blue and data types and functions are green.

If at some point the syntax colouring stops and all text after that point turns black, this is an indication that there is an error in the syntax at that point.

## Editor

The code editor supports copy and paste through the standard shortcut keys (CTRL+C, CTRL+V). A local undo is also implemented to allow you to undo and redo typing, deletions, pastes etc. The local undo applies to any changes you make during a particular session (Between clicking in the component and clicking away). After that you can still undo via the application's undo system, but this will only go back through changes made between edits.

You can scroll using the mouse wheel or use the cursor keys. You can page using the PGUP and PGDN keys. CTRL+HOME will go to the top and CTRL+END will go to the bottom.

You can Find search strings by using CTRL+F. After finding the first occurrence you can find again by pressing F3.

You can also Replace text by using CTRL+H. Highlight a selection of text first if you want to search within that as opposed to all the text in the component.

## Local Variables

In addition to the data you draw from your schematic via inputs you can also create local variables. These are defined in exactly the same way as the inputs and outputs.

There is only one type of variable at the moment and that's a float. This represents a floating point number. As with the inputs and outputs you need to give the variable a name and terminate the command with a semicolon.

Variables can be initialised at the same time as you declare them. You can write:

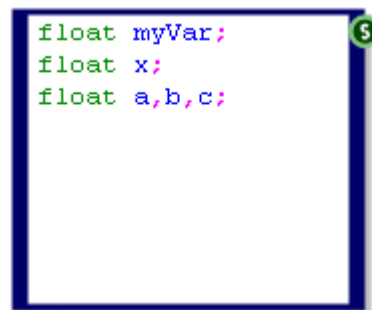
```
float x=3;
```

And x will be assigned the value 3. Variables that are not initialised explicitly will be set to zero.

You can declare multiple variables in a single line of code. For example:

```
float x,y,z;
```

Will create three variables x,y and z.



```
float myVar;
float x;
float a,b,c;
```

## Assignments

Having set up your inputs, outputs and local variables you can then move on to implementing your DSP algorithm.

To assign a value to a variable or to send it to an output, use the equals operator =. You can assign fixed values or you can use a regular expression to calculate the value.

The example opposite shows how you can combine all the elements discussed so far to create a simple moog filter.

```

streamin in;
streamin cutoff;
streamin res;
streamout out;
float out1,out2,out3,out4;
float in1,in2,in3,in4;
float f,fb,f2,input;
input=in;
f=cutoff*1.16;
f2=f*f;
fb=res*(1.0-0.15*f2);
input=input-out4*fb;
input=input*0.35013*f2*f2;
out1 = input+0.3*in1+(1-f)*out1;
in1=input;
out2=out1+0.3*in2+(1-f)*out2;
in2=out1;
out3=out2+0.3*in3+(1-f)*out3;
in3=out2;
out4=out3+0.3*in4+(1-f)*out4;
in4=out3;
out=out4;

```

## Expressions

Expressions can use a combination of mathematical operations and built in functions. The following represents a complete summary of the operators and functions that are currently supported.

<code>sin1(a)</code>	Mathematical sine, cosine and tangent
<code>cos1(a)</code>	The 1 indicates that the input value <code>a</code> is converted to a value in the range 0-1 and not 0 and $2\pi$ . So 0.5 would be equivalent to $\pi$ and 1.25 would be equivalent to $\pi/2$
<code>tan1(a)</code>	
<code>log10(a)</code>	Logarithm (base 10) of <code>a</code>
<code>max(a,b)</code>	The numerically largest (max) or smallest (min) of <code>a</code> and <code>b</code>
<code>min(a,b)</code>	
<code>rndint(a)</code>	Rounds <code>a</code> to the nearest integer value above or below (so 1.4 becomes 1 but 1.6 becomes 2)
<code>*</code> <code>+</code> <code>-</code> <code>/</code>	Standard mathematical operations
<code>%</code>	Calculates the remainder after dividing the left-hand side by the right-hand side
<code>&gt;</code> <code>&gt;=</code> <code>&lt;</code> <code>&lt;=</code>	Comparison operators These generate a mask based on the result of the comparison
<code>&amp;</code>	Bitwise AND operator. Use this to apply a mask



One limitation of the code component is that you can only have 8 successive operations in an expression. For example, the expression:

```
a = 0+1+2+3+4+5+6+7+9+10;
```

is not allowed because there are 9 addition operations in a row. You can easily get round this by splitting the expression up into smaller sections using brackets '(' and ')'. For example:

```
a = (0+1+2+3+4) + (5+6+7+9+10);
```

## Conditional Statements

Because FlowStone uses SSE to process four channels at a time, conditional statements can't be implemented in the traditional way. However, you can achieve the effect of a conditional statement by using a mask.

A mask can be applied to all four channels at the same time. Each channel is affected differently. FlowStone includes conditional operators that generate exactly these kind of masks.

For example, the following FlowStone code:

```
x = x - (x >= 1) & 1.0;
```

is equivalent to the following C/C++ code:

```
if( x >= 1 ) x = x - 1.0;
```

The >= operator creates a mask. The mask will effectively be true for each channel that meets the condition and false for each one that doesn't. The bitwise & will return a value of 1.0 or 0.0 for each SSE channel based on the mask.

## Comments

You can leave comments in your code. These are just snippets of text that remind you of what a particular part is doing. Comments take up no processing time. They can be put at the end of a line or they can have a whole line to themselves

For example:

```
index = index + 1; //Increment index
//Increment index
index = index + 1;
```

# Advanced Features

## Arrays

You can declare arrays in the code component. These act as buffers that you can use for storing past values for example or for using as a lookup table.

An array is declared as follows:

```
float buffer[100];
```

This defines an array called buffer which is 100 floats (4x100=400 bytes) in size.

### Initialising

By default all the entries in the array are set to zero. You have two other options for initialising an array:

1. Set all entries to a particular value

```
float buffer[100] = 0.62;
```

2. Randomise the contents of the array

```
float buffer[100] = rand(-0.5, 0.5);
```

### Accessing

To access a particular entry in an array use the array specifiers (square brackets) '[' and ']'.

```
out = buffer[25] ;  
buffer[6] = 3.14159;
```

Note that the array index is zero based so in the example above buffer[0] would be the first entry and buffer[99] would be the last entry in the array.

You must make sure that you use indexes that are within range for the array, failure to do so will produce unpredictable results and could even crash the software.

## Mem Input

You can access data from external Mem components directly in the code component. Once declared they behave exactly like arrays.

A mem input is declared as follows:

```
memin sound[1000];
```

This will create a Mem type connector on the component so that you can pass in data from outside. You need to specify a maximum size for the mem data this is so that enough space is reserved for mem data in the compiled code. Whilst Mem's do know their size we don't want to be recompiling every time a mem changes so by declaring a maximum from the outset the Mem size can change dynamically without affecting anything.

The above declaration would allow for Mem's that are 4 x 1000 bytes in size. If you pass in a Mem that is larger it will get truncated.

## Hop

There are some calculations that don't need to be calculated for every sample, it's sufficient to recalculate for every 4<sup>th</sup> sample say. For this purpose we have the hop command.

The syntax for the hop command is as follows:

```
hop(8)
{
    < code in here is executed only for every 8th sample >
}
```

Hops must be powers of 2 and the maximum hop length is 4096 so values of 2,4,8,16,32,64,128,256,512,1024,2048 and 4096 are the only acceptable values.

## Loop

Sometimes you need to execute the same code several times, changing one or two parameters on each pass. You can do this in the code component using the loop command.

Here's the syntax:

```
loop(10)
{
    < code in here is executed 10 times in succession >
}
```

## CHAPTER 9

```
}
```

You could use a loop to initialise an array for example:

```
float index;  
float buffer[10];  
loop(10)  
{  
    buffer[10] = sinl(index);  
    index = index+0.1;  
}
```

You should take care not to use too many loops or loops with many iterations as they can cause significant increases in cpu. Also for some loops (like the array initialisation example) you only want them to be executed once, this is where the stages come in.

## Stages

The code is split into four parts called stages. These are numbered 0,1,2 and 3.

### Stage(0)

Stage(0) runs only for the first sample and is intended for processing that you only want to happen at the very start, usually to set up other parameters that you will then use on every sample.

For the example we've just been looking at we could put the array initialisation into Stage(0) and it will run only once.

Defining the stage is very easy:

```
stage(0)
{
    < code in here is executed only for the first sample >
}
```

### Stage(2)

We'll jump to Stage(2) now because this is the default stage. Any code written outside of a Stage definition is assumed to be Stage(2).

Stage(2) code is executed for every sample (including the first) and happens after Stage(0).

### Stage(1) and Stage(3)

These two stages are only used for implementing delays. They are needed to make sure that data flows correctly when one or more delays are chained together in a feedback loop.

The software executes the code in stage order so after Stage(0) Stage(1) goes next.

Stage(1) should be used to move data from internal variables (most likely a buffer) and the outputs. This ensures that all delays have a consistent output value before doing any calculations.

Next the standard Stage(2) is executed. Any code that needs the outputs of any delays will have the values they need now.

Finally Stage(3) is executed. This should be used to move data from the inputs to internal variables (again most likely a buffer) and perform any calculations.

Note that each stage is executed for the whole of the schematic before proceeding to the next stage.

An example of how to use Stage(1 ) and Stage(3) for a delay is shown below.

## CHAPTER 9

```
streamin in;
streamout out;
streamin delay;
float mem[44100];
float index;
stage(1)
{
    out = mem[index];
}
stage(3)
{
    mem[index] = in;
    index = index + 1;
    index = (index<delay)&index;
}
```

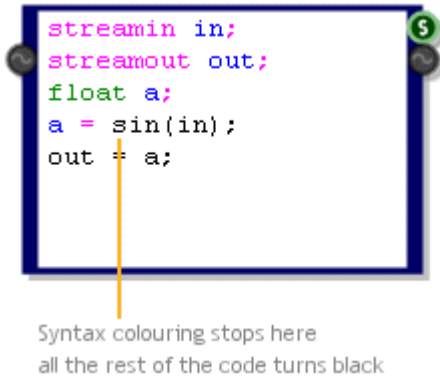
Note that because the transfer to outputs in Stage(1) occurs before the buffer update in Stage(3) this will always produce a minimum of 1 sample delay. If you need a delay that will work correctly when the input delay length is zero samples then you can implement this by modifying the above code as follows:

```
stage(1)
{
    out = mem[index]&(delay>0) + in&(delay==0);
}
```

## Debugging

### Using Syntax Colouring

We've already seen that the first indication you'll have that your code is incorrect will be a discontinuity in the syntax colouring. The following example shows this.



You can see that the colouring stops with the sin function. This indicates that there is something wrong with the expression. Sure enough we forgot the '1', the line should read:

```
a = sin1(in);
```

If we make this correction everything becomes coloured correctly.

### Checking the Assembler

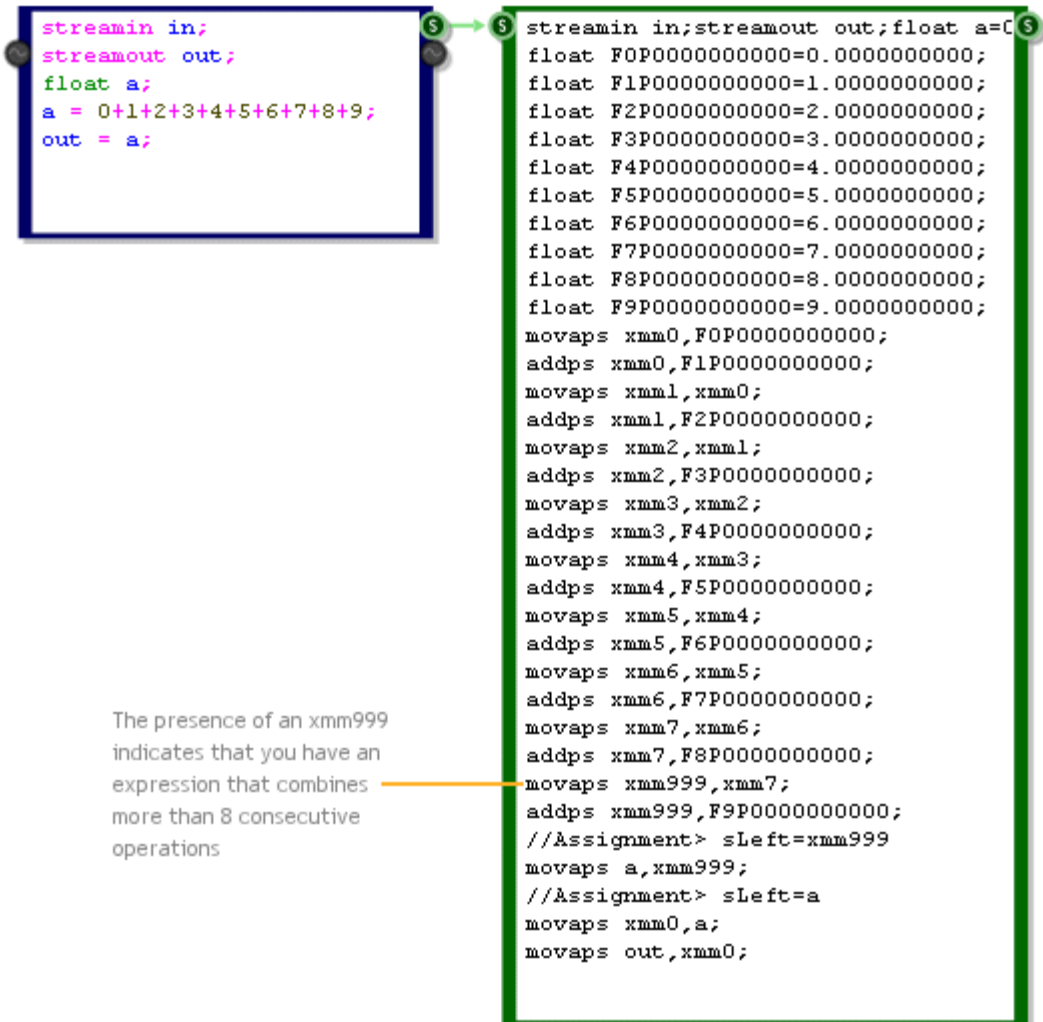
The code component converts the high-level language that it uses into x86 assembler prior to compiling. You can look at this assembler code by connecting a Text component to the String output of the code component.

If you understand assembler (and even if you don't) it can be useful to examine this code to make sure that nothing out of place is happening.

When there is a syntax error the assembler code will only be generated up to the point at which the syntax broke down.

If the syntax is correct then the other possible problem could be the limitation on expressions discussed earlier. This is much more difficult to spot in the code because the colouring will indicate correct syntax. However, a quick look at the assembler code can identify this type of problem.

The example below shows exactly this. The way to spot the problem is to look for xmm999 in the code. If this happens then you know to look at your expressions to make sure that the 8 consecutive operator limit has not been exceeded.





# Assembler

If you want you can bypass the high-level language of the Code Component and write your own x86 assembler. For this purpose we have the Assembler component.

## Syntax

Declaration of inputs, outputs and local variables is the same as for the Code component. However, the language you use is now x86 Assembler.

If you want you can copy assembler code that is generated by the Code component (as we shown in the previous section) and paste this directly into the Assembler component. This can be helpful if you are relatively new to a86 assembler.

We don't support the full range of opcodes and some opcodes are limited to certain registers. However, we do support a wide range and we will be adding more over time.

## Opcodes

Below you will find a list of the supported opcodes.

We have used the following abbreviations to refer to registers, variable types etc:

- xmm** - SSE 128bit registers (xmm0, ... xmm7).
- var** - 128bit variable in memory
- reg** - x86 general 32 bit registers (eax, ebx, ecx, edx)
- channel** - SSE channel (0..3)
- label** - label string
- int** - 32 bit integer value

**SUPPORTED X86 OPCODES**

<b>add</b> reg, var / reg, int	<b>fptan</b>	<b>movss</b> xmm, var / var, xmm / xmm, xmm
<b>addpd</b> xmm, var	<b>frndint</b>	<b>mulpd</b> xmm, xmm / xmm, var
<b>addps</b> xmm, xmm / xmm, var	<b>fscale</b>	<b>mulps</b> xmm, xmm / xmm, var
<b>and</b> reg, int	<b>fsin</b>	<b>or</b> reg, int / reg, reg
<b>andnps</b> xmm, xmm / xmm, var	<b>fsincos</b>	<b>orps</b> xmm, xmm / xmm, var
<b>andps</b> xmm, xmm / xmm, var	<b>fst</b> var[channel]	<b>paddb</b> xmm, xmm / xmm, var
<b>call</b> reg	<b>fstp</b> var[channel] / var[eax] / st[type]	<b>pand</b> xmm, var
<b>cmp</b> reg, int / reg, reg	<b>fsub</b>	<b>pcmpgtd</b> xmm, xmm
<b>cmpps</b> xmm, xmm, type / xmm, var, type	<b>fxch</b>	<b>pmuludq</b> xmm, xmm / xmm, var
<b>cvtdq2pd</b> xmm, xmm	<b>fyl2x</b>	<b>pop</b> reg
<b>cvtdq2ps</b> xmm, xmm / xmm, var	<b>inc</b> [reg]	<b>pslld</b> xmm, int
<b>cvtpd2ps</b> xmm, xmm	<b>jmp</b> label	<b>psrld</b> xmm, xmm / xmm, var
<b>cvtps2dq</b> xmm, xmm / xmm, var	<b>jl</b> label	<b>psubd</b> xmm, xmm / xmm, var
<b>cvtps2pd</b> xmm, xmm	<b>jle</b> label	<b>push</b> reg
<b>cvtps2dq</b> xmm, xmm / xmm, var	<b>jmp</b> label / int	<b>rcpps</b> xmm, xmm
<b>dfld</b> var[channel]	<b>jng</b> label	<b>rdtsc</b>
<b>divps</b> xmm, xmm / xmm, var	<b>jnl</b> label	<b>ret</b>
<b>f2xm1</b>	<b>jnz</b> label / int	<b>shl</b> reg, int
<b>fadd</b>	<b>jz</b> label	<b>shr</b> reg, int
<b>fcos</b>	<b>maxps</b> xmm, xmm / xmm, var	<b>shufps</b> xmm, xmm, int
<b>fld</b> var[channel] / [eax]	<b>minps</b> xmm, [reg]	<b>sqrtps</b> xmm, xmm
<b>fist</b> var[channel]	<b>minps</b> xmm, xmm / xmm, var	<b>sub</b> reg, reg / reg, int / reg, var
<b>fistp</b> var[channel]	<b>mov</b> reg, var / var, reg / reg, int / [eax], int / eax, [ebp+int] / [reg], reg	<b>subpd</b> xmm, var
<b>fld</b> [reg] / var[channel] / var[eax]	<b>movaps</b> [reg], xmm / xmm, [reg] / xmm, xmm / xmm, var / xmm, var[eax] / var, xmm / var[eax], var	<b>subps</b> xmm, xmm / xmm, var
<b>fmul</b>	<b>movd</b> reg, xmm	<b>xorps</b> xmm, xmm / xmm, var
<b>fprem</b>	<b>movd</b> xmm, reg / xmm, var / var, xmm	

# 10 DLL Component

THE ULTIMATE IN FLEXIBILITY AND PERFORMANCE

# Introduction

The DLL component allows you to call your own custom code which you write in C/C++ and compile into an external dynamic link library (DLL).

The DLL can be kept external to the software or you can choose to embed it directly inside a DLL component so it's completely self contained. This allows you to create your own components which you can easily share with others.

You can of course already make your own components from modules in FlowStone. However by using compiled code in a separate DLL you have the advantage of huge flexibility and streamlined performance. You can also keep any proprietary algorithms secret as, unlike a module which is build from lower level components, the compiled code is not readable within a schematic.

For the rest of this chapter we assume that you have an understanding of how to program in C/C++ and how to build DLLs using a compiler such as Visual Studio.

# The Component



The DLL component has 6 inputs and 3 outputs initially.

The **Dll** and **Func** inputs are for the path to the dll and the name of the function you want to call. The **Embed** input determines whether the dll is left external to the software or whether it is to be embedded into the component.

The 3 boolean outputs will indicate whether the inputs that lie opposite them have succeeded. So the first output will be true if the dll was found, the second will be true if the function was found within the dll and the third will be true if the dll can be successfully embedded into the component (some dlls can't be embedded because they depend on other dlls – more on that later).

The **Reset** input disconnects the software from the dll. This is useful if you want to make changes to your dll and then retry them in your schematic without having to close it. Note that all DLL components that use the dll will be disconnected because otherwise the dll would remain locked and you wouldn't be able to rebuild it.

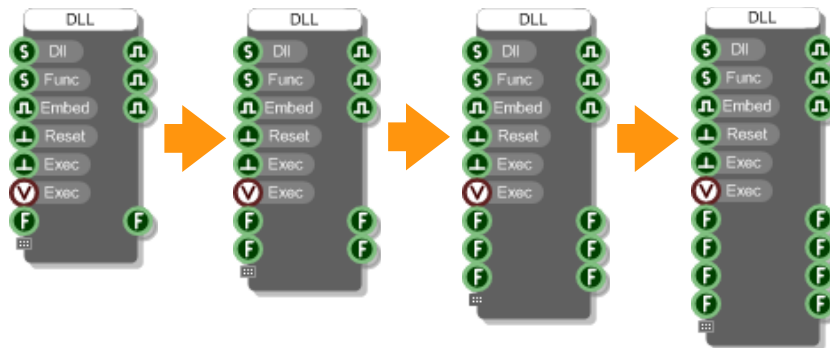
Finally the **Exec** inputs allow you to call the function in the dll. Two inputs are provided depending on whether you want to execute the call based on a green data trigger or a Ruby event.

## Defining Inputs and Outputs

In most cases you'll want to be able to send data to and receive data from the dll. You define this data by dragging the gripper control on the component itself. This is located below the very last input (similar to the Ruby component).

Dragging the gripper control down will create more inputs and outputs.

You'll notice that connectors are added in pairs – one input and one corresponding output.



We add the connectors in pairs in order to keep things simple and also because for Ruby Frame types you must have a matching input and output.

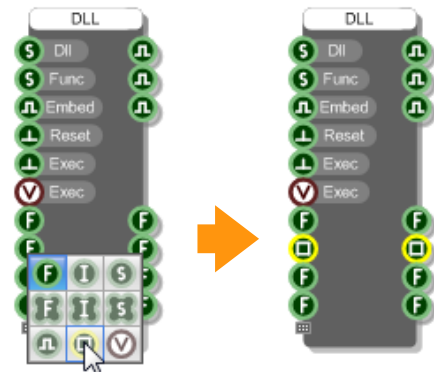
You still have all the flexibility you need. If you need an input but not an output, just ignore the corresponding output and vice-versa.

### Connector Types

The DLL component supports 9 FlowStone data types: Float, Int, String, Float Array, Int Array, String Array, Boolean, Bitmap and Ruby Frame. With this subset you can do just about anything you need to.

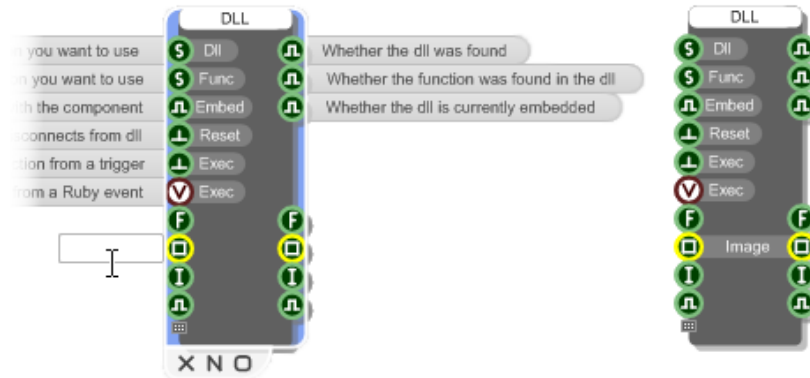
When you create new inputs and outputs as described above, the type of connector added will be the same as for the previous one (or a Float if you haven't added any yet).

To change the connector type, right-click on the input then choose the new type from the pop up menu. When you change the input type the corresponding output will change accordingly.



### Connector Labels

In order to make it easier to remember what each connector is for you can add labels. To do this first select the component then hold CTRL and hover over where the input labels would go. You'll see the outline of an edit box appear.



Click in the box, type a label then hit return. The label will appear on the component. You can change the label in exactly the same way.

Use the TAB key to move between labels and edit several at a time.

# The DLL

Now that we've looked at the component let's turn our attentions to the dll. As you've seen, the DLL component calls a particular function in your dll. This function must have a very specific declaration which goes as follows:

```
extern "C" __declspec(dllexport) void myFunction( int nParams, int* pIn, int* pOut )
```

**nParams** tells you how many parameters are being passed. This is equal to the number of inputs you created on the DLL component.

**pIn** is an array which contains the input values that have been passed to the component. The contents of each entry depend on the connector type.

**pOut** is an array which you use to pass values back to the component. Each entry maps onto an output on the component. You manage the contents of this array but the array itself is created for you by FlowStone.

The pOut array is filled with zeros to start with (each entry is set to zero). The only exception to this is if the type is Ruby Frame in which case the output entry points to the same value as the input entry.

We'll talk about data types next and have a look at how you access the information passed to you from the DLL component. We'll also look at how you pass data back to the DLL component using the pOut array.

## Data Types

The pIn and pOut arrays are declared as int\* but each entry is not necessarily an int. The value changes depending on the data type and to get at this data you may need to do some C casting. More on that in a moment.

First we'll look at the various FlowStone types and how they are represented in C when they arrive at your dll. We've seen in the last section that the DLL component allows you to use 9 different data types. These are represented in C as shown in the table below.



FlowStone Type	C Representation
Int	<code>int</code>
Float	<code>float</code>
Boolean	The first byte of the entry is a <code>bool</code>
String	<code>char*</code> an array of chars which is zero terminated
Float Array	<code>float[n+1]</code> where n is the number of array entries and the first 4 bytes is an int giving you the number of entries
Int Array	<code>int[n+1]</code> where n is the number of array entries and the first 4 bytes is an int giving you the number of entries
String Array	<code>char*[n+1]</code> where n is the number of array entries and the first 4 bytes is an int giving you the number of entries
Bitmap	<code>unsigned char[n+12]</code> where n = width x height x channels. The first 4 bytes give you the width, the next 4 bytes give you the height and the next 4 bytes give you the number of channels (all as ints).
Frame	<code>float[n+1]</code> where n is the number of array entries and the first 4 bytes is an int giving you the number of entries. This is the same as for a Float Array

Let's take a look at each type in more detail.

## Ints

Say we have a DLL component where input 'x' has been defined as an Int connector. The `pIn` array is already defined as an array of ints so to get at this value in C all we only need do the following:

```
int value = pIn[x];
```

If we then wanted to set the corresponding output value we'd do this:

```
pOut[x] = value;
```

## Floats

For the Float type we need to cast to a C float like this:

```
float value = *((float*)&pIn[x]);
```

Note that we can't just cast the value of `pIn[x]` to a float as this would convert from an int to a float. The 4 bytes that make up the int value need to be addressed as if they were 4 bytes making up a float. Hence we need to cast the address of the value to a float pointer and then dereference that.

To set the corresponding output we need to do the same cast for the `pOut` array:

```
*((float*)&pOut[x]) = value;
```

## Booleans

Boolean types become a C `bool` type. A `bool` in C is a single byte. We therefore cast as follows:

```
bool value = *((bool*)&pIn[x]);
```

To set the corresponding output we need to do the same cast for the `pOut` array:

```
*((bool*)&pOut[x]) = value;
```

## Strings

From the table you'll see that FlowStone Strings are represented as a zero terminated array of chars. In this case we need to cast as follows:

```
char* value = *((char**)&pIn[x]);
```

Up to now we've dealt with data whose value is contained within the 4 bytes that defined the array element. However, for strings the value is a pointer to other data i.e. an array of chars.

So when it comes to sending a string to the output array we need to create our own array of chars. The memory management we leave up to you.

This time, to set the corresponding output we need to do more than assign a value. Something like the following would be typical for strings:

```

// Delete previous string and reset pOut entry
if( pOut[x] )
{
    delete *((char**)&pOut[x]);
    pOut[x] = 0;
}

// Create new string (assume we have declared 'length' - add 1 for null terminator)
char* s = new char[length+1];

// {initialise the string here}

// Don't forget to null terminate
s[length] = 0;

// Assign to correct entry in the output array
*((char**)&pOut[x]) = s;

```

If you're using the same length string each time then you could create it once and save having to delete and make a new one each time. If you're working with strings of a maximum length then you could also stick with one string and move the null terminator each time. It's up to you.

## Float Arrays

These are represented as a C array of floats but with an extra entry at the start which provides 4 bytes to store the number of array elements. So we have two casts to make, one for the array size and one for the array itself:

```

int size = *((int*)pIn[x]);
float* array = (float*)pIn[x]+1;

```

You can see that we interpret the first entry as an int giving us the array length. The array itself then begins at the second entry, hence the '+1'.

To set the output value we need to create a new array. This is a similar process as for Strings. Once again the management of memory is up to you. Here's one way of doing it:

```

// Delete previous array and reset pOut entry
if( pOut[x] )
{
    delete *((float**)&pOut[x]);
    pOut[x] = 0;
}

// Create new array (assume we have declared 'length' - add 1 for size at front)
float* array = new float[length+1];

// Set the size
*((int*)array) = length;

```

```

// {initialise the array here}

// Assign to correct entry in the output array
*((float**)&pOut[x]) = array;

```

This will create a new array each time the function is called. Again, you could create a single array if you're working with fixed sizes or adjust the array only if you need more storage space.

## Int Arrays

Int arrays are handled in an identical way to Float arrays. The only difference is the base type, a C int in this case. Again, there's an extra entry at the start which provides 4 bytes to store the number of array elements. So we have two casts to make, one for the array size and one for the array itself:

```

int size = *((int*)pIn[x]);
int* array = (int*)pIn[x]+1;

```

You can see that we interpret the first entry as an int giving us the array length. The array itself then begins at the second entry, hence the '+1'.

To set the output value we need to create a new array. The management of memory is up to you. Here's one way of doing it:

```

// Delete previous array and reset pOut entry
if( pOut[x] )
{
    delete *((int**)&pOut[x]);
    pOut[x] = 0;
}

// Create new array (assume we have declared 'length' - add 1 for size at front)
int* array = new int[length+1];

// Set the size
*((int*)array) = length;

// {initialise the array here}

// Assign to correct entry in the output array
*((int**)&pOut[x]) = array;

```

This will create a new array each time the function is called. You could create a single array if you're working with fixed sizes or adjust the array only if you need more storage space.

## String Arrays

String arrays are probably the most complicated of all the data types to work with. You not only have to manage the array but you also have to manage each String in the array.

A string array is represented by a C array of `char*`. As with the other arrays, there is an additional element at the front which is used to store the size of the array.

Accessing the array is similar to Float and Int arrays in that there are two casts to make, one for the array size and one for the array itself:

```
int size = *((int*)pIn[x]);
char** array = (char**)pIn[x]+1;
```

You can see that we interpret the first entry as an int giving us the array length. The array itself then begins at the second entry, hence the '+1'.

To set the output value we need to create a new array and each string that goes in it. Here is one way of doing this:

```
// Delete previous array (and contents) and reset pOut entry
if( pOut[x] )
{
    int size = *((int*)pOut[x]);
    char** array = (char**)pOut[x]+1;

    for( int i=0; i<size; i++ )
    {
        if( array[i] )
            delete array[i];
    }
    delete (char**)pOut[x];
    pOut[x] = 0;
}

// Create new array (assume we have declared 'length' - add 1 for size at front)
char** array = new char*[length+1];

// Set the size
*((int*)array) = length;

// Create the strings in the array
for( int i=1; i<length+1; i++ )
{
    // Create new string (assume we have declared 'stringlen')
    char* s = new char[stringlen+1];

    // {initialise the string here}

    // Don't forget to null terminate
    s[stringlen] = 0;

    // Add to the sting array
    array[i] = s;
}
}
```

```
// Assign to correct entry in the output array
*((char***)&pOut[x]) = array;
```

## Bitmaps

Bitmap data is also provided as an array, this time of BYTES (unsigned chars). The array begins with 12 bytes of data descriptor information. This is used to supply 3 int parameters: bitmap width, bitmap height and number of channels (bytes per pixel).

You can access these values as follows:

```
int width = *((int*)pIn[x]);
int height = *((int*)pIn[x]+1);
int channels = *((int*)pIn[x]+2);
```

The number of bytes of pixel data we will have is then the product of these values:

```
int bytes = width * height * channels;
```

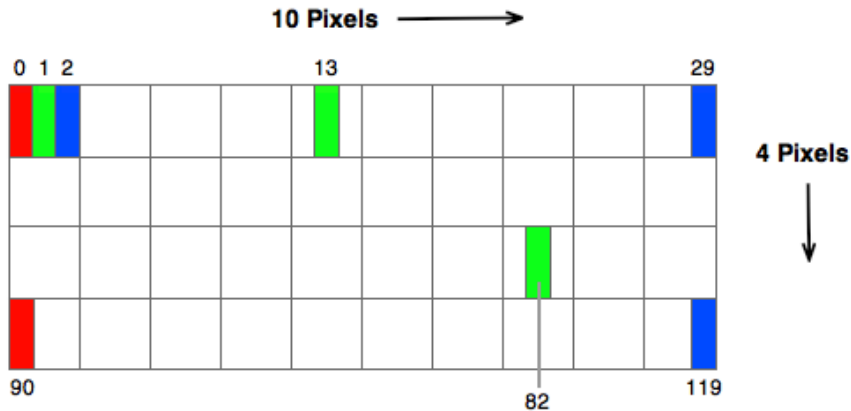
You can get at the array of pixel data like this:

```
unsigned char* pData = (unsigned char*)pIn[x]+12;
```

The data will give us the value for each channel (0-255) for each pixel in the bitmap. The pixel data runs from top-left across to the right before starting the next line.

### EXAMPLE

Let's say we have 3 channels (Red, Green and Blue) and a bitmap that is 10 pixels wide and 4 pixels high. We will have  $10 \times 4 \times 3 = 120$  bytes of pixel data. `pData[0]` will give you the red component of the top-left pixel. `pData[29]` will give you the green component for the rightmost pixel in the top row. `pData[90]` will give you the bottom-left pixel red component and `pData[119]` will give you the blue component of the very last (bottom-right) pixel.



To set the output value we need to create a new byte array. Again, this is very similar to the arrays and strings. Once again the management of memory is up to you:

```
// Create a new byte array (assuming we have already defined bytes)
unsigned char* pNewBytes = new unsigned char[bytes+4*3];
*((unsigned char**)(ampOut[x])) = pNewBytes;

// Set the data descriptors (assume width, height and channels have been defined)
*((int*)pNewBytes) = width
*((int*)pNewBytes+1) = height
*((int*)pNewBytes+2) = channels

// Get pointer to pixel data
unsigned char* pPixelData = pNewBytes+12;

{you then need to initialise your byte data}
```

This only shows you how to create the data. You may have to provide additional code to manage when the bitmap data is deleted. For example, you may want to create a new array and then keep reusing it (as opposed to deleting and recreating it every time). You may then want to delete and recreate it only when the bitmap changes size. All this is down to you to decide.

## Frames

By handling Ruby Frame objects (generated by the Mono To Frame component) you can process audio at sampling rate.

Like Float arrays, Frames are represented as a C array of floats. They have the same extra entry at the start providing 4 bytes to store the number of array elements followed by the float array itself. To separate these we do the following:

```
int size = *((int*)pIn[x]);
```

```
float* array = (float*)pIn[x]+1;
```

Unlike Float Arrays, Frames use the same array for both the input and the output. This means that you don't have to create a new array to pass to the output array – it has already been done for you. The main reason for this is that the output array must be the same size as the input one. However, there is also a benefit in terms of efficiency too.

So all you need to do is process the input array and update it. Just be careful not to assign a value to an array element until you've finished using it or you've stored its value otherwise you may end up using the new value instead of the one that came originally.

## Helpers

The casting can be a bit cumbersome so we came up with some simple macros that make accessing and setting the input and output arrays a bit easier and more readable.

Also, for most of the types you should always check that `pIn[x]` or `pOut[x]` are zero before attempting to extract values using the above casts so the macros incorporate this check too which reduces crashing.

The macros are shown below:

```
#define GETFLOAT(p) *((float*)&p)
#define GETBOOL(p) *((bool*)&p)
#define GETINT(p) p
#define GETSTRING(p) *((char**)&p)
#define GETFLOATARRAY(p) p ? ((float*)p+1) : 0
#define GETINTARRAY(p) p ? ((int*)p+1) : 0
#define GETSTRINGARRAY(p) p ? ((char**)p+1) : 0
#define GETARRAYSIZE(p) p ? *((int*)p) : 0
#define GETFRAME(p) p ? ((float*)p+1) : 0
#define GETFRAMESIZE(p) p ? *((int*)p) : 0
#define GETBITMAPWIDTH(p) p ? *((int*)p) : 0
#define GETBITMAPHEIGHT(p) p ? *((int*)p+1) : 0
#define GETBITMAPCHANNELS(p) p ? *((int*)p+2) : 0
#define GETBITMAPDATA(p) p ? ((BYTE*)p+12) : 0
#define GETBITMAPBYTES(p) p ? *((int*)p) * *((int*)p+1) * *((int*)p+2) : 0
#define NEWINTARRAY(p,n) if(n>0) { *((int**)&p)=new int[n+1]; ((int*)p)[0]=n; }
#define NEWFLOATARRAY(p,n) if(n>0) { *((float**)&p)=new float[n+1]; ((int*)p)[0]=n; }
#define NEWSTRINGARRAY(p,n) if(n>0) { *((char***)&p)=new char*[n+1]; ((int*)p)[0]=n; }
#define DELETESTRING(p) if(p) { delete *((char**)&p); p=0; }
#define DELETEINTARRAY(p) if(p) { delete *((int**)&p); p=0; }
#define DELETEFLOATARRAY(p) if(p) { delete *((float**)&p); p=0; }
```



```
#define DELETESTRINGARRAY(p) if(p) { for( int j=0; j<*((int*)p); j++ ) { if( ((char**)p+1)[j] ) delete
    ((char**)p+1)[j]; } delete *((char***)&p); p=0; }
```

To illustrate how these macros can help the code below shows how the code for resetting the output value for an Int Array, would reduce down to just two lines:

```
// Delete previous array and reset pOut entry
DELETEINTARRAY(pOut[x])

// Create new array (assume we have declared 'length' - add 1 for size at front)
NEWINTARRAY(pOut[x],length)
```

If you want to use the macros, simply copy and paste them in at the top of your dll source file.

## Example 1 – Float Add

This is a really simple example which simply adds two floats together and outputs the result. Whilst it is basic it shows you how all the things we've talked about come together.

```
extern "C" __declspec(dllexport) void addFloats( int nParams, int* pIn, int* pOut )
{
    if( pIn && pOut && nParams >= 2 )
    {
        float f0 = GETFLOAT(pIn[0]);
        float f1 = GETFLOAT(pIn[1]);
        GETFLOAT(pOut[0]) = f0+f1;
    }
}
```

## Example 2 – String Uppercase

This example takes a string and makes it uppercase. There's a bit more to it in that you need to manage the memory for the output string that holds the result.

```
extern "C" __declspec(dllexport) void makeUppercase( int nParams, int* pIn, int* pOut )
{
    if( pIn && pOut && nParams >= 1 )
    {
        char* strIn = GETSTRING(pIn[0]);

        DELETESTRING(pOut[0]);

        if( strIn )
        {
            int i=0;

            int len = strlen(strIn);
            char* strOut = new char[len+1];

            while( strIn[i] )
            {
                char c = strIn[i];
                strOut[i] = toupper(c);
                i++;
            }

            strOut[len]=0;
            GETSTRING(pOut[0]) = strOut;
        }
    }
}
```

### Example 3 – Audio Delay

This little delay effect will give you an idea of how to do audio processing in your dll. The effect expects 6 inputs, the frame, number of samples delay, feedback amount, mix level, a counter to maintain the position in the circular buffer and the circular buffer itself.

We could have created the buffer and counter as global variables in the dll but we've chosen to supply them from the schematic so that the dll could be used by more than one DLL component at a time.

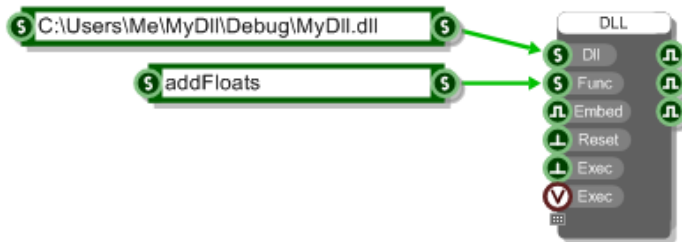
```
extern "C" __declspec(dllexport) void makeUppercase( int nParams, int* pIn, int* pOut )
{
    if( pIn && pOut && nParams >= 6 )
    {
        if( pIn[0] )
        {
            float* pData = GETFRAME(pIn[0]);
            int n = GETFRAMESIZE(pIn[0]);
            int delay = GETINT(pIn[1]);
            float feed = GETFLOAT(pIn[2]);
            float mix = GETFLOAT(pIn[3]);
            int ctr = GETINT(pOut[4]);
            float* pBuffer = GETFRAME(pIn[5]);
            int buffSize = GETFRAMESIZE(pIn[5]);

            if( pBuffer && buffSize >= delay && pData )
            {
                float curr;
                for( int i=0; i<n; i++ )
                {
                    curr = pData[i];
                    pData[i] = (mix)*pBuffer[ctr] + curr;
                    pBuffer[ctr] = pBuffer[ctr]*feed + curr;
                    ctr++;

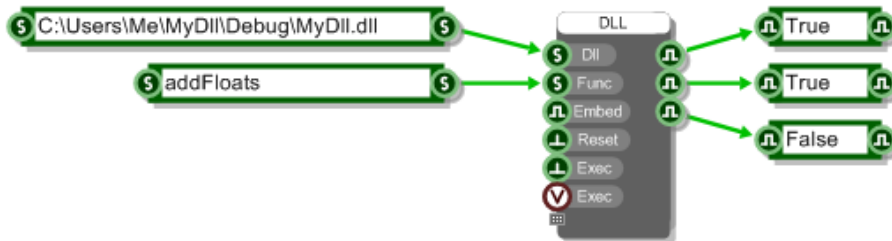
                    if( ctr >= delay ) ctr = 0;
                }
                GETINT(pOut,4) = ctr;
            }
        }
    }
}
```

# Connecting

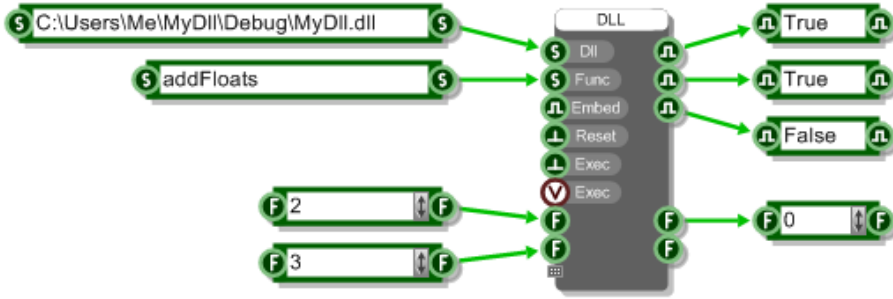
Once you have built your dll it's time to connect to it from FlowStone. Drag in a DLL component and connect Strings to specify the dll path and function name.



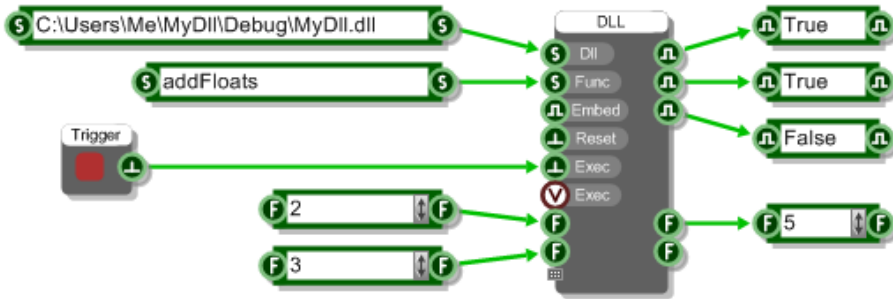
Connect Boolean components to the first three outputs so you can check that the component is communicating with your dll. The first two should read 'true' if the dll and function have been found. If they read 'false' then check carefully the path and function and make sure they match your dll.



Next add the inputs and outputs that your dll will need and wire them up as you require. For the addFloats we need two Float inputs and outputs.



Finally you need to have something attached to one of the two Exec inputs. Unless you are processing Ruby Frames you will need to use the green trigger Exec input (the first of the two). You can connect a trigger button to this for now but if you make it into a module then you may need to wire it up so that changes to the module inputs will trigger the Exec.



That's it! Click the trigger button and your dll will process the inputs and send any result out.

## A note about Outputs

As we saw in the previous section the array of output data is populated in your dll code. Anything you put there will be accessible from the outputs on the DLL component. However, how and indeed whether the outputs send out this data depends on which Exec input you use to invoke the call to your dll function.

### Exec 1 - green Trigger connector

If you use the first Exec input then all outputs that are not a Ruby Value connectors will be sent a trigger and hence anything you have connected to these outputs will be updated after the dll function has been executed.

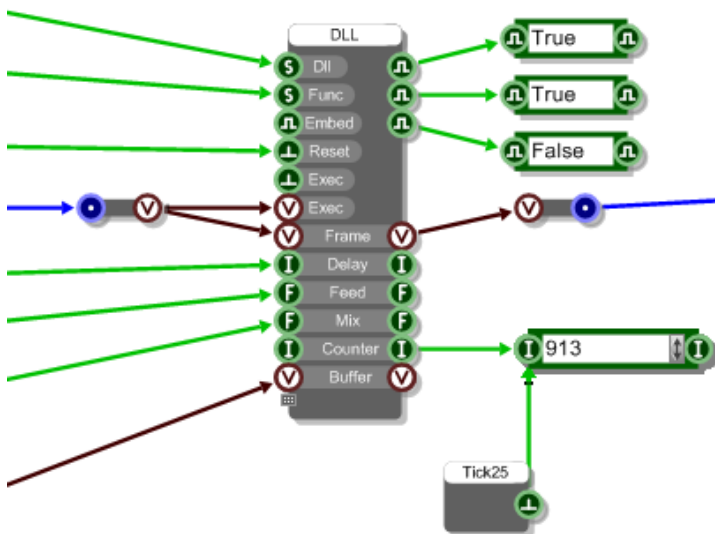
### Exec 2 – Ruby Value connector

If you use the second Exec input then all outputs that are also Ruby Value connectors will be sent a Ruby event after the dll function has been executed.

All other connectors will have their values updated but no triggers will be sent. This is because the Ruby Value input is usually used for audio which is running at sampling rate. Triggering green connectors at this frequency would be way too cpu intensive.

If you do need to be able to read a green triggered data output under these circumstances then the best way to do this is to connect a trigger from a Ticker (or something similar) to the same input that the output from your DLL component is connected to.

This is much better illustrated in a picture:



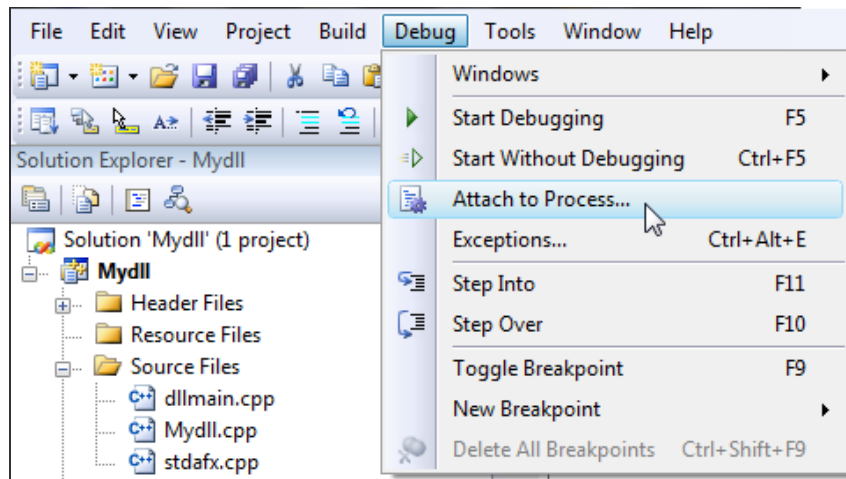
# Debugging

Working at such a low level, it's inevitable that you will run into bugs at some point. Thankfully most compilers will allow you to attach to the FlowStone process so you can step into your code and see where any crashes are occurring. The next section describes how to do this in Visual Studio.

## Debugging using Visual Studio

To debug your dll using Visual Studio:

1. First run FlowStone and open the schematic which you are using to connect to your dll.
2. Open your dll project in Visual Studio (make sure you've built it in debug or with debugging information).
3. From the Debug menu choose 'Attach To Process'



- From the process list select FlowStone and click 'Attach'

Process	ID	Title
devenv.exe	5508	sm - Microsoft Visual Studio
dwm.exe	1668	
explorer.exe	263284	
explorer.exe	1736	Start
FlowStone.exe	35884	FlowStone - [uppercase and add floats.fsm*]
iexplore.exe	249804	
iexplore.exe	249688	
jusched.exe	332	
...	...	...

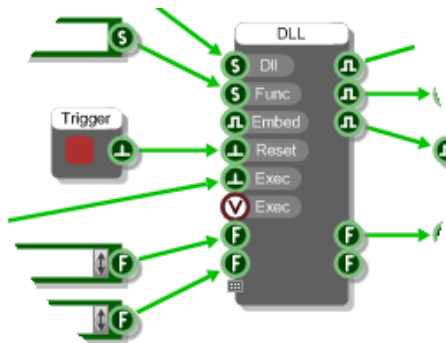
Visual Studio will now attach to FlowStone. You can set break points in your dll code then trigger the Exec input on the dll component inside FlowStone and Visual Studio will stop execution at the break point you have specified. You can now step through your code, trap crashes and browse variables in the usual way.

In our experience with making the example projects all the crashes we had were due to bugs in the dlls we were creating. So please, before reporting a suspected bug in FlowStone do check over your dll code as it is more than likely that this is where the problem will lie.

## Code – Test – Debug Cycle

If you spot a bug or need to change your dll then you can do this without exiting FlowStone or your schematic. However, if you've already connected and run your dll function you will probably find that when you rebuild your dll you get an error saying that the dll file could not be opened for writing.

This is because the dll gets file locked by the software when it loads it into memory. To release it simply trigger the Reset input on the DLL Component.





If the dll is used by other DLL Components in your schematic then it will be released from those too. However, it won't release if you have other schematics open that have locked it or if other DLL components in the same schematic are continuously triggering the Exec input.

Once the dll has been released you can rebuild it and then re-trigger the Exec input to see the results of your changes.

In this way you can rapidly run through the code – test – debug cycle.

# Sharing

The main advantage of the DLL Component is that it allows you to create your own components without compromising on performance safe in the knowledge that any trade secrets are kept hidden from the world. All you need to do is wrap your DLL Component in a module.

There are then two ways you can share your new modules with others.

## Share the DLL Separately

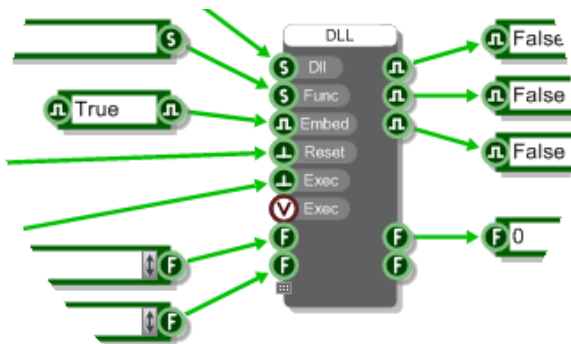
Simply share the dll and your schematic with the module in it. Any recipient may need to modify the path to the dll inside the schematic to match where they end up putting it. If they don't do this then the module will not work.

If you want to share the source code as well then this is the method of distribution you want to go for. It's also the route you'll have to take if your dll uses any other external libraries.

## Embed the DLL

If you're not distributing the source code and want to send the module as a kind of black box then you can embed the dll inside the DLL component.

This is really easy. Go to your DLL component and set the Embed input to True:



That's it – your dll is now embedded in the DLL Component. You can copy and paste it and the dll will be retained. If you want to you can delete the Dll path input value as it's not needed. You might do this if you wanted to hide where the dll was originally located but it's completely up to you.

To share this simply send the module in a schematic.

## Pros and Cons of Embedding

On most occasions embedding the dll will be the go to option. However, there are some disadvantages of embedding that are worth noting. We've listed the pros and cons below.

### PROS

Easy to distribute – there is no separate dll file and no need to worry about file paths.

Independent Globals – you can create global variables to store persistent data that won't be shared.

### CONS

Larger files – because the dll is stored in the schematic the file size will be larger.

No Debug – once embedded you can't debug the dll code (not usually an issue when sharing though).

No shared Globals – you can't create global data to share amongst calls from different dll components.

No external Libraries – you can't use external libraries as these dlls can't be embedded

## Exporting

When you export to EXE or VST you can choose whether to automatically embed any dlls which are currently external to the schematic. So that's dlls for which you haven't set the Embed input to true.

This is handy if you want to keep the dlls external until the point of exporting and saves you having to go through your whole schematic to embed each dll.

Look for the check box below on the VST or EXE export dialog box.

Embed DLLs from DLL components

If any dlls are found which can't be embedded the software will tell you about them.

# 11 Options

CUSTOMISING THE APPLICATION

## The Options Dialog

There are a number of application and schematic level options/features that you can change in order to make the software work the way you want it to. You can get at these by going to the Options menu and selecting one of the seven categories under which the various parameters are organised.

## Application

These options apply to features and settings that apply at application level.

Application

**Startup Options**

<p>Begin using FlowStone with:</p> <p><input type="radio"/> Example schematic</p> <p><input type="radio"/> Most recent schematic</p> <p><input checked="" type="radio"/> Empty schematic</p>	<p>When launching with a file:</p> <p><input checked="" type="radio"/> Open in Focus Mode</p> <p><input type="radio"/> Open normally</p>
--	--

**Most Recent File List**

Display at most  files Clear List

**Auto Recovery**

Save data every  secs Open Recovery Folder...

**Automatic File Compression**

Your schematic files can be compressed automatically to reduce their size. With very large schematics this can begin to affect save times so if you are storing large quantities of wave or image data you can disable this feature

**Check for updates on startup**

Check Now...

### Startup Options

By default the software will load up the example file that ships with the software. This is fine when you first start looking at the software but once you begin making your own creations this behaviour becomes undesirable.

Thankfully you can switch this off. You can choose between the software launching with a blank schematic (the fastest way to load up) or the schematic that you were working on last time you closed down.

If you double-click on a file in Explorer it will usually open in FlowStone for editing. However, you can choose to have double-clicked files open in Focus Mode and run the schematic as if it were an Exe.

### **Most Recent File List**

The most recently used schematics are shown at the bottom of the File menu for rapid access. The default size of this list is 4 items but you can choose to have up to 16 schematics displayed. You can also choose to clear the recently used file list by clicking the Clear List button.

### **Auto Recovery**

Auto Recovery is a handy little feature that saves a copy of your schematic periodically so that if your session should suddenly be closed down due to a crash or loss of power you won't lose your work. Instead, when you next launch FlowStone the file(s) you were editing will be recovered based on the state when they were last saved.

You can switch auto recovery on and off at any time. By default the save period is every 5 seconds. For larger schematics you may want to choose a longer save interval to allow for the increased save time.

Recovery files are saved to the App Data folder on your system. If you need to access this folder you can do so by clicking the Open Recovery Folder button.

### **Automatic File Compression**

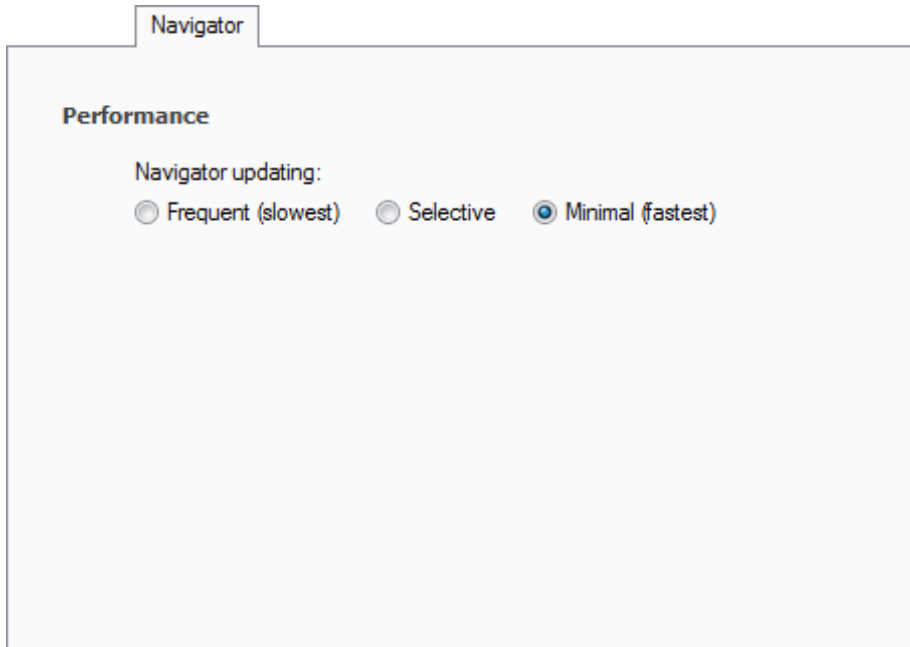
Schematic files are automatically compressed to keep file sizes down. With very big files that contain large amounts of image or sample data you may find that your save times begin to increase. In these cases you can reduce the save time by switching file compression off.

### **Check For Updates On Startup**

FlowStone is updated regularly so the software can check whether you have the latest version and notify you that there is an update waiting. If you prefer not to do this you can switch the behaviour off. You can also manually check for updates by clicking the Check Now button.

Please be assured that your privacy is maintained at all times. No data is transmitted during the update check and only the latest version information is download to your PC from our server.

## Navigator



### Performance

When you change a front panel item like a knob or a slider this often results in values changing in other parts of your schematic. Any modules that are shown in the navigator will be updated to reflect these changes. On some PCs this can be very slow.

You can therefore choose how often the Navigator updates. The default is for minimal updates and often this is all you need. If you want to see the results of your interactions reflected in just the current module in the navigator then choose Selective. To make the Navigator update on every change select Frequent.



## Toolbox

Toolbox

**Adaptive Scrolling**

Allows you to scroll quicker by moving the mouse wheel quicker whilst still giving you fine control when you need it

Brake Slowly 

 Brake Quickly

**Scroll Sensitivity**

How much the toolbox scrolls in response to mouse wheel movements

Small Steps 

 Large Steps

**Filter Selection**

Determines whether tag bar, filter pane and search bar filters are combined or whether selecting a different filter replaces any existing ones

Combine
  Replace  
 (hold SHIFT or use CAPS LOCK to combine)

### Adaptive Scrolling

Enabling this option makes the toolbox scrolling amount adapt to the speed at which you move your mouse wheel. Move the wheel slowly and you'll get small movements. Move it quickly and the toolbox scrolls rapidly, building momentum as it goes.

When you stop scrolling the toolbox will slowly brake to a halt. You can control how quickly or slowly the toolbox braking is applied.

### Scroll Sensitivity

This determines how far the toolbox scrolls in response to each mouse wheel movement.

### Filter Selection

There are numerous ways of filtering the toolbox to zero in on what you're looking for. You can use tags or the search bar and there several types of filter on the filter pane.

This option allows you to choose what happens to the currently applied filter when you click on a different one.

If you choose to **Combine** filters then the existing one(s) will be retained when you click on another one. For example, if you have the Audio tag selected and then click on the Float type filter only components with the Audio tag that also match the Float type will be shown. If you then type some search text this will only search the components that are currently shown.

If you choose to **Replace** filters then then the existing filter will be cleared when you click on another one. Taking the same example as above, if you have the Audio tag selected and then click on the Float type filter then all components matching the Float type will be shown (not just the ones with an Audio tag). If you then type some search text all components matching that search text will be shown.

When the Replace option is selected you can (at any time) hold SHIFT to temporarily combine filters. For search text you can use CAPS LOCK to save you having to hold SHIFT as you type.

## Schematic

### Mouse Behaviour

When drag selecting you can choose whether components will become part of the selection once they are fully enclosed in the drag rectangle or when they touch some part of the area covered by the drag rectangle.

You can decide whether a right-click brings up the standard context menu or if moves you up to the parent when inside a module.

### Zoom Level

The default zoom level is defined as a number of pixels. This represents the size of a single grid square so a higher number will result in a default zoom that appears magnified. If you click Use Current you'll get the zoom level for the current module.

### Auto Linking

You can decide how close components have to be before the software will suggest an automatic link. You can also choose whether links are then created after a short time or if the software waits for you to right-click to accept a suggested link.

### Links On Top

Check this box if you want links to appear on top of components in your schematic (applies to the schematic as a whole).

## Modules

Modules

**Animation**

Enable module animation

Allows you to animate module resizing as you change between front panel, minimized state and properties

Animation Speed:

Slower  Faster

**Properties**

Auto Close

Select this option if you want module properties to close automatically when the module is deselected

### Animation

You can choose whether to enable or disable module animation for when you minimize or maximize a module or open/close the properties panel.

### Properties

By default a module's properties panel will close as soon as you click away (provided it isn't pinned open). You can switch this off so that the properties remain open until you close them.

## Export

Export

**Target Folders**

Exported VST plugins will be saved here:

C:\Program Files\Steinberg\VstPlugins

Change...

Exported standalone EXEs will be saved here:

C:\Users\Guest\Desktop

Change...

Always overwrite exports without asking

**SSE/2 Support**

Save support for SSE and SSE2 in schematic files  
 Select this option if you will be distributing your plugins to other parties who may have a processor with different SSE capabilities from your own

### Target Folders

When you export standalone exes they are saved to a particular folder. You can set this here.

### SSE/2 Support

The PC on which you generate your exports may support SSE2 or it may just support SSE. When you export the software saves code that is optimised for the SSE capabilities of your own PC. If you then give the export to someone whose PC has different SSE capabilities from you the software needs to make adjustments on loading so that the export will work.

These adjustments can slow down the loading process. To avoid this you can choose to include support for all SSE setups at export time. This way there is no reduction in loading speed when exports are used on PCs with different SSE capabilities.

**Advanced**

Advanced


  

**R&D Components**

Show any research and development (R&D) components in the toolbox

Be aware that such components are still under development and may change or be dropped at any time

**Link Curvature**

Straight  Curved

**Code Compilation**

Compile as you type in the code component

Only enable this if you need to see or hear the effect of your code changes as you are making them

Maximum code size:  bytes

**SSE/2 Support**

Save support for SSE and SSE2 in schematic files (results in slower saves)

Select this option if you will be distributing your schematics to other parties who may have a processor with different SSE capabilities from your own

**Floating Point Numbers**

Display floats as 32 bit binary representations

**Ruby Timeout**

Interrupt Ruby operations after:  milliseconds

**R&D Components**

We will periodically offer some components on a trial basis to gather feedback or for testing. You can decide whether to show these components by checking the box.

### **Link Curvature**

When you bend links they take a curved form. You can decide how curved you want them to be from highly curved to perfectly straight lines. Note that this affects the way that links look in any schematic you open.

### **Code Compilation**

In order to increase performance code compilation is switched off while you are typing in a code component. However, sometimes you want to be able to see the effects of your code as you type.

### **SSE/2 Support**

The PC on which you create your schematics may support SSE2 or it may just support SSE. When you save a schematic the software saves code that is optimised for the SSE capabilities of your own PC. If you then give the schematic to someone whose PC has different SSE capabilities from you the software needs to make adjustments on loading so that the schematic will work.

These adjustments can slow down the loading process. To avoid this you can choose to include support for all SSE setups at save time. This way there is no reduction in loading speed when schematics are used on PCs with different SSE capabilities.

You only need to enable this option for schematics if you intend to distribute them. One side affect of enabling this option is that your save times are increased.

### **Floating Point Numbers**

In computing floating point numbers cannot represent every possible number using 32bits. The system therefore has to approximate some numbers.

For example, 0.01 is represented as  $9.9999998 \times 10^{-3}$

These small inaccuracies can sometimes lead to unexpected results in calculations because FlowStone will display 0.01 as 0.01 when in fact the number being used is of course the floating point approximation.

These situations are quite rare but if you are doing precise calculations where these inaccuracies may be significant in your results then you can choose to display the underlying representations instead so that all your calculations are completely transparent.

**Ruby Timeout**

When code in a Ruby component is executed, FlowStone waits until it completes before allowing the schematic to proceed. It is sometimes possible to inadvertently write code that performs an infinite loop or a very long calculation. To protect against this FlowStone has a maximum time that it allows a Ruby component before interrupting the calculation and switching the component off as a precaution.

This is called the Ruby Timeout. It defaults to 8000 milliseconds but we allow you to change it here in case you have some Ruby code that would not hang the software but that takes longer than 8 seconds.