

Slint Documentation

Snapshot of <https://docs.slint.dev/latest/docs/sl/>

Welcome to Slint

Seamlessly build elegant GUIs for Embedded, Desktop, and Mobile

The documentation is split into several sections:

Guide

Get up and running with the tooling for Slint development including the IDE integration via the Slint Language Server (LSP) and Slint Viewer.

Learn all the key concepts of Slint to understand and use the Slint language.

Reference

Browse the API reference for all aspects of the Slint language: The elements, properties, functions, callbacks, namespaces, as well as the std-widgets library. A set of cross platform components that can be used to build desktop applications.

Tutorial

Learn by example and see how to structure a simple application via a step-by-step tutorial that walks you through the creation of a simple memory game example.

Language Integrations

API reference for the Rust, C++, JavaScript, and Python versions of Slint.

Documentation features

The documentation includes a lot of code snippets. The language hint lets you know what language the snippet is written in.

Some snippets of Slint code are interactive. You can click the `run` button to run the snippet in a web based live-editing tool called [SlintPad↗](#).

To easily copy the text of a snippet use the copy button.

Examples that want to help focus on a specific part of the code will have highlights. They are only a documentation feature and you won't see this kind of highlight when writing your own code.

Get in touch

Chat

Discussions

Report Bugs

Email

Next

Visual Studio Code

© 2025 SixtyFPS GmbH

Reference Overview

This section contains the API reference for all aspects of the Slint language. All the elements, properties, functions, callbacks and namespaces. It also contains the API reference for the `std-widgets` library. A set of cross platform components that can be used to build desktop applications.

Next

Primitive Types

© 2025 SixtyFPS GmbH

Intro

This tutorial introduces you to the Slint framework in a playful way by implementing a memory game. It combines the Slint language for the graphics with the game rules implemented in C++, Rust, NodeJS, or Python.

The game consists of a grid of 16 rectangular tiles. Clicking on a tile uncovers an icon underneath. There are 8 different icons in total, so each tile has a sibling somewhere in the grid with the same icon. The objective is to locate all icon pairs. The player can uncover two tiles at the same time. If they aren't the same, the game obscures the icons again. If the player uncovers two tiles with the same icon, then they remain visible - they're solved.

This is how the game looks in action:

Next

Getting Started

© 2025 SixtyFPS GmbH

Visual Studio Code

If you are new to Slint and want to quickly get started and learn the basics, we recommend using Visual Studio Code (VS Code). VS Code is popular, free and thanks to the Slint extension it's also the easiest to get started with.

Note

We support many other tools and editors, see [here↗](#).

Setting Up VS Code

- 1 **Install VS Code.** Download it [here↗](#).

- 2 **Install the Slint extension.** Find it [here↗](#).
- 3 **Create a new project based on a Slint template.** This is done via the command palette (CTRL+Shift+P) or on MacOS (CMD+Shift+P). Start typing 'slint' and from the options select 'Slint: Create New Project from Template'.
- 4 **Choose your language.** Then select from the list of languages.
- 5 **Choose a folder to save the project in.**
- 6 **Name the project.** Give the project a name and now a new project will be created in the selected folder based on a simple template to get you started.

Previous

Overview

Next

Figma Variable Export

© 2025 SixtyFPS GmbH

Figma Variable Export

Introduction

The “Figma to Slint” plugin bridges the gap between Figma designs and Slint user interfaces. It provides two primary functionalities:

- 1. Object Inspection:** Allows developers to select any object on the Figma canvas and view its properties (like dimensions, colors, fonts, etc.) formatted as a Slint code snippet. This is useful for quickly translating visual design elements into Slint code. When paired with the variable export, this inserts variables into the code, if they are used in the Figma file.
- 2. Variable Export:** Enables the reliable export of design tokens (variables for colors, numbers, strings, and booleans) defined in Figma directly into `.slint` files. This feature fully supports Figma’s modes, allowing for themeable designs (e.g., light and dark modes) and generates the necessary Slint structs, enums, and global instances to use these tokens seamlessly in your Slint application.

Setting up the Figma Inspector

First of all, find and enable the “Figma to Slint” plugin in the Figma Plugin Manager. The inspector currently has 2 functions. One is to inspect individual Figma objects for their properties, and the other is to export the variables defined in Figma so they are useable in Slint directly.

Key Features of Variable Export

- **Type Conversion:**

The plugin maps Figma variable types to their corresponding Slint types as follows:

Figma Variable Type	Slint Type
Color	brush
Float	length
String	string
Boolean	bool

- **Mode Support:** For a figma file where Mode MyCollection has modes light and dark , the exporter will create a Slint enum for mode selection (e.g., MyCollectionMode { light, dark }). The file also contains a Scheme struct (MyCollection-Scheme) representing the structure of variables. A Scheme–Mode struct (MyCollection–Scheme–Mode) holds instances of the Scheme for each mode. And finally a global instance (my_collection) which contains:
 - mode : An instance of Scheme–Mode holding the resolved values for each mode.
 - current–mode : An in–out property using the mode enum to control the active mode.
 - current : An out property dynamically selecting the correct scheme instance based on current–mode .
- **Reliable Variable Resolution:** Automatically resolves all variable aliases (references) to their concrete values, ensuring consistent and predictable output regardless of how variables reference each other.
- **Hierarchy Generation:** Interprets / in Figma like colors/background/primary to generate nested Slint structs so the corresponding variables are nested for dot notation in Slint, thus the above example turns into colors.background.primary
- **Robust Mode Handling:** Intelligently matches Figma modes to ensure all variables export with their correct values, even when mode IDs don't match perfectly between collections and variables.

- **Name Sanitization:** Cleans up collection, variable, and mode names to be valid Slint identifiers (e.g., converting spaces and special characters to underscores, handling leading digits).
- **Export Options:**
 - **Separate Files:** Exports each Figma collection into its own `.slint` file. Creates `<collection_name>.slint` files with automatic cross-collection imports when needed.
 - **Single File:** Combines all collections into a single `design-tokens.slint` file for simpler project integration.
- **README Generation:** Creates a `README.md` file alongside the export, summarizing exported collections, renamed variables, and any warnings.

Usage

1. Inspector:

- Bring up the plugin UI and select any object you'd like to inspect.

2. Variables:

- To see and export variables, check the “Use Figma Variables” checkbox in the plugin UI.
- Now when selecting objects to inspect you should see variable names instead of resolved values (if they are assigned)

3. Export: Click the “Export” button and choose:

- `Separate Files Per Collection...` : Recommended for organization. Creates `<collection_name>.slint` files.
- `Single Design-Tokens File...` : Creates `design-tokens.slint` for simpler project integration.

4. Integrate:

- Place the generated `.slint` file(s) in your Slint project.

Design-Token File Structure

1. Import

```
// If separate files:  
import { Colors } from "colors.slint";  
import { Spacing } from "spacing.slint";
```

slint

```
// If single file:  
import { Colors, Spacing } from "design-tokens.slint";
```

slint

2. Use variables

```
// Use the tokens:  
MyComponent := Rectangle {  
    background: Colors.current.background.primary; // Access via .current  
    height: Spacing.medium; // Access directly if single-mode or not mode  
}
```

slint

3. Mode Switching:

(For multi-mode collections) Modify the `current-mode` property of the imported global:

```
// Example: In your main component's logic  
init => {  
    // Set initial mode  
    Colors.current-mode = ColorsMode.dark;  
}  
// Or in response to a user action:  
clicked => {  
    Colors.current-mode = (Colors.current-mode == ColorsMode.light) ? Col
```

slint

- All properties accessed via `<collection_name>.current.*` will automatically update.

Naming Conventions & Sanitization

- **Hierarchy:** Figma uses `/` in variable names (e.g., `radius/small`, `font/body/weight`). The export will use these to create nested Slint structs for code completion clarity.
- **Sanitization:** Collection names, variable names (path segments), and mode names are automatically sanitized:
 - Spaces and invalid characters (`&` , `+` , `:` , `-` , etc.) are typically converted to `-` for collection/struct names and `_` for property/enum names.
 - Leading/trailing invalid characters are removed.
 - Names starting with a digit are prefixed with `_` or `m_`.
 - Any variable named exactly `mode` at the root of a collection is renamed to `mode-var` in the Slint output to avoid conflicts with the generated scheme `mode` property.

Example

Variable `Color & Shade` (only `#hex` values) in a collection named `Color Primitives` would get turned into `color-primitives.color_and_shade_only_hex_values`

Previous

Visual Studio Code

Next

Slint Language

© 2025 SixtyFPS GmbH

Slint Language

This following section gives you an insight into the thinking behind the language and the core concepts it's made up from.

As covered in the video above, the Slint declarative UI language is designed to be a simple, yet powerful way to create any user interface you can imagine.

```
Text {  
    text: "Hello World!";  
    font-size: 24px;  
    color: #0044ff;  
}
```

slint

This example shows the core of how Slint works. Use elements with their name followed by open and closed braces, e.g. `Text {}`. Then within the braces customize it with properties e.g. `font-size: 24px`.

To nest elements inside each other place them within the parents braces. For example the following `Rectangle` has a `Text` element as its child.

```
Rectangle {  
    width: 150px;  
    height: 60px;  
    background: white;  
    border-radius: 10px;  
  
    Text {  
        text: "Hello World!";  
        font-size: 24px;  
        color: black;  
    }  
}
```

slint

The final core part are binding expressions:

```
1  property <int> counter: 0;  
2  
3  Rectangle {  
4      width: 150px;  
5      height: 60px;  
6      background: white;  
7      border-radius: 10px;  
8  
9      Text {  
10         text: "Count: " + counter;  
11         font-size: 24px;  
12         color: black;  
13     }  
14 }
```

slint

```

15   TouchArea {
16     clicked => {
17       counter += 1;
18     }
19   }
20 }
```

In this example a property called `counter` is declared. Then the `Rectangle` has a `TouchArea` inside, that automatically fills its parent and responds to clicks or taps. On click it increments `counter`. Here is where the magic of fine grained reactivity comes in.

The `Text` element's `text` property depends on the `counter` property. When `counter` changes, the UI is automatically updated. There's no need to opt in. In Slint every expression is automatically re-evaluated. However this is done in a performant way that only updates expressions where the dependencies have changed.

Slint makes it trivial to create your own components by composing together the built in elements or other components. `Text` and `Rectangles` can become buttons. Buttons and input fields can become forms or dialogs. Forms and dialogs can become Views. And finally Views combine to become applications.

```

export component MyView {
  MyDialog {
    title: "Can UI Development Be Easy?";

    MyButton {
      text: "Yes";
    }
  }
}
```

slint

With practice you can reduce any level of complexity to simple, maintainable UI components.

Why Slint?

The objective of the concepts section of the guide is to give you an overview of the language from a high level. If you want to dive straight in the details then check out the [coding section](#) and the language reference sections.

The Slint language describes your application's User Interface in a declarative way.

A User Interface is quite different from abstract code. It's made up of text, images, colors, animations, and so on. Even though it's a mirage, the buttons and elements do not really exist in the physical world, it's meant to look and behave as if it does. Buttons have pressed effects, lists can be flicked and behave as if they had real inertia. When being designed they are described in terms of UI components, forms, and views.

Meanwhile the world of code is a quite different abstraction. It's made up of functions, variables, and so on. Even when some aspects of a UI exist such as buttons and menus they also go hand in hand with a lot of code to manage the implementation details.

```
const button = document.createElement('button');  
button.textContent = 'Click me';  
document.body.appendChild(button);
```

js

Take this simple web example. A button is created. Then a property to show 'Click me' text is set. At this point technically the button exists, but it won't show up as its not attached to anything. So the final line makes it a child of the main view.

```
const buttonWithListener = document.createElement('button');  
buttonWithListener.textContent = 'Click me';  
buttonWithListener.addEventListener('click', () => {  
    console.log('Button clicked!');  
});  
document.body.appendChild(buttonWithListener);
```

js

In this second example a button is created and an event listener is added. Within that is a callback function to log out that the button is pressed. It's a lot of code to do simple things.

It's quite abstract. For example once a few more buttons and components are added its almost impossible to be able to think how the real interface would look. It's hard to think

about the design of a UI using this kind of code.

It's also too complex to edit without understanding how to code. This means UI designers cannot work hands on to ensure all their design intent is implemented. They are forced to use other tools and frameworks to create prototypes and design guides that may look or behave differently to the actual UI implementation. It doesn't have to be this way.

Declarative Style

There have been attempts to make describing a UI in code more declarative. For example [React ↗](#) and [SwiftUI ↗](#).

```
function ContentView() {  
    return (  
        <p style={{ fontSize: '2rem', color: 'green' }}>  
            Hello World  
        </p>  
    );  
}
```

```
struct ContentView: View {  
    var body: some View {  
        Text("Hello World")  
            .font(.title)  
            .foregroundColor(.green)  
    }  
}
```

These languages take normal code and let it be used in a declarative way. But it's still functions with arguments that act as properties. It is simpler and behavior such as parent child relationships can be inferred.

For Slint, instead of tweaking a normal language to be more declarative, we created a pure declarative language from the ground up.

The Business Logic Problem

One attempt to solve the problem of describing a UI in code has been to create a separate static markup language. Platforms such as Android and [WPF](#) have done this. The UI is described in an XML-like format. While the rest of the code is written in both a separate language and also separate files. Known as a code behind file. The issue here is that XML, despite claims to the contrary, is not human readable or editable. It's also too static and rigid and it can be frustrating to jump between the UI file and the separate code behind file to describe the behavior of the UI.

Meanwhile the React web framework has solved this problem by using JSX. HTML, CSS and JavaScript can be mixed together in a single file. On one hand this is great as it means you can use the same language to describe the UI layout and behavior. On the other hand there is no limit to what code you can put in a JSX file. Logic for handling network requests, processing data and pretty much anything quickly ends up mixed in with UI code. It leads to the issue where it can be fast to create an initial application, but it becomes so hard to maintain it that it's too slow or costly to evolve the application.

A True Declarative UI Language

Slint provides a declarative language to describe an application's user interface

```
1  Rectangle {  
2      Button {  
3          text: "Click me!";  
4          clicked => {  
5              debug("Button clicked!");  
6          }  
7      }  
8  }
```

slint

If you can understand this Slint code — you've already grasped a majority of how simple to use the language is. On line 2 we declare a `Button`. On line 3 we set its `text` property to `"Click me!"` and on line 4 we set a callback to print out `"Button clicked!"` to the console via the built in `debug` function.

The Slint compiler will take this code and see that it needs to generate a Rectangle that has a Button as a child. Similarly, when the clicked callback activates, it will run the `debug` function. There is no need to think about component and event listener life cycles.

With the Slint language you can think much more closely to how a user interface looks and behaves. Instead of describing the ‘how’, the how should the details be implemented in traditional code, you “declare” how the interface should look and behave. Hence Slint being a ‘declarative UI language’.

It’s not only simpler for software developers to use, it’s also now something designers can potentially edit or more easily contribute to.

Slint isn’t the first declarative UI language, but it takes advantage of being able to learn from earlier more complex attempts that hinted at the potential of a declarative UI language to finalize into a modern and complete system.

At first glance it has the simplicity of a static markup language, but with a modern take that removes things like angle brackets and tags. But also dynamic features such as complex property expressions, functions, callbacks, and automatic reactivity. However these can only be used in the context of what helps a UI component. If you want to make network requests, process data, or many other things that count as ‘business logic’ then those have to live in separate files written in Rust, C++, JavaScript, etc. Slint allows you to express any UI and only the UI.

It then provides a system of adapters to allow the business logic side of the app to easily communicate with the UI and visa versa.

Previous

Figma Variable Export

Next

Reactivity

Reactivity

Reactivity

Reactivity is core concept within Slint. It allows the creation of complex dynamic user interfaces with a fraction of the code. The following examples will help you understand the basics of reactivity.

```
export component MyComponent {  
    width: 400px; height: 400px;  
  
    Rectangle {  
        background: #151515;  
    }  
  
    ta := TouchArea {}  
  
    myRect := Rectangle {  
        x: ta.mouse-x;  
        y: ta.mouse-y;  
        width: 60px;  
        height: 60px;  
        background: ta.pressed ? orange : skyblue;  
        Text {  
            x: 5px; y: 5px;  
            text: "x: " + myRect.x / 1px;  
            color: white;  
        }  
        Text {  
            x: 5px; y: 15px;  
            text: "y: " + myRect.y / 1px;  
            color: white;  
        }  
    }  
}
```

slint

```

        x: 5px; y: 20px;
        text: "y: " + myRect.y / 1px;
        color: white;
    }
}

}

```

As the name suggests, Reactivity is all about parts of the user interface automatically updating or 'reacting' to changes. The above example looks simple, but when run it does several things:

- The `Rectangle` will follow the mouse around as you move it.
- If you `click` anywhere the `Rectangle` will change color.
- The `Text` elements will update their text to show the current position of the `Rectangle`.

The 'magic' here is built into the Slint language directly. There is no need to opt into this or define specific stateful items. The `Rectangle` will automatically update because its `x` and `y` properties are bound to the `mouse-x` and `mouse-y` properties of the `TouchArea` element. This was done by giving the `TouchArea` a name to identify it `ta` and then using the name in what Slint calls an `expression` to track values. It's as simple as `x: ta.mouse-x;` and `y: ta.mouse-y;`. The `mouse-x` and `mouse-y` properties are built into the `TouchArea` and automatically update as the cursor moves over them.

The `TouchArea` also has a `pressed` property that is only `true` when the cursor is pressed or clicked down. So the ternary expression `background: ta.pressed ? orange : white;` will change the background color of the `Rectangle` to orange when `ta.pressed` is true, or white when it isn't.

Similarly the 2 text items are updating by tracking the rectangle's `x` and `y` position.

Overwritten Bindings

Bindings in Slint are only active as long as the property is assigned using a binding expression (`x: other.value`) or a two-way binding (`<=>`). If a property's value is later changed using an imperative assignment in code (e.g. `foo.bar = 42;`), the original binding

is broken. From that point on, the property will no longer react to changes in the values it was previously bound to. If needed, the property can still be updated again later through another assignment, but the automatic reactivity is lost.

This behavior also applies to most built-in `in-out` properties, such as the `text` property of a `TextInput`. When the user interacts with the widget, for example by typing into the input, this is considered an imperative assignment that will break any existing binding to that property.

To maintain reactivity in such cases, you can use a two-way binding (`<=>`), or you can make use of the `changed` callback to track and respond to property updates.

Performance

From a performance perspective, Slint works out what properties are changed. It then finds all the expressions that depend on that value. These dependencies are then re-evaluated based on the new values and the UI will update.

The re-evaluation happens lazily when the property is queried.

Internally, a dependency is registered for any property accessed while evaluating a binding. When a property changes, the dependencies are notified and all dependent bindings are marked as dirty.

Property Expressions

Expressions can vary in complexity:

```
// Tracks the `x` value of an element called foo
x: foo.x;

// Tracks the value, but sets it to 0px or 400px based on if
// foo.x is greater than 400px
x: foo.x > 100px ? 0px : 400px;

// Tracks the value, but clamps it between 0px and 400px
```

slint

```
x: clamp(foo.x, 0px, 400px);
```

As the last example shows functions can be used as part of a property expression. This can be useful for when an expression is too complex to be readable or maintained as a single line.

```
export component MyComponent {  
  width: 400px; height: 400px;  
  
  pure function lengthToInt(n: length) -> int {  
    return (n / 1px);  
  }  
  
  Rectangle {  
    background: #151515;  
  }  
  
  ta := TouchArea {}  
  
  myRect := Rectangle {  
    x: ta.mouse-x;  
    y: ta.mouse-y;  
    width: 60px;  
    height: 60px;  
    background: ta.pressed ? orange : skyblue;  
    Text {  
      x: 5px; y: 5px;  
      text: "x: " + lengthToInt(myRect.x);  
      color: white;  
    }  
    Text {  
      x: 5px; y: 20px;  
      text: "y: " + lengthToInt(myRect.y);  
      color: white;  
    }  
  }  
}
```

```
}
```

Here the earlier example was updated to use a function to convert the length to an integer. This also truncates the x and y values to be more readable i.e. '4' instead of '4.124488'.

Purity

For any reactive system to work well, evaluating a property shouldn't change any observable state but the property itself. If this is the case, then the expression is "pure", otherwise it's said to have side-effects. Side-effects are problematic because it's not always clear when they will happen: Lazy evaluation may change their order or affect whether they happen at all. In addition, changes to properties during their binding evaluation due to a side-effect may result in unexpected behavior.

For this reason, bindings in Slint **must** be pure. The Slint compiler enforces code in pure contexts to be free of side effects. Pure contexts include binding expressions, bodies of pure functions, and bodies of pure callback handlers. In such a context, it's not allowed to change a property, or call a non-pure callback or function.

Annotate callbacks and public functions with the `pure` keyword to make them accessible from property bindings and other pure callbacks and functions.

The purity of private functions is automatically inferred. You may declare private functions explicitly "pure" to have the compiler enforce their purity.

```
export component Example {  
    pure callback foo() -> int;  
    public pure function bar(x: int) -> int  
    { return x + foo(); }  
}
```

slint

Two-Way Bindings

Create two-way bindings between properties with the `<=>` syntax. These properties will be linked together and always contain the same value. Also known as bidirectional or bi-

directional bindings.

The right hand side of the `<=>` must be a reference to a property of the same type. The property type is optional with two-way bindings, it will be inferred if not specified. The initial value of a linked property will be the value of the right hand side of the binding. The two linked properties must be compatible in terms of input/output.

```
export component Example {  
    in property<brush> rect-color <=> r.background;  
    // It's allowed to omit the type to have it automatically inferred  
    in property rect-color2 <=> r.background;  
    r:= Rectangle {  
        width: parent.width;  
        height: parent.height;  
        background: blue;  
    }  
}
```

slint

Previous

[Slint Language](#)

Next

[Reactivity vs React.js](#)

Reactivity vs React.js

Comparison to React.js

The following sections are for those coming from or are familiar with the [React.js](#) web framework. We're going to compare patterns from React.js based app development with Slint.

Note

There is no web browser as part of Slint to render the UI. There is no DOM or shadow DOM.

Component Life Cycle Management

React.js has a model where on a state change a component is destroyed and recreated. By default this will also include the destruction of all child components of an element and these then all need to be recreated. To manage performance, careful use of `useMemo()` and `useCallback()` are needed to avoid unnecessary re-renders. Even though the need for this has been reduced via the React Compiler it's still necessary to understand this model to understand how a React app behaves.

Slint is much simpler and uses fine-grained reactivity: Components update, but they aren't destroyed and recreated. There is no equivalent of `useMemo()` and `useCallback()` as they are unnecessary.

State

React.js refers to properties that update and re-render the component as state. They are opt-in and by default are not tracked.

```
function Counter() {  
    const [count, setCount] = useState(0);  
    return <button onClick={() => {  
        setCount((currentCount) => currentCount + 1)  
    }}>{count}</button>;  
}
```

jsx

```
import { Button } from "std-widgets.slint";  
  
export component Counter {  
    property <int> count: 0;  
    Button {  
        text: count;  
        clicked => {  
            count += 1;  
        }  
    }  
}
```

slint

The classic counter example also shows key differences. First the `count` property is declared and a `count` value and `setCount()` function are deconstructed from the `useState` hook. Note that 'count' cannot be directly accessed and must be updated via `setCount()`.

The counter button is then used to update the `count` property and to ensure it's correctly updated must rely on the `currentCount` value returned by `setCount()` is used to update the values. As using `setCount(count + 1)` can cause issues in more complex scenarios where the state is updated later.

While the Slint example may not look much simpler, it does the same job and has less gotchas. As everything in Slint is reactive by default, the property is declared in one single way. The language has strong types and for numbers has both `float`s and `int`s. The

property can also be safely modified directly which also in this example allows the use of the `+≡` operator.

Previous

Reactivity

Next

The ` .slint` File

© 2025 SixtyFPS GmbH

The `*.sling` File

You write user interfaces in the Slint language and saved in files with the `.sling` extension.

Each `.sling` file defines one or several components. These components declare a tree of elements. Components form the basis of composition in Slint. Use them to build your own reusable set of UI controls. You can use each declared component under its name as an element in another component.

Below is an example of components and elements:

```
component MyButton inherits Text {  
    color: black;  
    // ...  
}  
  
export component MyApp inherits Window {  
    preferred-width: 200px;  
    preferred-height: 100px;  
    Rectangle {  
        width: 200px;  
        height: 100px;  
        background: green;  
    }  
    MyButton {  
        x:0;y:0;  
        text: "hello";  
    }  
    MyButton {  
        y:0;  
    }  
}
```

sling

```

        x: 50px;
        text: "world";
    }
}

```

Both `MyButton` and `MyApp` are components. `Window` and `Rectangle` are built-in elements used by `MyApp`. `MyApp` also re-uses the `MyButton` component as two separate elements.

Elements have properties, which you can assign values to. The example above assigns a string constant “hello” to the first `MyButton` ‘s `text` property. You can also assign entire expressions. Slint re-evaluates the expressions when any of the properties they depend on change, which makes the user-interface reactive.

You can name elements using the `:=` syntax:

```

component MyButton inherits Text {
    // ...
}

export component MyApp inherits Window {
    preferred-width: 200px;
    preferred-height: 100px;

    hello := MyButton {
        x:0;y:0;
        text: "hello";
    }

    world := MyButton {
        y:0;
        text: "world";
        x: 50px;
    }
}

```

Note

Names have to be valid [identifiers](#).

Some elements are also accessible under pre-defined names:

- `root` refers to the outermost element of a component.
- `self` refers to the current element.
- `parent` refers to the parent element of the current element.

These names are reserved and you can't re-define them.

Comments

Comments are lines of code that are ignored by the Slint compiler. They are used to explain the code or to temporarily disable code.

Single Line Comments

Single line comments are denoted by `//` and are terminated by a new line.

```
// Amazing text! This is a comment
```

slint

Multi Line Comments

Multi line comments are denoted by `/*` and `*/` and are terminated by a new line.

```
/*
    This is a multi line comment.
    It can span multiple lines.
*/
```

slint

Elements and Components

The core part of the Slint language are elements and components. Technically they are the same thing so once you know how to declare and use one you know the other. Elements are the basic building blocks that are part of the Slint Language, while components (also known as widgets) are larger items that are built up from multiple elements and properties.

If you have come from other languages such as HTML or React you might be used to opening and closing tags as well as self closing tags.

```
<!-- opening and closing tag -->  
<Button>Hello World</Button>  
<!-- self closing tag -->  
<img/>
```

html

Slint simply has a single way to declare an item the `element-name` followed by a set of curly braces `{}` that contain the properties of the element.

```
// valid  
Text {}  
  
Text {  
}  
// Valid, but considered bad Slint practice  
Text  
{  
}  
  
// Not valid due to terminating semicolon  
Text {};
```

lint

Note

The Slint tooling provides a code formatter that enforces what is considered good practice.

If you are new to coding then you can make friends with fellow developers by discussing aspects of code formatting you don't like. It's a type of small talk developers love and

I appreciate.

The Root Element

```
component MyApp {  
  Text {  
    text: "Hello World";  
    font-size: 24px;  
  }  
}
```

slint

To be a valid Slint file the root element must be a component. Then inside the component you can declare any number of elements. This is explained in more detail later, it's not important to understand at this point.

Properties

Properties are the values that are assigned to an element. They are set using the `property-name: value;` syntax.

Identifiers

Identifiers can be composed of letter (`a-zA-Z`), of numbers (`0-9`), or of the underscore (`_`) or the dash (`-`). They can't start with a number or a dash (but they can start with underscore) The underscores are normalized to dashes. Which means that these two identifiers are the same: `foo_bar` and `foo-bar` .

Conditional Elements

The `if` construct instantiates an element only if a given condition is true. The syntax is `if condition : id := Element { ... }`

```
export component Example inherits Window {  
    preferred-width: 50px;  
    preferred-height: 50px;  
    if area.pressed : foo := Rectangle { background: blue; }  
    if !area.pressed : Rectangle { background: red; }  
    area := TouchArea {}  
}
```

slint

Modules

Components declared in a `.slint` file can be used as elements in other `.slint` files, by means of exporting and importing them.

By default, every type declared in a `.slint` file is private. The `export` keyword changes this.

```
component ButtonHelper inherits Rectangle {  
    // ...  
}  
  
component Button inherits Rectangle {  
    // ...  
    ButtonHelper {  
        // ...  
    }  
}  
  
export { Button }
```

slint

In the above example, `Button` is accessible from other `.slint` files, but `ButtonHelper` isn't.

It's also possible to change the name just for the purpose of exporting, without affecting its internal use:

```
component Button inherits Rectangle {  
    // ...  
}  
  
export { Button as ColorButton }
```

slint

In the above example, `Button` isn't accessible from the outside, but is available under the name `ColorButton` instead.

For convenience, a third way of exporting a component is to declare it exported right away:

```
export component Button inherits Rectangle {  
    // ...  
}
```

slint

Similarly, components exported from other files may be imported:

```
import { Button } from "./button.slnt";  
  
export component App inherits Rectangle {  
    // ...  
    Button {  
        // ...  
    }  
}
```

slint

In the event that two files export a type under the same name, then you have the option of assigning a different name at import time:

```
import { Button } from "./button.slnt";  
import { Button as CoolButton } from "../other_theme/button.slnt";  
  
export component App inherits Rectangle {  
    // ...  
    CoolButton {} // from other_theme/button.slnt
```

slint

```
    Button {} // from button.slint  
}
```

Elements, globals and structs can be exported and imported.

It's also possible to export globals (see [Global Singletons](#)) imported from other files:

```
import { Logic as MathLogic } from "math.slint";  
export { MathLogic } // known as "MathLogic" when using native APIs to access
```

sling

Module Syntax

The following syntax is supported for importing types:

```
import { MyButton } from "module.sling";  
import { MyButton, MySwitch } from "module.sling";  
import { MyButton as OtherButton } from "module.sling";  
import {  
    MyButton,  
    /* ... *,  
    MySwitch as OtherSwitch,  
} from "module.sling";
```

sling

The following syntax is supported for exporting types:

```
// Export declarations  
export component MyButton inherits Rectangle { /* ... */ }  
  
// Export lists  
component MySwitch inherits Rectangle { /* ... */ }  
export { MySwitch }  
export { MySwitch as Alias1, MyButton as Alias2 }  
  
// Re-export types from other module  
export { MyCheckBox, MyButton as OtherButton } from "other_module.sling";
```

sling

```
// Re-export all types from other module (only possible once per file)
export * from "other_module.slint";
```

Component Libraries

Splitting your code base into separate module files promotes re-use and improves encapsulation by allow you to hide helper components. This works well within a project's own directory structure. To share libraries of components between projects without hardcoding their relative paths, use the component library syntax:

```
import { MySwitch } from "@mylibrary/switch.slint";
import { MyButton } from "@otherlibrary";
```

slint

In the above example, the `MySwitch` component will be imported from a component library called `mylibrary`, in which Slint looks for the `switch.slint` file. Therefore `mylibrary` must be declared to refer to a directory, so that the subsequent search for `switch.slint` succeeds. `MyButton` will be imported from `otherlibrary`. Therefore `otherlibrary` must be declared to refer to a `.sln` file that exports `MyButton`.

The path to each library, as file or directory, must be defined separately at compilation time. Use one of the following methods to help the Slint compiler resolve libraries to the correct path on disk:

- When using Rust and `build.rs`, call [`with_library_paths`](#) to provide a mapping from library name to path.
- When using C++, use `LIBRARY_PATHS` with [`slint target sources`](#).
- When invoking the `slint-viewer` from the command line, pass `-Lmylibrary=/path/to/my/library` for each component library.
- When using the VS Code extension, configure the Slint extension's library path using the `Slint: Library Paths` setting. Example below:

```
"slint.libraryPaths": {
    "mylibrary": "/path/to/my/library",
    "otherlibrary": "/path/to/otherlib/index.sln",
```

json

},

This can also be edited in the `.vscode/settings.json` file committed to your repository. Relative paths are resolved against the workspace root.

- With other editors, you can configure them to pass the `-L` argument to the `slint-lsp` just like for the `slint-viewer`.

Previous

Reactivity vs React.js

Next

Properties

© 2025 SixtyFPS GmbH

Properties

All elements have properties. Built-in elements come with common properties such as color or dimensional properties.

Assigning bindings

You can assign values or entire [Expressions](#) to them:

```
export component Example inherits Window {  
    // Simple expression: ends with a semi colon  
    width: 42px;  
    // or a code block (no semicolon needed)  
    height: { 42px }  
}
```

slint

Properties in Slint can be assigned using either a simple **expression**, which ends with a semicolon (;), or a **code block**, enclosed in { ... } , which does not require a semicolon. These two forms are interchangeable for simple values, but a block is useful when you need multiple statements.

For example, these two bindings are equivalent:

```
background: touch-area.is-pressed ? red : blue;  
  
background: {  
    if (touch-area.is-pressed) {  
        return red;
```

slint

```
    } else {
        return blue;
    }
}
```

Both forms are reactive, and Slint will automatically track dependencies used within the expression or block.

The default value of a property is the default value of the type. For example a boolean property defaults to `false`, an `int` property to zero, etc.

Declaring Properties

In addition to the existing properties, define extra properties by specifying the type, the name, and optionally a default value:

```
export component Example {
    // declare a property of type int with the name `my-property`
    property<int> my-property;

    // declare a property with a default value
    property<int> my-second-property: 42;
}
```

slint

Annotate extra properties with a qualifier that specifies how the property can be read and written:

- **private** (the default): The property can only be accessed from within the component.
- **in** : The property is an input. It can be set and modified by the user of this component, for example through bindings or by assignment in callbacks. The component can provide a default binding, but it can't overwrite it by assignment
- **out** : An output property that can only be set by the component. It's read-only for the users of the components.
- **in-out** : The property can be read and modified by everyone.

```
export component Button {  
    // This is meant to be set by the user of the component.  
    in property <string> text;  
    // This property is meant to be read by the user of the component.  
    out property <bool> pressed;  
    // This property is meant to both be changed by the user and the component  
    in-out property <bool> checked;  
  
    // This property is internal to this component.  
    private property <bool> has-mouse;  
}
```

slint

All properties declared at the top level of a component that aren't `private` are accessible from the outside when using a component as an element, or via the language bindings from the business logic.

Change Callbacks

In Slint, it's possible to define a callback that is invoked when a property's value changes.

```
import { LineEdit } from "std-widgets.slint";  
export component Example inherits Window {  
    VerticalLayout {  
        LineEdit {  
            // This callback is invoked when the `text` property of the LineEdit  
            // changes. text => { t.text = self.text; }  
        }  
        t := Text {}  
    }  
}
```

slint

Note that these callbacks aren't invoked immediately. Instead, they're queued for invocation in the subsequent iteration of the event loop. A callback is invoked only if the property's value has indeed changed. If a property's value changes multiple times within the same

event loop cycle, the callback is invoked only once. Additionally, if a property's value changes and then reverts to its original state before the callback is executed, the callback won't be invoked.

Warning: Altering properties during a change event in a way that could lead to the same property being affected is undefined behavior.

```
export component Example {  
    in-out property <int> foo;  
    property <int> bar: foo + 1;  
    // This setup creates a potential loop between `foo` and `bar`, and the ou  
    changed bar => { foo += 1; }  
}
```

slint

The above represents an infinite loop. Slint will break the loop after a few iterations. Consequently, if there's a sequence of changed callbacks where one callback triggers another change callback, this sequence might break, and further callbacks won't be invoked.

Therefore, it's crucial not to overuse changed callbacks.

Warning: Utilize changed callbacks only when an alternative through binding isn't feasible.

For instance, avoid doing this:

```
changed bar => { foo = bar + 1; }
```

slint

Instead, opt for:

```
foo: bar + 1;
```

slint

Declarative bindings automatically manage dependencies. Using a changed callback forces immediate evaluation of bindings, which are typically evaluated lazily. This practice also compromises the purity of bindings, complicating edits via graphical editors. Accumulating excessive changed events can introduce issues and bugs, especially in scenarios involving loops, where a change callback modifies a property, potentially triggering changes to the same property.

Previous

The `.`slint`` File

Next

Expressions and Statements

© 2025 SixtyFPS GmbH

Expressions

Expressions are a powerful way to declare relationships and connections in your user interface. They're typically used to combine basic arithmetic with access to properties of other elements. When these properties change, the expression is automatically re-evaluated and a new value is assigned to the property the expression is associated with:

```
export component Example {  
    // declare a property of type int  
    in-out property<int> my-property;  
  
    // This accesses the property  
    width: root.my-property * 20px;  
}
```

slint

When `my-property` changes, the width changes automatically, too.

Arithmetic in expression with numbers works like in most programming languages with the operators `*`, `+`, `-`, `/`:

```
export component Example {  
    in-out property <int> p: 1 * 2 + 3 * 4; // same as (1 * 2) + (3 * 4)  
}
```

slint

Concatenate strings with `+`.

The operators `&&` and `||` express logical *and* and *or* between boolean values. The operators `==`, `!=`, `>`, `<`, `>=` and `<=` compare values of the same type.

Access an element's properties by using its name, followed by a `.` and the property name:

```
export component Example {  
  foo := Rectangle {  
    x: 42px;  
  }  
  x: foo.x;  
}
```

slint

The ternary operator `... ? ... : ...` is also supported, like in C or JavaScript:

```
export component Example inherits Window {  
  preferred-width: 100px;  
  preferred-height: 100px;  
  
  Rectangle {  
    touch := TouchArea {}  
    background: touch.pressed ? #111 : #eee;  
    border-width: 5px;  
    border-color: !touch.enabled ? #888  
      : touch.pressed ? #aaa  
      : #555;  
  }  
}
```

slint

Statements

Let statements (local variables)

The `let` keyword can be used to create local variables. Local variables are immutable and cannot be redeclared (even in other scopes). They optionally have a type annotation.

```
clicked => {  
  let foo = "hello world"; // no type annotation, inferred type  
  debug(foo); // prints "hello world"
```

slint

```
let bar: int = 2; // explicit type annotation
debug(bar); // prints "2"
}
```

Assignment

```
clicked => { some-property = 42; }
```

slint

Self-assignment with += -= *= /=

```
clicked => { some-property += 42; }
```

slint

Calling a callback

```
clicked => { root.someCallback(); }
```

slint

Conditional statements

```
clicked => {
  if (condition) {
    foo = 42;
  } else if (other-condition) {
    bar = 28;
  } else {
    foo = 4;
  }
}
```

slint

Empty expression

```
clicked => {}  
// or  
clicked => { ; }
```

slint

Previous

Properties

Next

Positioning & Layouts

© 2025 SixtyFPS GmbH

Positioning and Layouts

All visual elements are shown in a window. The `x` and `y` properties store the elements coordinates relative to their parent element. Slint determines the absolute position of an element by adding the parent's position to the element's position. If the parent has a parent element itself, then that one is added as well. This calculation continues until the top-level element is reached.

The `width` and `height` properties store the size of visual elements.

You can create an entire graphical user interface by placing the elements in two ways:

- Explicitly - by setting the `x`, `y`, `width`, and `height` properties.
- Automatically - by using layout elements.

Explicit placement is great for static scenes with few elements. Layouts are suitable for complex user interfaces and help create scalable user interfaces. Layout elements express geometric relationships between elements.

Explicit Placement

The following example places two rectangles into a window, a blue one and a green one. The green rectangle is a child of the blue:

```
// Explicit positioning                                         slint
export component Example inherits Window {
    width: 200px;
    height: 200px;
    Rectangle {
```

```

x: 100px;
y: 70px;
width: parent.width - self.x;
height: parent.height - self.y;
background: blue;

Rectangle {
    x: 10px;
    y: 5px;
    width: 50px;
    height: 30px;
    background: green;
}

}
}

```

The positions of both rectangles and the size of the inner green one are fixed. The outer blue rectangle has a size that's automatically calculated using binding expressions for the `width` and `height` properties. The calculation results in the bottom left corner aligning with the corner of the window - it updates whenever the `width` and `height` of the window changes.

When specifying explicit values for any of the geometric properties, Slint requires you to attach a unit to the number. You can choose between two different units:

- Logical pixels, using the `px` unit suffix. This is the recommended unit.
- Physical pixels, using the `phx` unit suffix

Logical pixels scale automatically with the device pixel ratio that your system is configured with. For example, on a modern High-DPI display the device pixel ratio can be 2, so every logical pixel occupies 2 physical pixels. On an older screen the user interface scales without any adaptations.

You can also specify the `width` and `height` properties as a `%` percentage unit, which applies relative to the parent element. For example a `width: 50%` means half of the parent's `width`.

The default values for `x` and `y` properties center elements within their parent.

The default values for `width` and `height` depend on the type of element. Elements such as `Image`, `Text`, as well as most widgets are sized automatically based on their content. The following elements don't have content and default to fill their parent element when they do not have children:

- `Rectangle`
- `TouchArea`
- `FocusScope`
- `Flickable`

Layouts are also defaulting to fill the parent, regardless of their own preferred size.

Other elements (including custom ones without base) default to using their preferred size.

Preferred Size

You can specify the preferred size of elements with the `preferred-width` and `preferred-height` properties.

When not explicitly set, the preferred size depends on the children, and is the preferred size of the child that has the bigger preferred size, whose `x` and `y` property are not set. The preferred size is therefore computed from the child to the parent, just like other constraints (maximum and minimum size), unless explicitly overwritten.

A special case is to set the preferred size to be the size of the parent using `100%` as value. For example, this component uses the size of the parent by default:

```
export component MyComponent {  
  preferred-width: 100%;  
  preferred-height: 100%;  
  // ...  
}
```

slint

Automatic Placement Using Layouts

Slint comes with different layout elements that automatically calculate the position and size of their children:

- `VerticalLayout` / `HorizontalLayout` : The children are placed along the vertical or horizontal axis.
- `GridLayout` : The children are placed in a grid of columns and rows.

You can also nest layouts to create complex user interfaces.

You can tune the automatic placement using different constraints, to accommodate the design of your user interface. Each element has a minimum, a maximum size, and a preferred size. Set these explicitly using the following properties:

- `min-width`
- `min-height`
- `max-width`
- `max-height`
- `preferred-width`
- `preferred-height`

Any element with a specified `width` and `height` has a fixed size in a layout.

When there is extra space in a layout, elements can stretch along the layout axis. You can control this stretch factor between the element and its siblings with these properties:

- `horizontal-stretch`
- `vertical-stretch`

A value of `0` means that the element won't stretch at all. All elements stretch equally if they all have a stretch factor of `0`.

The default value of these constraint properties may depends on the content of the element. If the element's `x` or `y` isn't set, these constraints are also automatically applied to the parent element.

Common Properties on Layout Elements

All layout elements have the following properties in common:

- `spacing` : This controls the spacing between the children.
- `padding` : This specifies the padding within the layout, the space between the elements and the border of the layout.

For more fine grained control, you can split the `padding` property into properties for each side of the layout:

- `padding-left`
- `padding-right`
- `padding-top`
- `padding-bottom`

VerticalLayout and HorizontalLayout

The `VerticalLayout` and `HorizontalLayout` elements place their children in a column or a row. By default, they stretch or shrink to take the whole space. You can adjust the element's alignment as needed.

The following example places the blue and yellow rectangle in a row and evenly stretched across the 200 logical pixels of `width` :

```
// Stretch by default
export component Example inherits Window {
    width: 200px;
    height: 200px;
    HorizontalLayout {
        Rectangle { background: blue; min-width: 20px; }
        Rectangle { background: yellow; min-width: 30px; }
    }
}
```

slint

The example below, on the other hand, specifies that the rectangles align to the start of the layout (the visual left). That results in no stretching but instead the rectangles retain their

specified minimum width:

```
// Unless an alignment is specified
export component Example inherits Window {
    width: 200px;
    height: 200px;
    HorizontalLayout {
        alignment: start;
        Rectangle { background: blue; min-width: 20px; }
        Rectangle { background: yellow; min-width: 30px; }
    }
}
```

slint

The example below nests two layouts for a more complex scene:

```
export component Example inherits Window {
    width: 200px;
    height: 200px;
    HorizontalLayout {
        // Side panel
        Rectangle { background: green; width: 10px; }

        VerticalLayout {
            padding: 0px;
            //toolbar
            Rectangle { background: blue; height: 7px; }

            Rectangle {
                border-color: red; border-width: 2px;
                HorizontalLayout {
                    Rectangle { border-color: blue; border-w
                    Rectangle { border-color: green; border-
                }
            }
            Rectangle {
                border-color: orange; border-width: 2px;
            }
        }
    }
}
```

slint

```

        HorizontalLayout {
            Rectangle { border-color: black; border-
            Rectangle { border-color: pink; border-w
        }
    }
}
}
}

```

Relative Lengths

Sometimes it's convenient to express the relationships of length properties in terms of relative percentages. For example the following inner blue rectangle has half the size of the outer green window:

```

export component Example inherits Window {
    preferred-width: 100px;
    preferred-height: 100px;

    background: green;
    Rectangle {
        background: blue;
        width: parent.width * 50%;
        height: parent.height * 50%;
    }
}

```

slint

This pattern of expressing the `width` or `height` in percent of the parent's property with the same name is common. For convenience, a short-hand syntax exists for this scenario:

- The property is `width` or `height`
- A binding expression evaluates to a percentage.

If these conditions are met, then it's not necessary to specify the `parent` property, instead you can simply use the percentage. The earlier example then looks like this:

```
export component Example inherits Window {  
    preferred-width: 100px;  
    preferred-height: 100px;  
  
    background: green;  
    Rectangle {  
        background: blue;  
        width: 50%;  
        height: 50%;  
    }  
}
```

slint

Alignment

Each element is sized according to their `width` or `height` if specified, otherwise it's set to the minimum size which is set with the `min-width` or `min-height` property, or the minimum size of an inner layout, whatever is bigger.

The elements are placed according to the alignment. The size of elements is bigger than the minimum size only if the `alignment` property of the layout is `LayoutAlignment.stretch` (the default)

This example show the different alignment possibilities:

```
export component Example inherits Window {  
    width: 300px;  
    height: 200px;  
    VerticalLayout {  
        HorizontalLayout {  
            alignment: stretch;  
            Text { text: "stretch (default)"; }  
            Rectangle { background: blue; min-width: 20px; }  
            Rectangle { background: yellow; min-width: 30px; }  
        }  
        HorizontalLayout {  
            alignment: center;  
            Text { text: "center"; }  
            Rectangle { background: blue; min-width: 20px; }  
            Rectangle { background: yellow; min-width: 30px; }  
        }  
    }  
}
```

slint

```
    alignment: start;
    Text { text: "start"; }
    Rectangle { background: blue; min-width: 20px; }
    Rectangle { background: yellow; min-width: 30px;
}
}

HorizontalLayout {
    alignment: end;
    Text { text: "end"; }
    Rectangle { background: blue; min-width: 20px; }
    Rectangle { background: yellow; min-width: 30px;
}
}

HorizontalLayout {
    alignment: start;
    Text { text: "start"; }
    Rectangle { background: blue; min-width: 20px; }
    Rectangle { background: yellow; min-width: 30px;
}
}

HorizontalLayout {
    alignment: center;
    Text { text: "center"; }
    Rectangle { background: blue; min-width: 20px; }
    Rectangle { background: yellow; min-width: 30px;
}
}

HorizontalLayout {
    alignment: space-between;
    Text { text: "space-between"; }
    Rectangle { background: blue; min-width: 20px; }
    Rectangle { background: yellow; min-width: 30px;
}
}

HorizontalLayout {
    alignment: space-around;
    Text { text: "space-around"; }
    Rectangle { background: blue; min-width: 20px; }
    Rectangle { background: yellow; min-width: 30px;
}
}

}
```

```
}
```

Stretch algorithm

When the `alignment` is set to stretch (the default), the elements are sized to their minimum size, then the extra space is shared amongst element proportional to their stretch factor set with the `horizontal-stretch` and `vertical-stretch` properties. The stretched size won't exceed the maximum size. The stretch factor is a floating point number. The elements that have a default content size usually defaults to 0 while elements that default to the size of their parents defaults to 1. An element of a stretch factor of 0 keep its minimum size, unless all the other elements also have a stretch factor of 0 or reached their maximum size.

Examples:

```
export component Example inherits Window {  
    width: 300px;  
    height: 200px;  
    VerticalLayout {  
        // Same stretch factor (1 by default): the size is distributed evenly  
        HorizontalLayout {  
            Rectangle { background: blue; }  
            Rectangle { background: yellow; }  
            Rectangle { background: green; }  
        }  
        // Elements with a bigger min-width are given a bigger share of the extra space  
        HorizontalLayout {  
            Rectangle { background: cyan; min-width: 100px; }  
            Rectangle { background: magenta; min-width: 50px; }  
            Rectangle { background: gold; }  
        }  
        // Stretch factor twice as big: grows twice as much  
        HorizontalLayout {  
            Rectangle { background: navy; horizontal-stretch: 2; }  
            Rectangle { background: gray; }  
        }  
    }  
}
```

```
// All elements not having a maximum width have a static width
HorizontalLayout {
    Rectangle { background: red; max-width: 20px; }
    Rectangle { background: orange; horizontal-stretch: true; }
    Rectangle { background: pink; horizontal-stretch: true; }
}
}
```

for

The `VerticalLayout` and `HorizontalLayout` can also contain `for` or `if` expressions:

```
export component Example inherits Window { slint
    width: 200px;
    height: 50px;
    HorizontalLayout {
        Rectangle { background: green; }
        for t in [ "Hello", "World", "!" ] : Text {
            text: t;
        }
        Rectangle { background: blue; }
    }
}
```

GridLayout

The `GridLayout` lays the element in a grid. Each element gains the properties `row`, `col`, `rowspan`, and `colspan`. You can either use a `Row` sub-element, or set the `row` property explicitly. These properties must be statically known at compile time, so it's impossible to use arithmetic or depend on properties. As of now, the use of `for` or `if` isn't allowed in a grid layout.

This example use the `Row` element

```
export component Foo inherits Window {  
    width: 200px;  
    height: 200px;  
    GridLayout {  
        spacing: 5px;  
        Row {  
            Rectangle { background: red; }  
            Rectangle { background: blue; }  
        }  
        Row {  
            Rectangle { background: yellow; }  
            Rectangle { background: green; }  
        }  
    }  
}
```

slint

This example use the `col` and `row` property:

```
export component Foo inherits Window {  
    width: 200px;  
    height: 150px;  
    GridLayout {  
        spacing: 0px;  
        Rectangle { background: red; }  
        Rectangle { background: blue; }  
        Rectangle { background: yellow; row: 1; }  
        Rectangle { background: green; }  
        Rectangle { background: black; col: 2; row: 0; }  
    }  
}
```

slint

Container Components (@children)

When creating components, it's sometimes useful to influence where child elements are placed when used. For example, a component that draws a label above an element inside:

```
export component MyApp inherits Window {  
  
    BoxWithLabel {  
        Text {  
            // ...  
        }  
    }  
  
    // ...  
}
```

slint

You can implement such a `BoxWithLabel` using a layout. By default child elements like the `Text` element become direct children of the `BoxWithLabel`, but for this example they need to become children of the layout instead. To do this can change the default child placement by using the `@children` expression inside the element hierarchy of a component:

```
component BoxWithLabel inherits GridLayout {  
  
    Row {  
        Text { text: "label text here"; }  
    }  
    Row {  
        @children  
    }  
}  
  
export component MyApp inherits Window {  
    preferred-height: 100px;  
    BoxWithLabel {  
        Rectangle { background: blue; }  
        Rectangle { background: yellow; }  
    }  
}
```

slint

Previous

Expressions and Statements

Next

Globals

© 2025 SixtyFPS GmbH

Globals

Declare a global singleton with `global Name { /* .. properties or callbacks .. */ }` to make properties and callbacks available throughout the entire project. Access them using `Name.property`.

For example, this can be useful for a common color palette:

```
global Palette {  
    in-out property<color> primary: blue;  
    in-out property<color> secondary: green;  
}  
  
export component Example inherits Rectangle {  
    background: Palette.primary;  
    border-color: Palette.secondary;  
    border-width: 2px;  
}
```

slint

Export a global to make it accessible from other files (see [Modules](#)). To make your global visible to native code with the business logic, re-export a global from the file also exporting the main application component.

```
export global Logic {  
    in-out property <int> the-value;  
    pure callback magic-operation(int) -> int;  
}  
// ...
```

slint

```
slint::slint!{ rust
    export global Logic {
        in-out property <int> the-value;
        pure callback magic-operation(int) -> int;
    }

    export component App inherits Window {
        // ...
    }
}

fn main() {
    let app = App::new();
    app.global::<Logic>().on_magic_operation(|value| {
        eprintln!("magic operation input: {}", value);
        value * 2
    });
    app.global::<Logic>().set_the_value(42);
    // ...
}
```

It's possible to re-expose a callback or properties from a global using the two way binding syntax.

```
global Logic { slint
    in-out property <int> the-value;
    pure callback magic-operation(int) -> int;
}

component SomeComponent inherits Text {
    // use the global in any component
    text: "The magic value is:" + Logic.magic-operation(42);
}
```

```
export component MainWindow inherits Window {  
    // re-expose the global properties such that the native code  
    // can access or modify them  
    in-out property the-value <=> Logic.the-value;  
    pure callback magic-operation <=> Logic.magic-operation;  
  
    SomeComponent {}  
}
```

Previous

[Positioning & Layouts](#)

Next

[Repetition and Data Models](#)

© 2025 SixtyFPS GmbH

Repetition

Use the `for - in` syntax to create an element multiple times.

The syntax looks like this: `for name[index] in model : id := Element { ... }`

The *model* can be of the following type:

- an integer, in which case the element will be repeated that amount of time
- an [array type or a model](#) declared natively, in which case the element will be instantiated for each element in the array or model.

The *name* will be available for lookup within the element and is going to be like a pseudo-property set to the value of the model. The *index* is optional and will be set to the index of this element in the model. The *id* is also optional.

Examples

```
export component Example inherits Window {  
  preferred-width: 300px;  
  preferred-height: 100px;  
  for my-color[index] in [ #e11, #1a2, #23d ]: Rectangle {  
    height: 100px;  
    width: 60px;  
    x: self.width * index;  
    background: my-color;  
  }  
}
```

slint

```

export component Example inherits Window {
    preferred-width: 50px;
    preferred-height: 50px;
    in property <[{foo: string, col: color}]> model: [
        {foo: "abc", col: #f00 },
        {foo: "def", col: #00f },
    ];
    VerticalLayout {
        for data in root.model: my-repeated-text := Text {
            color: data.col;
            text: data.foo;
        }
    }
}

```

slint

Arrays and Models

Arrays are declared by wrapping [and] square brackets around the type of the array elements.

Array literals as well as properties holding arrays act as models in `for` expressions.

```

export component Example {
    in-out property<[int]> list-of-int: [1,2,3];
    in-out property<[{a: int, b: string}]> list-of-structs: [{ a: 1, b: "hello" }];
}

```

slint

Arrays define the following operations:

- **array.length** : One can query the length of an array and model using the builtin `.length` property.
- **array[index]** : The index operator retrieves individual elements of an array.

Out of bound access into an array will return default-constructed values.

```
export component Example {  
    in-out property<[int]> list-of-int: [1,2,3];  
  
    out property <int> list-len: list-of-int.length;  
    out property <int> first-int: list-of-int[0];  
}
```

slint

Previous

Globals

Next

Animations

© 2025 SixtyFPS GmbH

Animations

Declare animations for properties with the `animate` keyword like this:

```
export component Example inherits Window {  
  preferred-width: 100px;  
  preferred-height: 100px;  
  
  background: area.pressed ? blue : red;  
  animate background {  
    duration: 250ms;  
  }  
  
  area := TouchArea {}  
}
```

slint

This will animate the color property for 250ms whenever it changes.

It's also possible to animate several properties with the same animation, so:

```
animate x, y { duration: 100ms; easing: ease-out-bounce; }
```

slint

is the same as:

```
animate x { duration: 100ms; easing: ease-out-bounce; }  
animate y { duration: 100ms; easing: ease-out-bounce; }
```

slint

Fine-tune animations using the following parameters:

delay

`duration` default: 0ms

The amount of time to wait before starting the animation.

duration

`duration` default: 0ms

The amount of time it takes for the animation to complete.

iteration-count

`float` default: 0.0

The number of times an animation should run. A negative value specifies infinite reruns.

Fractional values are possible. For permanently running animations, see `'animation-tick()'`.

easing

`easing` default: linear

Can be any of the following. See [easings.net](#) ↗ for a visual reference:

direction

`enum AnimationDirection` default: the first enum value

Use this to set or change the direction of the animation.

Previous

Repetition and Data Models

Next

States and Transitions

States

The `states` statement allows to declare states and set properties of multiple elements in one go:

```
export component Example inherits Window {  
  preferred-width: 100px;  
  preferred-height: 100px;  
  default-font-size: 24px;  
  
  label := Text {}  
  ta := TouchArea {  
    clicked => {  
      active = !active;  
    }  
  }  
  property <bool> active: true;  
  states [  
    active when active && !ta.has-hover: {  
      label.text: "Active";  
      root.background: blue;  
    }  
    active-hover when active && ta.has-hover: {  
      label.text: "Active\nHover";  
      root.background: green;  
    }  
    inactive when !active: {  
      label.text: "Inactive";  
    }  
  ]  
}
```

```

        root.background: gray;
    }
]
}

```

In this example, the `active` and `active-hovered` states are defined depending on the value of the `active` boolean property and the `TouchArea`'s `has-hover`. When the user hovers the example with the mouse, it will toggle between a blue and a green background, and adjust the text label accordingly. Clicking toggles the `active` property and thus enters the `inactive` state.

Transitions

Transitions bind animations to state changes.

This example defines two transitions. First the `out` keyword is used to animate all properties for 800ms when leaving the `disabled` state. The second transition uses the `in` keyword to animate the background when transitioning into the `down` state.

```

export component Example inherits Window {
    preferred-width: 100px;
    preferred-height: 100px;

    text := Text { text: "hello"; }

    in-out property<bool> pressed;
    in-out property<bool> is-enabled;

    states [
        disabled when !root.is-enabled : {
            background: gray; // same as root.background: gray;
            text.color: white;
            out {
                animate * { duration: 800ms; }
            }
        }
    ]
}

```

slint

```
down when pressed : {
    background: blue;
    in {
        animate background { duration: 300ms; }
    }
}
]
```

Transition Types

There are three types of transitions you can define:

- **in** : Animates properties when entering a state
- **out** : Animates properties when leaving a state
- **in-out** : Animates properties both when entering and leaving a state

The `in-out` transition is useful when you want the same animation to play for both entering and exiting a state, avoiding the need to duplicate the animation definition.

Previous

Animations

Next

Functions and Callbacks

Functions

Similar to other programming languages, functions in Slint are way to name, organize and reuse a piece of logic/code.

Functions can be defined as part of a component, or as part of an element within a component. It is not possible to declare global (top-level) functions, or to declare them as part of a struct or enum. It is also not possible to nest functions within other functions.

Declaring Functions

Functions in Slint are declared using the `function` keyword. For example:

```
export component Example {  
    // ...  
    function my-function(parameter: int) -> string {  
        // Function code goes here  
        return "result";  
    }  
}
```

slint

Functions can have parameters which are declared within parentheses, following the format `name: type`. These parameters can be referenced by their names within the function body. Parameters are passed by value.

Functions can also return a value. The return type is specified after `->` in the function signature. The `return` keyword is used within the function body to return an expression of the declared type. If a function does not explicitly return a value, the value of the last statement is returned by default.

Functions can be annotated with the `pure` keyword. This indicates that the function does not cause any side effects. More details can be found in the [Purity](#) chapter.

Calling Functions

A function can be called without an element name (like a function call in other languages) or with an element name (like a method call in other languages):

```
import { Button, VerticalBox } from "std-widgets.slint";          slint

export component Example {
    // Call without an element name:
    property <string> my-property: my-function();
    // Call with an element name:
    property <int> my-other-property: my_button.my-other-function();

    pure function my-function() -> string {
        return "result";
    }

    VerticalBox {
        Text {
            // Called with a pre-defined element:
            text: root.my-function();
        }
    }

    my_button := Button {
        text: "Click me";
        clicked => { self.text = root.my-other-property; }
        pure function my-other-function() -> int {
            return 42;
        }
    }
}
```

```
}
```

Function Visibility

By default, functions are private and cannot be accessed from other components.

However, their accessibility can be modified using the `public` or `protected` keywords.

- A root-level function annotated with `public` can be accessed by any component.

To access such a function from a different component, you always need a target, which in practice means the calling component must declare the called component as one of its child elements.

```
export component HasFunction {  
    public pure function double(x: int) -> int {  
        return x * 2;  
    }  
}  
  
export component CallsFunction {  
    property <int> test: my-friend.double(1);  
  
    my-friend := HasFunction {  
    }  
}
```

slint

If a function is declared in a child element, even if marked public, it is not possible to call it from another component, as the child elements themselves are not public and a valid target for the function does not exist:

```
export component HasFunction {  
    t := Text {  
        public pure function double(x: int) -> int {  
            return x * 2;  
        }  
    }  
}
```

slint

```

        }
    }
}

export component CallsFunction {
    // Compiler error!
    // property <int> test: my-friend.t.double(1);

    my-friend := HasFunction {
    }
}

```

Functions marked `public` in an exported component can also be invoked from backend code (Rust, C++, JS). See the language-specific documentation for the generated code to use.

- A function annotated with `protected` can only be accessed by components that directly inherit from it.

Functions vs. Callbacks

There are a lot of similarities between functions and [callbacks](#):

- They are both callable blocks of logic/code
- They are invoked in the same way
- They can both have parameters and return values
- They can both be declared `pure`

But there are also differences:

- The code/logic in the callback can be set in the backend code and implemented in the backend language (Rust, C++, JS), while functions must be defined entirely in slint
- The syntax for defining a callback is different
- Callbacks can be declared without assigning a block of code to them

- Callbacks have a special syntax for declaring aliases using the two-way binding operator `<=>`
- Callback visibility is always similar to `public` functions

In general, the biggest reason to use callbacks is to be able to handle them from the backend code. Use a function if that is not needed.

Callbacks

Components may declare callbacks, that communicate changes of state to the outside. Callbacks are invoked by “calling” them like you would call a function.

You react to callback invocation by declaring a handler using the `=>` arrow syntax. The built-in `TouchArea` element declares a `clicked` callback, that’s invoked when the user touches the rectangular area covered by the element, or clicks into it with the mouse. In the example below, the invocation of that callback is forwarded to another custom callback (`hello`) by declaring a handler and invoking our custom callback:

```
export component Example inherits Rectangle {slint
    // declare a callback
    callback hello;

    area := TouchArea {
        // sets a handler with `=>`
        clicked => {
            // emit the callback
            root.hello()
        }
    }
}
```

It’s possible to add parameters to a callback:

```
export component Example inherits Rectangle {slint
    // declares a callback
```

```
callback hello(int, string);
hello(aa, bb) => { /* ... */ }
}
```

Callbacks may also return a value:

```
export component Example inherits Rectangle {
    // declares a callback with a return value
    callback hello(int, int) -> int;
    hello(aa, bb) => { aa + bb }
}
```

slint

Callback arguments can also have names. The names of arguments have currently no semantic value, but they improve readability of your code.

```
export component Example inherits Rectangle {
    // Declare a callback with named argument
    callback hello(foo: int, bar: string);
    // The names can be overridden with
    // anything when setting a handler
    hello(aa, bb) => { /* ... */ }
}
```

slint

Aliases

It's possible to declare callback aliases in a similar way to two-way bindings:

```
export component Example inherits Rectangle {
    callback clicked <=> area.clicked;
    area := TouchArea {}
}
```

slint

Previous

States and Transitions

Next

Name Resolution (Scope)

© 2025 SixtyFPS GmbH

Name Resolution (Scope)

Function calls have the same name resolution rules as properties and callbacks. When called without an element name:

- If the element that the function is called in (`self`) defines a function with that name, it is chosen.
- If not, name resolution continues to its parent element, and so on, until the root component.

When called with an element name (or `self`, `parent` or `root`), the function must be defined on that element. Name resolution does not look at ancestor elements in this case. Note that this means calling a function without an element name is *not* equivalent to calling it with `self` (which is how methods work in many languages).

Multiple functions with the same name are allowed in the same component, as long as they are defined on different elements. Therefore it is possible for a function to shadow another function from an ancestor element.

```
export component Example {  
    property <int> secret_number: my-function();  
    public pure function my-function() -> int {  
        return 1;  
    }  
  
    VerticalLayout {  
        public pure function my-function() -> int {  
            return 2;  
        }  
    }  
}
```

slint

```

Text {
    text: "The secret number is " + my-function();
}

public pure function my-function() -> int {
    return 3;
}

Text {
    text: "The other secret number is " + my-function();
}

}

```

In the example above, the property `secret_number` will be set to 1, and the text labels will say “The secret number is 3” and “The other secret number is 2”.

Previous

Functions and Callbacks

Next

Structs and Enums

Structs and Enums

Structs

Define named structures using the `struct` keyword:

```
export struct Player {  
    name: string,  
    score: int,  
}  
  
export component Example {  
    in-out property<Player> player: { name: "Foo", score: 100 };  
}
```

slint

The default value of a struct, is initialized with all its fields set to their default value.

Anonymous Structures

Declare anonymous structures using `{ identifier1: type1, identifier2: type2 }` syntax, and initialize them using `{ identifier1: expression1, identifier2: expression2 }`.

You may have a trailing `,` after the last expression or type.

```
export component Example {  
    in-out property<{name: string, score: int}> player: { name: "Foo", score:  
        in-out property<{a: int, }> foo: { a: 3 };  
}
```

slint

Enums

Define an enumeration with the `enum` keyword:

```
export enum CardSuit { clubs, diamonds, hearts, spade }  
slint  
  
export component Example {  
    in-out property<CardSuit> card: spade;  
    out property<bool> is-clubs: card == CardSuit.clubs;  
}
```

Enum values can be referenced by using the name of the enum and the name of the value separated by a dot. (eg: `CardSuit.spade`)

The name of the enum can be omitted in bindings of the type of that enum, or if the return value of a callback is of that enum.

The default value of each enum type is always the first value.

Previous

Name Resolution (Scope)

Next

Debugging Techniques

Debugging Techniques

On this page we share different techniques and tools we've built into Slint that help you track down different issues you may be running into, during the design and development.

Debugging Property Values

Use the [debug\(\)](#) function to print the values of properties to stderr.

Slow Motion Animations

Animations in the user interface need to be carefully designed to have the correct duration and changes in element positioning or size need to follow an easing curve.

To inspect the animations in your application, set the `SLINT_SLOW_ANIMATIONS` environment variable before running the program. This variable accepts an unsigned integer value that is the factor by which to globally slow down the steps of all animations, automatically. This means that you don't have to make any manual changes to the `.slint` markup and recompile. For example, `SLINT_SLOW_ANIMATIONS=4` slows down animations by a factor of four.

User Interface Scaling

The use of logical pixel lengths throughout `.slint` files lets Slint compute the number of physical pixels, dynamically, depending on the device-pixel ratio of the screen. To get an impression of how the individual elements look like when rendered on a screen with a different device-pixel ratio, set the `SLINT_SCALE_FACTOR` environment variable before running

the program. This variable accepts a floating pointer number that is used to convert logical pixel lengths to physical pixel lengths. For example, `SLINT_SCALE_FACTOR=2` renders the user interface in a way where every logical pixel has twice the width and height.

Note: Currently, only the FemtoVG and Skia renderers support this environment variable.

Debugging for Performance Improvements

Slint attempts to use hardware-acceleration to ensure that rendering the user interface consumes a minimal amount of CPU resources while maintaining smooth animations. However, depending on the complexity of the user interface, quality of the graphics drivers, or the power of the GPU in your system, you may hit limits and experience slowness. To address this issue, set the `SLINT_DEBUG_PERFORMANCE` environment variable before running the program, to inspect the frame rate. The following options affect the frame rate inspection and reporting:

- `refresh_lazy` : The frame rate is measured only when an actual frame is rendered, for example due to a running animation, user interaction, or some other state change that results in a visual difference in the user interface. If there is no change, a low frame rate is reported. Use this option to verify that no unnecessary repainting happens when there are no visual changes. For example, in a user interface that shows a text input field with a cursor that blinks once per second, the reported frame rate should be two.
- `refresh_full_speed` : The user interface is continuously refreshed, even if nothing is changed. This continuous refresh results in a higher load on the system. Use this option to identify any bottlenecks that prevent you from achieving smooth animations. Also disables partial rendering with the software renderer.
- `console` : The frame rate is printed to `stderr` on the console.
- `overlay` : The frame rate is as an overlay text label on top of the user interface in each window.

Use these options in combination, separated by a comma. You must select a combination of one frame rate measurement method and a reporting method. For example,

`SLINT_DEBUG_PERFORMANCE=refresh_full_speed,overlay` repeatedly re-renders the entire user interface in each window and prints the achieved frame rate in the top-left corner. In comparison, `SLINT_DEBUG_PERFORMANCE=refresh_lazy,console,overlay` measures the frame

rate only when something in the user interface changes and the measured value is printed to `stderr` as well as rendered as an overlay text label.

The environment variable must be set before running the program. If the application runs on a microcontroller without the standard library, the environment variable must be set during compilation.

Tuning Rendering Performance

If you're not satisfied with the performance, it might be worthwhile to descend into a low-level investigation. Tools such as [RenderDoc](#) permit recording the rendering output of your application and can give you detailed insight into the OpenGL/Vulkan commands Slint's renderers produce for your user interface.

As a general rule of thumb, it's best to minimize the number of commands per frame. With OpenGL you'll see `glDrawArrays()` and `glDrawElementsInstanced*` calls filling the color buffers. Reducing the number of calls tends to improve performance.

For example, if you draw a series of bar charts by filling rectangles with a gradient, you'll observe that each rectangle is a draw call on its own:

```
component BarChart inherits Rectangle {  
    background: black;  
    height: 150px;  
    HorizontalLayout {  
        spacing: 10px;  
        for i in 5: Rectangle {  
            height: 10px + i * 20px;  
            width: 20px;  
            background: @linear-gradient(180deg, #f00 0%, #0f0);  
        }  
    }  
}
```

slint

For example when using the Skia renderer, if you replace the gradient with a plain color, the calls will all be batched together into one call. But when the UI design requires the use of a

gradient, that's not possible. But there might be another way. If the underlying data for your bar chart rarely changes, it might be worthwhile to render the entire chart once into a texture and therefore replace multiple calls with just one. This can be done by setting the [cache-rendering-hint](#) to `true` on the `BarChar` itself, which surrounds and captures the individual bar charts.

Note that extensive use of this technique comes at the expense of increased GPU memory usage and memory throughput.

Previous

Structs and Enums

Next

Focus Handling

© 2025 SixtyFPS GmbH

Focus Handling

Certain elements such as `TextInput` accept input from the mouse/finger and also key events originating from (virtual) keyboards. In order for an item to receive these events, it must have focus. This is visible through the `has-focus` (out) property.

You can manually activate the focus on an element by calling `focus()`:

```
import { Button } from "std-widgets.slint";slint

export component App inherits Window {
    VerticalLayout {
        alignment: start;
        Button {
            text: "press me";
            clicked => { input.focus(); }
        }
        input := TextInput {
            text: "I am a text input field";
        }
    }
}
```

Similarly, you can manually clear the focus on an element that's currently focused, by calling `clear-focus()`:

```
import { Button } from "std-widgets.slint";slint

export component App inherits Window {
```

```

VerticalLayout {
    alignment: start;
    Button {
        text: "press me";
        clicked => { input.clear-focus(); }
    }
    input := TextInput {
        text: "I am a text input field";
    }
}

```

After clearing the focus, keyboard input to the window is discarded until another element is explicitly focused. For example by calling `focus()`, an element acquiring focus when the user clicks on it, or when pressing tab and the first focusable element is found.

If you have wrapped the `TextInput` in a component, then you can forward such a focus activation using the `forward-focus` property to refer to the element that should receive it:

```

import { Button } from "std-widgets.slint";          slint

component LabeledInput inherits GridLayout {
    forward-focus: input;
    Row {
        Text {
            text: "Input Label:";
        }
        input := TextInput {}
    }
}

export component App inherits Window {
    GridLayout {
        Button {
            text: "press me";
            clicked => { label.focus(); }
        }
    }
}
```

```
        }
    label := LabeledInput {
        }
    }
}
```

If you use the `forward-focus` property on a `Window` or a `PopupWindow`, then the specified element will receive the focus the first time the window receives the focus - it becomes the initial focus element.

Previous

Debugging Techniques

Next

Translations

© 2025 SixtyFPS GmbH

Translations

Slint's translation infrastructure makes your application available in different languages.

Prerequisite

Install the [slint-tr-extractor ↗](#) tool to extract translatable strings from `.slint` files:

```
cargo install slint-tr-extractor
```

bash

You can choose between runtime translations using `gettext` or bundling translations directly into your executable for platforms where `gettext` is unavailable or impractical.

To translate your application, follow these steps:

1. Identify all user-visible strings that need translation and annotate them with the `@tr()` macro.
2. Extract annotated strings by running the `slint-tr-extractor` tool to generate `.pot` files.
3. Use a third-party tool to translate the strings into `.po` files for each target language.
4. **(For runtime `gettext` translations only)** Convert `.po` files into `.mo` files using [gettext's `msgfmt` ↗](#).
5. **(For bundled translations only)** Configure bundling during the build process to embed translations into your application.
6. Use Slint's API to select the appropriate translation based on the user's locale.

At this point, all strings marked for translation will be automatically rendered in the selected language.

Annotating Translatable Strings

Use the `@tr` macro in `.slint` files to mark strings for translation. This macro supports formatting and pluralization, and can include contextual information.

The first argument must be a plain string literal, followed by the arguments:

Basic Example

```
export component Example {  
  property <string> name;  
  Text {  
    text: @tr("Hello, {}", name);  
  }  
}
```

slint

Formatting

The `@tr` macro replaces each `{}` placeholder in the string marked for translation with the corresponding argument. It's also possible to re-order the arguments using `{0}`, `{1}`, and so on. Translators can use ordered placeholders even if the original string did not.

You can include the literal characters `{` and `}` in a string by preceding them with the same character. For example, escaping the `{` character with `\{` and the `}` character with `\}`.

Plurals

Use plural formatting when the translation of text involving a variable number of elements should change depending on whether there is a single element or multiple.

Given `count` and an expression that represents the count of something, form the plural with the `|` and `%` symbols like so:

```
@tr("I have {n} item" | "I have {n} items" % count).
```

Use `{n}` in the format string to access the expression after the `%`.

```
export component Example inherits Text {  
    in property <int> score;  
    in property <int> name;  
    text: @tr("Hello {0}, you have one point" | "Hello {0}, you have {n} point  
}  
slint
```

Context

Disambiguate translations for strings with the same source text but different contextual meanings by adding a context to the `@tr(...)` macro using the `"..." => syntax`.

Use the context to provide additional context information to translators, ensuring accurate and contextually appropriate translations.

The context must be a plain string literal and it appears as `msgctx` in the `.pot` files. If not specified, the context defaults to the name of the surrounding component.

```
export component MenuItem {  
    property <string> name : @tr("Default Name"); // Default: `MenuItem` will  
    property <string> tooltip : @tr("ToolTip" => "ToolTip for {}", name); // S  
}  
slint
```

Extracting Translatable Strings

Use [slint-tr-extractor](#) to generate a `.pot` file with all strings marked for translation:

```
find -name \*.slint | xargs slint-tr-extractor -o MY_PROJECT.pot
```

bash

This creates a file called `MY_PROJECT.pot`. Replace “MY_PROJECT” with your actual project name. To learn how the project name affects the lookup of translations, read the sections below.

Tip

- .pot files are [Gettext](#) template files.

Translating Strings

Start a new translation by creating a .po file from a .pot file. Both file formats are identical. You can either copy the file manually or use a tool like Gettext's msginit to start a new .po file.

The .po file contains the strings in a target language.

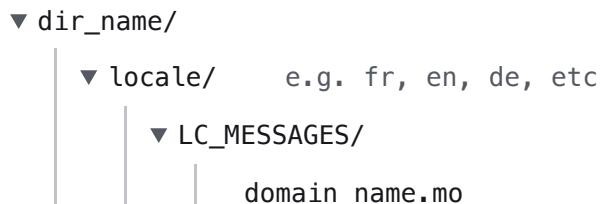
.po and .pot files are plain text files that you can edit with a text editor. We recommend using a dedicated translation tool for working with them, such as the following:

- [poedit](#)
- [OmegaT](#)
- [Lokalize](#)
- [Transifex](#) (web interface)

Runtime Translations with Gettext

Slint can use the [Gettext](#) library to load translations at run-time.

Gettext expects translation files - called message catalogs - in following directory hierarchy:



- dir_name : the base directory that you can choose freely.
- locale : The name of the user's locale for a given target language, such as fr for French, or de for German.

The locale is typically determined using environment variables that your operating system sets.

- `domain_name` : Selected based on the programming language you're using Slint with.

Tip

Read the [Gettext documentation](#) ↗ for more information.

Convert .po Files to .mo Files

Convert the human readable `.po` files into machine-friendly `.mo` files, which are a binary representation that is more efficient to read by code.

Use [Gettext](#) ↗'s `msgfmt` command line tool to convert `.po` files to `.mo` files:

```
msgfmt translation.po -o translation.mo
```

bash

Select and Load Translations

C++

Rust

First, enable the `gettext` feature of the `slint` crate in the `features` section to gain access to the translations API and activate run-time translation support.

Next, use the [`slint::init_translations!`](#) macro to specify the base location of your `.mo` files. This is the `dir_name` in the scheme of the previous section. Slint expects the `.mo` files to be in the corresponding sub-directories and their file name - `domain_name` - must match the package name in your `Cargo.toml` . This is often the same as the crate name.

For example:

```
slint::init_translations!(concat!(env!("CARGO_MANIFEST_DIR"), "/lang/"));rust
```

For example, if your `Cargo.toml` contains the following lines and the user's locale is `fr` :

[package]

toml

```
name = "gallery"
```

With these settings, Slint looks for `gallery.mo` in the `lang/fr/LC_MESSAGES/gallery.mo`.

Bundled Translations

Bundled translations embed the translated strings directly into your application binary. This approach is ideal for platforms like WASM or microcontrollers where `gettext` is unavailable.

Configure the Slint compiler to bundle translations by providing a path to the translations. Translation files should be organized in the following hierarchy:

```
path/<lang>/LC_MESSAGES/<domain>.po
```

plaintext

Bundling

C++ **Rust**

Use `slint_build::CompilerConfiguration`'s [`with_bundled_translations\(\)`](#) function to set up bundling in `build.rs`:

```
let config = slint_build::CompilerConfiguration::new()  
    .with_bundled_translations("path/to/translations");  
slint_build::compile_with_config("path/to/main-ui.slint", config).unwrap();
```

rust

The `<domain>` is the crate name.

Selecting a Translation

If you enable the `std` feature with Slint, language for translations is detected based on the locale: if one of the bundled language matches the selected locale, it will be used.

C++ **Rust**

Use the [`slint::select_bundled_translation`](#) function to change translations at runtime.

Previewing Translations with `slint-viewer`

Make sure the `gettext` feature was enabled when building `slint-viewer`. Use the `--translation-domain` and `--translation-dir` command line options to load translations for preview.

Previous

[Focus Handling](#)

Next

[Font Handling](#)

© 2025 SixtyFPS GmbH

Font Handling

Elements such as `Text` and `TextInput` can render text and allow customizing the appearance of the text through different properties. The properties prefixed with `font-`, such as `font-family`, `font-size` and `font-weight` affect the choice of font used for rendering to the screen. If any of these properties isn't specified, the `default-font-` values in the surrounding `Window` element apply, such as `default-font-family`.

The fonts chosen for rendering are automatically picked up from the system running the application. It's also possible to include custom fonts in your design. A custom font must be a TrueType font (`.ttf`), a TrueType font collection (`.ttc`) or an OpenType font (`.otf`). You can select a custom font with the `import` statement: `import "./my_custom_font.ttf"` in a `.slint` file. This instructs the Slint compiler to include the font and makes the font families globally available for use with `font-family` properties.

For example:

```
import "./NotoSans-Regular.ttf";  
  
export component Example inherits Window {  
    default-font-family: "Noto Sans";  
  
    Text {  
        text: "Hello World";  
    }  
}
```

slint

Previous

Next

Translations

Custom Controls

© 2025 SixtyFPS GmbH

Custom Controls

Custom Control Introduction

A Clickable Button

```
import { VerticalBox, Button } from "std-widgets.slint";
export component Recipe inherits Window {
    in-out property <int> counter: 0;
    VerticalBox {
        button := Button {
            text: "Button, pressed " + root.counter + " times";
            clicked => {
                root.counter += 1;
            }
        }
    }
}
```

slint

In this first example, you see the basics of the Slint language:

- We import the `VerticalBox` layout and the `Button` widget from the standard library using the `import` statement. This statement can import widgets or your own components declared in different files. You don't need to import built-in element such as `Window` or `Rectangle`.
- We declare the `Recipe` component using the `component` keyword. `Recipe` inherits from `Window` and has elements: A layout (`VerticalBox`) with one button.

- You instantiate elements using their name followed by a pair of braces (with optional contents. You can assign a name to a specific element using `:=`
- Elements have properties. Use `:` to set property values. Here we assign a binding that computes a string by concatenating some string literals, and the `counter` property to the `Button`'s `text` property.
- You can declare custom properties for any element with `property <...>`. A property needs to have a type, and can have a default value and an access specifier. Access specifiers like `private`, `in`, `out` or `in-out` defines how outside elements can interact with the property. `Private` is the default value and stops any outside element from accessing the property. The `counter` property is custom in this example.
- Elements can also have callback. In this case we assign a callback handler to the `clicked` callback of the button with `=> { ... }`.
- Property bindings are automatically re-evaluated if any of the properties the binding depends on changes. The `text` binding of the button is automatically re-computed whenever the `counter` changes.

React to a Button Click in Native Code

This example increments the `counter` using native code:

```
import { VerticalBox, Button } from "std-widgets.slint";
export component Recipe inherits Window {
    in-out property <int> counter: 0;
    callback button-pressed <=> button.clicked;
    VerticalBox {
        button := Button {
            text: "Button, pressed " + root.counter + " times";
        }
    }
}
```

slint

The `<=>` syntax binds two callbacks together. Here the new `button-pressed` callback binds to `button.clicked`.

The root element of the main component exposes all non-`private` properties and callbacks to native code.

In Slint, `-` and `_` are equivalent and interchangeable in all identifiers. This is different in native code: Most programming languages forbid `-` in identifiers, so `-` is replaced with `_`.

Rust

C++

Python

For technical reasons, this example uses `export {Recipe}` in the `slint!` macro. In real code, you can put the whole Slint code in the `slint!` macro, or use an external `.slint` file together with a build script.

```
slint::slint!(export { Recipe } from "docs/reference/src/recipes/button_native" rust
fn main() {
    let recipe = Recipe::new().unwrap();
    let recipe_weak = recipe.as_weak();
    recipe.on_button_pressed(move || {
        let recipe = recipe_weak.upgrade().unwrap();
        let mut value = recipe.get_counter();
        value = value + 1;
        recipe.set_counter(value);
    });
    recipe.run().unwrap();
}
```

The Slint compiler generates a `struct Recipe` with a getter (`get_counter`) and a setter (`set_counter`) for each accessible property of the root element of the `Recipe` component. It also generates a function for each accessible callback, like in this case `on_button_pressed`.

The `Recipe` struct implements the [`slint::ComponentHandle`](#) trait. A component manages a strong and a weak reference count, similar to an `Rc`. We call the `as_weak` function to get a weak handle to the component, which we can move into the callback.

We can't use a strong handle here, because that would form a cycle: The component handle has ownership of the callback, which itself has ownership of the closure's captured variables.

Use Property Bindings to Synchronize Controls

```
import { VerticalBox, Slider } from "std-widgets.slint";  
export component Recipe inherits Window {  
    VerticalBox {  
        slider := Slider {  
            maximum: 100;  
        }  
        Text {  
            text: "Value: \{round(slider.value)\}";  
        }  
    }  
}
```

slint

This example introduces the `Slider` widget.

It also introduces interpolation in string literals: Use `\{...\}` to render the result of code between the curly braces as a string.

Animation Examples

Animate the Position of an Element

```
import { CheckBox } from "std-widgets.slint";  
export component Recipe inherits Window {  
    width: 200px;  
    height: 100px;  
  
    rect := Rectangle {  
        x: 0;  
        y: 5px;  
        width: 40px;  
        height: 40px;  
        background: blue;  
    }  
}
```

slint

```

        animate x {
            duration: 500ms;
            easing: ease-in-out;
        }
    }

CheckBox {
    y: 25px;
    text: "Align rect to the right";
    toggled => {
        if (self.checked) {
            rect.x = parent.width - rect.width;
        } else {
            rect.x = 0px;
        }
    }
}

```

Layouts position elements automatically. In this example we manually position elements instead, using the `x` , `y` , `width` , `height` properties.

Notice the `animate x` block that specifies an animation. It's run whenever the property changes: Either because a callback sets the property, or because its binding value changes.

Animation Sequence

```

import { CheckBox } from "std-widgets.slint";          slint
export component Recipe inherits Window {
    width: 200px;
    height: 100px;

    rect := Rectangle {
        x: 0;

```

```

y: 5px;
width: 40px;
height: 40px;
background: blue;
animate x {
    duration: 500ms;
    easing: ease-in-out;
}
animate y {
    duration: 250ms;
    delay: 500ms;
    easing: ease-in;
}
}

CheckBox {
    y: 25px;
    text: "Align rect bottom right";
    toggled => {
        if (self.checked) {
            rect.x = parent.width - rect.width;
            rect.y = parent.height - rect.height;
        } else {
            rect.x = 0px;
            rect.y = 0px;
        }
    }
}

```

This example uses the `delay` property to make one animation run after another.

States Examples

Associate Property Values With States

```
import { HorizontalBox, VerticalBox, Button } from "std-widgets.slint";  
  
component Circle inherits Rectangle {  
    width: 30px;  
    height: 30px;  
    border-radius: root.width / 2;  
    animate x { duration: 250ms; easing: ease-in; }  
    animate y { duration: 250ms; easing: ease-in-out; }  
    animate background { duration: 250ms; }  
}  
  
export component Recipe inherits Window {  
    states [  
        left-aligned when b1.pressed: {  
            circle1.x: 0px; circle1.y: 40px; circle1.background: green;  
            circle2.x: 0px; circle2.y: 0px; circle2.background: blue;  
        }  
        right-aligned when b2.pressed: {  
            circle1.x: 170px; circle1.y: 70px; circle1.background: green;  
            circle2.x: 170px; circle2.y: 0px; circle2.background: blue;  
        }  
    ]  
  
    VerticalBox {  
        HorizontalBox {  
            max-height: self.min-height;  
            b1 := Button {  
                text: "State 1";  
            }  
            b2 := Button {  
                text: "State 2";  
            }  
        }  
    }  
}
```

```

Rectangle {
    background: root.background.darker(20%);
    width: 200px;
    height: 100px;

    circle1 := Circle { y:0; background: green; x: 85px; }
    circle2 := Circle { background: green; x: 85px; y: 40px; }
}

}

```

Transitions

```

import { HorizontalBox, VerticalBox, Button } from "std-widgets.slint";      lint

component Circle inherits Rectangle {
    width: 30px;
    height: 30px;
    border-radius: root.width / 2;
}

export component Recipe inherits Window {
    states [
        left-aligned when b1.pressed: {
            circle1.x: 0px; circle1.y: 40px;
            circle2.x: 0px; circle2.y: 0px;
            in {
                animate circle1.x, circle2.x { duration: 250ms; }
            }
            out {
                animate circle1.x, circle2.x { duration: 500ms; }
            }
        }
        right-aligned when !b1.pressed: {
    
```

```

        circle1.x: 170px; circle1.y: 70px;
        circle2.x: 170px; circle2.y: 00px;
    }
}

VerticalBox {
    HorizontalBox {
        max-height: self.min-height;
        b1 := Button {
            text: "Press and hold to change state";
        }
    }
    Rectangle {
        background: root.background.darker(20%);
        width: 250px;
        height: 100px;

        circle1 := Circle { y:0; background: green; x: 85px; }
        circle2 := Circle { background: blue; x: 85px; y: 40px; }
    }
}
}

```

Layout Examples

Vertical

```

import { VerticalBox, Button } from "std-widgets.slint";
export component Recipe inherits Window {
    VerticalBox {
        Button { text: "First"; }
        Button { text: "Second"; }
        Button { text: "Third"; }
    }
}

```

slint

```
}
```

Horizontal

```
import { HorizontalBox, Button } from "std-widgets.slint";  
export component Recipe inherits Window {  
    HorizontalBox {  
        Button { text: "First"; }  
        Button { text: "Second"; }  
        Button { text: "Third"; }  
    }  
}
```

slint

Grid

```
import { GridBox, Button, Slider } from "std-widgets.slint";  
export component Recipe inherits Window {  
    GridBox {  
        Row {  
            Button { text: "First"; }  
            Button { text: "Second"; }  
        }  
        Row {  
            Button { text: "Third"; }  
            Button { text: "Fourth"; }  
        }  
        Row {  
            Slider {  
                colspan: 2;  
            }  
        }  
    }  
}
```

slint

```
}
```

Global Callbacks

Invoke a Globally Registered Native Callback from Slint

This example uses a global singleton to implement common logic in native code. This singleton may also store properties that are accessible to native code.

Note: The preview visualize the Slint code only. It's not connected to the native code.

```
import { HorizontalBox, VerticalBox, LineEdit } from "std-widgets.slint";slint

export global Logic {
    pure callback to-upper-case(string) -> string;
    // You can collect other global properties here
}

export component Recipe inherits Window {
    VerticalBox {
        input := LineEdit {
            text: "Text to be transformed";
        }
        HorizontalBox {
            Text { text: "Transformed:"; }
            // Callback invoked in binding expression
            Text {
                text: {
                    Logic.to-upper-case(input.text);
                }
            }
        }
    }
}
```

```
}
```

[Rust](#)[C++](#)[NodeJS](#)[Python](#)

In Rust you can set the callback like this:

```
fn main() {
    let recipe = Recipe::new().unwrap();
    recipe.global::<Logic>().on_to_upper_case(|string| {
        string.as_str().to_uppercase().into()
    });
    // ...
}
```

rust

Custom Widgets

Custom Button

```
component Button inherits Rectangle {
    in-out property text <=> txt.text;
    callback clicked <=> touch.clicked;
    border-radius: root.height / 2;
    border-width: 1px;
    border-color: root.background.darker(25%);
    background: touch.pressed ? #6b8282 : touch.has-hover ? #6c616c : #456;
    height: txt.preferred-height * 1.33;
    min-width: txt.preferred-width + 20px;
    txt := Text {
        x: (parent.width - self.width)/2 + (touch.pressed ? 2px : 0);
        y: (parent.height - self.height)/2 + (touch.pressed ? 1px : 0);
        color: touch.pressed ? #fff : #eee;
    }
    touch := TouchArea { }
}
```

slint

```

export component Recipe inherits Window {
    VerticalLayout {
        alignment: start;
        Button { text: "Button"; }
    }
}

```

ToggleSwitch

```

export component ToggleSwitch inherits Rectangle {
    callback toggled;
    in-out property <string> text;
    in-out property <bool> checked;
    in-out property<bool> enabled <=> touch-area.enabled;
    height: 20px;
    horizontal-stretch: 0;
    vertical-stretch: 0;

    HorizontalLayout {
        spacing: 8px;
        indicator := Rectangle {
            width: 40px;
            border-width: 1px;
            border-radius: root.height / 2;
            border-color: self.background.darker(25%);
            background: root.enabled ? (root.checked ? blue: white) : white;
            animate background { duration: 100ms; }

            bubble := Rectangle {
                width: root.height - 8px;
                height: bubble.width;
                border-radius: bubble.height / 2;
                y: 4px;
                x: 4px + self.a * (indicator.width - bubble.width - 8px);
            }
        }
    }
}

```

```

        property <float> a: root.checked ? 1 : 0;
        background: root.checked ? white : (root.enabled ? blue : gray
        animate a, background { duration: 200ms; easing: ease; }
    }

}

Text {
    min-width: max(100px, self.preferred-width);
    text: root.text;
    vertical-align: center;
    color: root.enabled ? black : gray;
}

}

touch-area := TouchArea {
    width: root.width;
    height: root.height;
    clicked => {
        if (root.enabled) {
            root.checked = !root.checked;
            root.toggled();
        }
    }
}

export component Recipe inherits Window {
    VerticalLayout {
        alignment: start;
        ToggleSwitch { text: "Toggle me"; }
        ToggleSwitch { text: "Disabled"; enabled: false; }
    }
}

```

CustomSlider

The `TouchArea` is covering the entire widget, so you can drag this slider from any point within itself.

```
import { VerticalBox } from "std-widgets.slint";  
  
export component MySlider inherits Rectangle {  
    in-out property<float> maximum: 100;  
    in-out property<float> minimum: 0;  
    in-out property<float> value;  
  
    min-height: 24px;  
    min-width: 100px;  
    horizontal-stretch: 1;  
    vertical-stretch: 0;  
  
    border-radius: root.height/2;  
    background: touch.pressed ? #eee: #ddd;  
    border-width: 1px;  
    border-color: root.background.darker(25%);  
  
    handle := Rectangle {  
        width: self.height;  
        height: parent.height;  
        border-width: 3px;  
        border-radius: self.height / 2;  
        background: touch.pressed ? #f8f: touch.has-hover ? #66f : #0000ff;  
        border-color: self.background.darker(15%);  
        x: (root.width - handle.width) * (root.value - root.minimum)/(root.maximum - root.minimum);  
    }  
    touch := TouchArea {  
        property <float> pressed-value;  
        pointer-event(event) => {  
            if (event.button == PointerEventButton.left && event.kind == PointerEventKind.down)  
                self.pressed-value = root.value;  
        }  
    }  
}
```

lint

```

        }
    }

    moved => {
        if (self.enabled && self.pressed) {
            root.value = max(root.minimum, min(root.maximum,
                self.pressed-value + (touch.mouse-x - touch.pressed-x) * (
                    1 - self.pressed)
            ))
        }
    }
}

export component Recipe inherits Window {
    VerticalBox {
        alignment: start;
        slider := MySlider {
            maximum: 100;
        }
        Text {
            text: "Value: \{round(slider.value)\}";
        }
    }
}

```

This example shows another implementation that has a drag-able handle: The handle only moves when we click on that handle. The TouchArea is within the handle and moves with the handle.

```

import { VerticalBox } from "std-widgets.slint";          slint

export component MySlider inherits Rectangle {
    in-out property<float> maximum: 100;
    in-out property<float> minimum: 0;
    in-out property<float> value;

    min-height: 24px;
}

```

```

min-width: 100px;
horizontal-stretch: 1;
vertical-stretch: 0;

border-radius: root.height/2;
background: touch.pressed ? #eee: #ddd;
border-width: 1px;
border-color: root.background.darker(25%);

handle := Rectangle {
    width: self.height;
    height: parent.height;
    border-width: 3px;
    border-radius: self.height / 2;
    background: touch.pressed ? #f8f: touch.has-hover ? #66f : #0000ff;
    border-color: self.background.darker(15%);
    x: (root.width - handle.width) * (root.value - root.minimum)/(root.maximum - root.minimum);
}

touch := TouchArea {
    moved => {
        if (self.enabled && self.pressed) {
            root.value = max(root.minimum, min(root.maximum,
                root.value + (self.mouse-x - self.pressed-x) * (root.maximum - root.minimum)));
        }
    }
}
}

export component Recipe inherits Window {
    VerticalBox {
        alignment: start;
        slider := MySlider {
            maximum: 100;
        }
        Text {

```

```

        text: "Value: \{round(slider.value)\}";
    }
}
}

```

Custom Tabs

Use this recipe as a basis to when you want to create your own custom tab widget.

```

import { Button } from "std-widgets.slint";          lint

export component Recipe inherits Window {
    preferred-height: 200px;
    in-out property <int> active-tab;
    VerticalLayout {
        tab_bar := HorizontalLayout {
            spacing: 3px;
            Button {
                text: "Red";
                clicked => { root.active-tab = 0; }
            }
            Button {
                text: "Blue";
                clicked => { root.active-tab = 1; }
            }
            Button {
                text: "Green";
                clicked => { root.active-tab = 2; }
            }
        }
        Rectangle {
            clip: true;
            Rectangle {
                background: red;
                x: root.active-tab == 0 ? 0 : root.active-tab < 0 ? - self.wid

```

```

        animate x { duration: 125ms; easing: ease; }

    }

    Rectangle {
        background: blue;
        x: root.active-tab == 1 ? 0 : root.active-tab < 1 ? - self.width : self.width;
        animate x { duration: 125ms; easing: ease; }
    }

    Rectangle {
        background: green;
        x: root.active-tab == 2 ? 0 : root.active-tab < 2 ? - self.width : self.width;
        animate x { duration: 125ms; easing: ease; }
    }

}

}

```

Custom Table View

Slint provides a table widget, but you can also do something custom based on a `ListView`.

```

import { VerticalBox, ListView } from "std-widgets.slint";          slint

component TableView inherits Rectangle {
    in property <[string]> columns;
    in property <[[string]]> values;

    private property <length> e: self.width / root.columns.length;
    private property <[length]> column_sizes: [
        root.e, root.e, root.e, root.e, root.e, root.e, root.e, root.e,
        root.e, root.e, root.e, root.e, root.e, root.e, root.e, root.e,
        root.e, root.e, root.e, root.e, root.e, root.e, root.e, root.e
    ];

    VerticalBox {
        padding: 5px;

```

```

HorizontalLayout {
    padding: 5px; spacing: 5px;
    vertical-stretch: 0;
    for title[idx] in root.columns : HorizontalLayout {
        width: root.column_sizes[idx];
        Text { overflow: elide; text: title; }
        Rectangle {
            width: 1px;
            background: gray;
            TouchArea {
                width: 10px;
                x: (parent.width - self.width) / 2;
                property <length> cached;
                pointer-event(event) => {
                    if (event.button == PointerEventButton.left && event.buttons > 0)
                        self.cached = root.column_sizes[idx];
                }
            }
            moved => {
                if (self.pressed) {
                    root.column_sizes[idx] += (self.mouse-x - self.last.x);
                    if (root.column_sizes[idx] < 0) {
                        root.column_sizes[idx] = 0;
                    }
                }
            }
            mouse-cursor: ew-resize;
        }
    }
}

ListView {
    for r in root.values : HorizontalLayout {
        padding: 5px;
        spacing: 5px;
        for t[idx] in r : HorizontalLayout {

```

```

        width: root.column_sizes[idx];
      Text { overflow: elide; text: t; }
    }
  }
}

export component Example inherits Window {
  TableView {
    columns: ["Device", "Mount Point", "Total", "Free"];
    values: [
      ["/dev/sda1", "/", "255GB", "82.2GB"] ,
      ["/dev/sda2", "/tmp", "60.5GB", "44.5GB"] ,
      ["/dev/sdb1", "/home", "255GB", "32.2GB"] ,
    ];
  }
}

```

Breakpoints for Responsive User Interfaces

Use recipe implements a responsive SideBar that collapses when the parent width is smaller than the given break-point. When clicking the Button, the SideBar expands again. Use the blue Splitter to resize the container and test the responsive behavior.

```

import { Button, Palette } from "std-widgets.slint";                      slint

export component SideBar inherits Rectangle {
  private property <bool> collapsed: root.reference-width < root.break-point

  /// Defines the reference width to check `break-point`.
  in-out property <length> reference-width;

  /// If `reference-width` is less `break-point` the `SideBar` collapses.
  in-out property <length> break-point: 600px;

```

```

/// Set the text of the expand button.
in-out property <string> expand-button-text;

width: 160px;

container := Rectangle {
    private property <bool> expanded;

    width: parent.width;
    background: Palette.background.darker(0.2);

    VerticalLayout {
        padding: 2px;
        alignment: start;

        HorizontalLayout {
            alignment: start;

            if (root.collapsed) : Button {
                checked: container.expanded;
                text: root.expand-button-text;

                clicked => {
                    container.expanded = !container.expanded;
                }
            }
        }
    }

    @children
}

states [
    expanded when container.expanded && root.collapsed : {
        width: 160px;

        in {

```

```

        animate width { duration: 200ms; }
    }
    out {
        animate width { duration: 200ms; }
    }
    in {
        animate width { duration: 200ms; }
    }
    out {
        animate width { duration: 200ms; }
    }
}
]

}

states [
collapsed when root.collapsed : {
    width: 62px;
}
]
}

component Splitter inherits TouchArea {
width: 4px;
mouse-cursor: ew-resize;

Rectangle {
    width: 100%;
    height: 100%;
    background: blue;
}
}

export component SideBarTest inherits Window {
preferred-width: 700px;
min-height: 400px;
}

```

```

background: gray;

GridLayout {
    x: 0;
    width: splitter.x;

    Rectangle {
        height: 100%;
        col: 1;
        background: white;

        HorizontalLayout {
            padding: 8px;

            Text {
                color: black;
                text: "Content";
            }
        }
    }
}

SideBar {
    col: 0;
    reference-width: parent.width;
    expand-button-text: "E";
}
}

splitter := Splitter {
    x: root.width - self.width;
    height: 100%;

    moved => {
        self.x = min(root.width - self.width, max(400px, self.x + self.mouseDelta));
    }
}
}

```

Previous

Font Handling

Next

Best Practices

© 2025 SixtyFPS GmbH

Best Practices

In the following sections, we provide you with lessons we've learned of the years. This will help you avoid some pitfalls, tricky situations, and improve maintainability of your UI code.

Accessibility

When designing custom components, consider early on to declare [accessibility properties](#). At least a role, possibly a label, as well as actions.

If you're developing on Windows, then [Accessibility Insights](#)↗ tool for windows is a great tool to help you find and fix accessibility issues.

If you're developing on macOS, the [Accessibility Inspector](#)↗ offers similar functionality. Note that it requires the application to be built as bundle.

```
component CustomButton {  
    // ...  
    in property <string> text;  
  
    accessible-role: button;  
    accessible-label: self.text;  
    accessible-action-default => { /* simulate click */ }  
}
```

slint

Separate Code, UI, and Assets

Many projects start out small, with just a few files. But before you know it, your team grows, files get added, and it becomes harder to see forest for the trees. We recommend starting with the following basic directory structure:

```
my-project                                         plaintext
  └── src
      ├── main.cpp / main.rs / main.js / main.py
      │   <this is where your main business logic lives>
  └── ui
      ├── app-window.slint <the entry point for your Slint UI>
      ├── <additional .sling files here>
      └── images
          ├── logo.svg
          ├── highlight-marker.svg
          └── <all your images go here>
```

Translations

- When adding user-visible strings to your UI, consider early on to mark them as [translatable](#) by wrapping them in `@tr("...")`.
- Avoid `+` for concatenating strings, prefer `{}` substitutions. This gives translators the option of re-ordering the arguments for the most natural translation.

Caution

```
export component Example {
    property <string> name;

    Text {
        text: "Ink Level Controls"; // Ooops, forgot to mark as translatable
    }
    Text {
        text: @tr("Hello, ") + name; // Ooops, this is difficult to translate
    }
}
```

slint

```
}
```

Tip

```
export component Example {
    property <string> name;

    Text {
        text: @tr("Ink Level Controls");
    }
    Text {
        text: @tr("Hello, {}", name); // Good, now the translator can re-
    }
}
```

slint

Previous

Custom Controls

Next

Desktop

© 2025 SixtyFPS GmbH

Desktop

Generally, Slint runs on Windows, macOS, and popular Linux distributions. The following tables cover versions that we specifically test. The general objective is to support the operating systems that are supported by their vendors at the time of a Slint version release.

Windows	macOS	Linux
Operating System	Architecture	
Windows 10	x86-64	
Windows 11	x86-64, aarch64	

Handle the Console Window

When you running an application a console window will show by [default ↗](#).

Disable the console by specifying a `WINDOWS` subsystem.

When running the application from the command line, if the subsystem is set to windows it will no longer output stdout. To get it back consider using `FreeConsole()`.

See more details at [#3235 ↗](#)

Rust	C++	Python
Add the code to the top of <code>.rs</code> file which contains <code>fn main()</code> :		
<pre>#! [windows_subsystem = "windows"]</pre>		rust

Or if you want to keep console output in debug mode:

```
#![cfg_attr(not(debug_assertions), windows_subsystem = "windows")]
```

rust

Rust: Stack Overflows

When developing Rust applications on Windows, you might sooner or later observe the program aborting with `STATUS_STACK_OVERFLOW`, especially in debug builds. This is a known issue that's a combination of a high demand for stack space and MSVC defaulting to a stack size for the main thread that's significantly smaller compared to other operating systems.

This is fixed by configuring the linker. Create a [.cargo\config.toml ↗](#) file in your project (note the `.cargo` sub-directory) with the following contents:

```
[target.x86_64-pc-windows-msvc]
# Increase default stack size to avoid running out of stack
# space in debug builds. The size matches Linux's default.
rustflags = ["-C", "link-arg=/STACK:8000000"]

[target.aarch64-pc-windows-msvc]
# Increase default stack size to avoid running out of stack
# space in debug builds. The size matches Linux's default.
rustflags = ["-C", "link-arg=/STACK:8000000"]
```

toml

Other Platforms

[Contact us↗](#) if you want to use Slint on other platforms/versions.

Previous

Best Practices

Next

Embedded

© 2025 SixtyFPS GmbH

Embedded

Slint runs on many embedded platforms.

The platform descriptions below cover what has been tested for deployment. For the development environment, we recommend using a recent desktop operating system and compiler version.

Embedded Linux

Slint runs on a variety of embedded Linux platforms. Generally speaking, Slint requires a modern Linux userspace with working OpenGL ES 2.0 (or newer) or Vulkan drivers. We've had success running Slint on

- Yocto based distributions.
- BuildRoot based distributions.
- Torizon.

Yocto Linux

Torizon

For C++ applications see [meta-slint ↗](#) for recipes.

Rust applications work out of the box with Yocto's Rust support.

Microcontrollers

Slint's platform abstraction allows for integration into any Rust or C++ based Microcontroller development environment. Developers need to implement functionality to feed input events

such as touch or keyboard, as well as displaying the pixels rendered by Slint into a frame- or linebuffer.

C++

Rust

You will need to use the `mcu-board-support` crate. This crate re-export a `entry` attribute macro to apply to the `main` function, and a `init()` function that should be called before creating the Slint UI.

In order to use this backend, the final program must depend on both `slint` and `mcu-board-support`. The `main.rs` will look something like this

```
#![no_std]
#![cfg_attr(not(feature = "simulator"), no_main)]
slint::include_modules!();

#[allow(unused_imports)]
use mcu_board_support::prelude::*;

#[mcu_board_support::entry]
fn main() -> ! {
    mcu_board_support::init();
    MainWindow::new().run();
    panic!("The event loop should not return");
}
```

Since `mcu-board-support` is at the moment an internal crate not uploaded to crates.io, you must use the git version of `slint`, `slint-build`, and `mcu-board-support`

```
[dependencies]
slint = { git = "https://github.com/slnt-ui/slnt", default-features = false }
mcu-board-support = { git = "https://github.com/slnt-ui/slnt" }
# ...

[build-dependencies]
slint-build = { git = "https://github.com/slnt-ui/slnt" }
```

In your build.rs, you must include a call to `slint_build::print_rustc_flags().unwrap()` to set some of the flags.

Espressif (ESP32)

C++

To use Slint with your C++ application, you can follow the instructions on the [Espressif Documentation site ↗](#)

ST (STM32)

C++

Rust

Follow the steps below to run the Slint Printer Demo

STM32H735G-DK

Using [probe-rs ↗](#).

```
CARGO_PROFILE_RELEASE_OPT_LEVEL=s CARGO_TARGET_THUMBV7EM_NONE_EABIHF_RUNNER="p  
bash
```

Raspberry Pi (Pico)

Only Rust programs are currently supported on the Raspberry Pi Pico.

On the Raspberry Pi Pico

Ensure the right target is set:

```
rustup target add thumbv6m-none-eabi  
bash
```

Build the Slint Printer demo with:

```
cargo build -p printerdemo_mcu --no-default-features --features=mcu-board-sup  
bash
```

The resulting file can be flashed with [elf2uf2-rs ↗](#). Install it using:

```
cargo install elf2uf2-rs
```

bash

macOS

Linux

Now power off the Pico and connect it while holding down the "bootsel" button. The device will show up as a storage device with the name RPI-RP2 .

Then flash the demo to the Pico with:

```
elf2uf2-rs -d target/thumbv6m-none-eabi/release/printerdemo_mcu
```

bash

When the flashing completes the Pico will reboot and show the Slint Printer demo. The Mac will warn the drive was unmounted unexpectedly. This is expected and can be ignored.

On the Raspberry Pi Pico2

Build the Slint Printer demo with:

```
cargo build -p printerdemo_mcu --no-default-features --features=mcu-board-sup
```

bash

The resulting file can be flashed conveniently with [picotool ↗](#). You should build it from source.

Then upload the demo to the Raspberry Pi Pico: push the "bootsel" white button on the device while connecting the micro-usb cable to the device, this connects some USB storage on your workstation where you can store the binary.

Or from the command on linux (connect the device while pressing the "bootsel" button):

```
# If you're on Linux: mount the device
```

```
udisksctl mount -b /dev/sda1
```

bash

```
# upload
```

```
picotool load -u -v -x -t elf target/thumbv8m.main-none-eabihf/release/printer
```

Using probe-rs

This requires [probe-rs](#) and to connect the pico via a probe (for example another pico running the probe).

Then you can simply run with `cargo run`

```
CARGO_TARGET_THUMBV6M_NONE_EABI_LINKER="flip-link" CARGO_TARGET_THUMBV6M_NONE  
bash
```

Flashing and Debugging the Pico with `probe-rs`'s VSCode Plugin

Install `probe-rs-debugger` and the VSCode plugin as described [here](#).

Add this build task to your `.vscode/tasks.json`:

```
{  
  "version": "2.0.0",  
  "tasks": [  
    {  
      "type": "cargo",  
      "command": "build",  
      "args": [  
        "--package=printerdemo_mcu",  
        "--features=mcu-pico-st7789",  
        "--target=thumbv6m-none-eabi",  
        "--profile=release-with-debug"  
      ],  
      "problemMatcher": [  
        "$rustc"  
      ],  
      "group": "build",  
      "label": "build mcu demo for pico"  
    },  
  ]  
}
```

The `release-with-debug` profile is needed, because the debug build does not fit into flash.

You can define it like this in your top level `Cargo.toml` :

```
[profile.release-with-debug]
inherits = "release"
debug = true
```

toml

Now you can add the launch configuration to `.vscode/launch.json` :

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "preLaunchTask": "build mcu demo for pico",
      "type": "probe-rs-debug",
      "request": "launch",
      "name": "Flash and Debug MCU Demo",
      "cwd": "${workspaceFolder}",
      "connectUnderReset": false,
      "chip": "RP2040",
      "flashingConfig": {
        "flashingEnabled": true,
        "resetAfterFlashing": true,
        "haltAfterReset": true
      },
      "coreConfigs": [
        {
          "coreIndex": 0,
          "rttEnabled": true,
          "programBinary": "./target/thumbv6m-none-eabi/release-with"
        }
      ]
    },
  ]
}
```

json

This was tested using a second Raspberry Pi Pico programmed as a probe with [DapperMime](#) ↗.

Other Platforms

[Contact us](#) ↗ if you want to use Slint on other platforms/versions.

Previous

Desktop

Next

Android

© 2025 SixtyFPS GmbH

Android

Slint on Android is only supported for the Rust programming language. See the documentation of the [android module in our Rust API documentation](#).

Previous

Embedded

Next

iOS

© 2025 SixtyFPS GmbH

iOS

Note

When developing Slint applications for iOS, you can only use Rust as the programming language.

A Rust-based Slint application can be cross-compiled to iOS and runs on iPhones, iPads, and their respective simulators. This is implemented through the [Winit backend](#) and the [Skia Renderer](#).

Prerequisites

- A computer running macOS.
- An up-to-date installation of [Xcode↗](#).
- [Xcodegen↗](#).
- [Rust↗](#)
- The Rust device and simulator toolchains. Run `rustup target add aarch64-apple-ios` and `rustup target add aarch64-apple-ios-sim` to add them.

Adding iOS Support to an existing Rust Application

The following steps assume that you have a Rust application with Slint prepared. If you're just getting started, use our [Slint Rust Template↗](#) to get a minimal application running.

Use XCode to building, deploy, and submit iOS applications to the App Store. Use [Xcodegen](#) ↗ to create an Xcode project from a minimal description.

1. Verify that your application compiles for iOS, by running:

```
cargo build --target=aarch64-apple-ios
```

bash

2. Create a file called `project.yml` with the following contents:

```
name: My App
options:
  bundleIdPrefix: com.company
settings:
  ENABLE_USER_SCRIPT_SANDBOXING: NO
targets:
  MyApp:
    type: application
    platform: iOS
    deploymentTarget: "12.0"
    info:
      path: Info.plist
      properties:
        UILaunchScreen:
          - ImageRespectSafeAreaInsets: false
sources: []
postCompileScripts:
  - script: |
    ./build_for_ios_with_cargo.bash slint-rust-template
outputFileLists:
  $TARGET_BUILD_DIR/$EXECUTABLE_PATH
```

Adjust the name, bundle id, and other fields as needed.

This configuration file delegates the build process to cargo through a shell script.

Note

The shell script is invoked with the name of the binary that cargo produces. Update it to match the name of your project.

2. In a new file called `build_for_ios_with_cargo.bash`, paste the following script code:

```
#!/usr/bin/env bash                                bash
# Copyright © SixtyFPS GmbH <info@slint.dev>
# SPDX-License-Identifier: MIT

set -euvx

# Fix up PATH to work around https://github.com/rust-lang/rust/issues/80817 and
export PATH="/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:$PATH:$HOME/.cargo/b

# based on https://github.com/mozilla/glean/blob/main/build-scripts/xc-univers

if [[ "$CONFIGURATION" != "Debug" ]]; then
    CARGO_PROFILE=release
    MAYBE_RELEASE=--release
else
    CARGO_PROFILE=debug
    MAYBE_RELEASE=
fi

# Make Cargo output cache files in Xcode's directories
export CARGO_TARGET_DIR="$DERIVED_FILE_DIR/cargo"

IS_SIMULATOR=0
if [ "${LLVM_TARGET_TRIPLE_SUFFIX-}" = "-simulator" ]; then
    IS_SIMULATOR=1
fi

executables=()
for arch in $ARCHS; do
    case "$arch" in
        arm64)
```

```

if [ $IS_SIMULATOR -eq 0 ]; then
    CARGO_TARGET=aarch64-apple-ios
else
    CARGO_TARGET=aarch64-apple-ios-sim
fi
;;
x86_64)
export CFLAGS_x86_64_apple_ios="-target x86_64-apple-ios"
CARGO_TARGET=x86_64-apple-ios
;;
esac

cargo build $MAYBE_RELEASE --target $CARGO_TARGET --bin $1

executables+=("$DERIVED_FILE_DIR/cargo/$CARGO_TARGET/$CARGO_PROFILE/$1")
done

# Combine executables, and place them at the output path excepted by Xcode
lipo -create -output "$TARGET_BUILD_DIR/$EXECUTABLE_PATH" "${executables[@]}"

# Force code signing every run for device builds (non-simulator)
if [ $IS_SIMULATOR -eq 0 ]; then
    codesign --force --sign "${EXPANDED_CODE_SIGN_IDENTITY}" \
        --entitlements "${TARGET_TEMP_DIR}/${PRODUCT_NAME}.app.xcent" \
        "${TARGET_BUILD_DIR}/${EXECUTABLE_PATH}"
fi

```

3. Make the script executable with `chmod +x build_for_ios_with_cargo.bash`.

4. Run `xcodegen` to create `My App.xcodeproj`, and open it in Xcode. Now you can build, deploy, and debug your iOS application.

Previous

Android

Next

Web

© 2025 SixtyFPS GmbH

Web

Caution

Only Rust supports using Slint with WebAssembly.

Slint applications run on desktop, embedded, and mobile platforms. Slint applications written with Rust can also be cross-compiled to WebAssembly (Wasm) and run in the web browser.

Note

Slint cross-compiled for Wasm uses the [Winit backend](#) with the FemtoVG renderer.

Slint renders your UI into a HTML `<canvas>` element using [WebGL](#), without using the DOM or CSS. This results in a consistent look across platforms, but also introduces some limitations:

- Slint renders text directly, instead of benefitting from the browser's text rendering.
- Accessibility features (such as screen readers) are not available.
- The UI does not behave like a typical web application, since it doesn't use standard HTML elements.

Because of these trade-offs, running Slint in the browser is currently not recommended for building general-purpose web applications. Instead, it is best suited for:

- Demos and examples that can run directly in the browser without requiring installation.
- Applications where the web is not the primary platform, but a consistent UI is still needed.

- Tools or dashboards where native-style rendering is more important than web integration.

Building for Wasm

For a step-by-step walkthrough, check out the last chapter of the [quickstart](#).

Below is a summary of the main steps:

In your `Cargo.toml`, set the crate type to "cdylib" and add [wasm-bindgen ↗](#) as a dependency for the "wasm" target:

```
[lib] toml
#...
crate-type = ["cdylib"]

[target.'cfg(target_family = "wasm")'.dependencies]
wasm-bindgen = { version = "0.2" }
```

Use the `wasm-bindgen(start)` attribute to mark the application's entry point. The UI is created and run as usual:

```
##[cfg(target_family = "wasm")]
use wasm_bindgen::prelude::*;

slint::include_modules!(); // or slint!(...)

#[cfg_attr(target_family = "wasm", wasm_bindgen(start))]
pub fn main() {
    // Usual application code
    let main_window = MainWindow::new().unwrap();
    main_window.run().unwrap();
}
```

Build the application using [wasm-pack ↗](#).

```
wasm-pack build --release --target web
```

bash

This creates a `pkg/` directory with `.wasm` and `.js` files, including a JavaScript file named after your package.

Import the wasm binary in your HTML file. Slint expects a `<canvas>` HTML element with `id = "canvas"`.

```
<canvas id="canvas"></canvas>
<script type="module">
  import init from "./pkg/YOUR_APPLICATION.js";
  init();
</script>
```

html

Replace `YOUR_APPLICATION` with the name of your crate.

Note

Many web browser load `.wasm` files only in trusted contexts. If during development you observe the browser producing permission errors, then you may need to serve files through a web server, instead of the file system directly.

Clipboard Support

Slint's text input widgets supports copy and paste when running in the browser. This is implemented using the browser's clipboard API via the `web-sys` crate. Because the clipboard API in `web-sys` is still unstable, you need to opt in by setting the following environment variable when building:

```
RUSTFLAGS=--cfg=web_sys_unstable_apis
```

bash

You can read more in the [web-sys unstable APIs documentation ↗](#).

Previous

iOS

Next

Other Platforms

© 2025 SixtyFPS GmbH

Other Platforms

[Contact us↗](#) if you want to use Slint on other platforms/versions.

Previous

Web

Next

Overview

© 2025 SixtyFPS GmbH

Backends & Renderers

In Slint, a backend is the module that encapsulates the interaction with the operating system, in particular the windowing sub-system. Multiple backends can be compiled into Slint and one backend is selected for use at run-time on application start-up. You can configure Slint without any built-in backends, and instead develop your own backend by implementing Slint's platform abstraction and window adapter interfaces.

The backend is selected as follows:

1. The developer provides their own backend and sets it programmatically.
2. Else, the backend is selected by the value of the `SLINT_BACKEND` environment variable, if it is set.
3. Else, backends are tried for initialization in the following order:
 1. qt
 2. winit
 3. linuxkms

The following table provides an overview over the built-in backends. For more information about the backend's capabilities and their configuration options, see the respective sub-pages.

Backend Name	Description	Built-in by Default
qt	The Qt library is used for windowing system integration, rendering, and native widget styling.	On Linux if Qt is installed

winit	The winit library is used to interact with the windowing system.	Yes
linuxkms	Linux's KMS/DRI infrastructure is used for rendering. No windowing system or compositor is required.	No

A backend is also responsible for selecting a renderer. See the [Renderers](#) section for an overview. Override the choice of renderer by adding the name to the `SLINT_BACKEND` environment variable, separated by a dash. For example if you want to choose the `winit` backend in combination with the `software` renderer, set `SLINT_BACKEND=winit-software`. Similarly, `SLINT_BACKEND=linuxkms-skia` chooses the `linuxkms` backend and then instructs the LinuxKMS backend to use Skia for rendering.

Renderers

Slint comes with different renderers that use different techniques and libraries to turn your scene of elements into pixels. Slint picks a renderer based on your choice of backend as well as the features you've selected at Slint compilation time.

C++ **Rust** NodeJS Python

With Rust, enable one of the [renderer-*](#) features.

Qt Renderer

The Qt renderer comes with the [Qt backend](#) and renders using QPainter:

- Software rendering, no GPU acceleration.
- Available only in the Qt backend.

Software Renderer

- Runs anywhere, highly portable, and lightweight.
- Software rendering, no GPU acceleration.
- Supports partial rendering.

- Supports line-by-line rendering (Rust only).
- Suitable for Microcontrollers.
- Some features haven't been implemented yet:
 - No support for `Path`.
 - No image rotation or smooth scaling.
 - No support for `drop-shadow-*` properties.
 - No support for `border-radius` in combination with `clip: true`.
 - No text stroking/outlining.
- Text rendering currently limited to western scripts.
- Available in the [Winit backend](#).
- Public [Rust](#) and [C++](#) API.

FemtoVG Renderer

- Highly portable.
- GPU acceleration with OpenGL (required). When selected as `renderer-femtovg-wgpu`, GPU acceleration with Metal, Vulkan, and Direct3D.
- Text and path rendering quality sometimes sub-optimal.
- Available in the [Winit backend](#) and [LinuxKMS backend](#).
- Public [Rust](#) API.

Skia Renderer

- Sophisticated GPU acceleration with OpenGL, Metal, Vulkan, and Direct3D.
- Heavy disk-footprint compared to other renderers.
- Available in the [Winit backend](#) and [LinuxKMS backend](#).
- Public [C++](#) API.

Troubleshooting

You may run into compile issues when enabling the Skia renderer. The following sections track issues we're aware of and how to resolve them.

- Compilation error on Windows with messages about multiple source files and unused linker input

You may see compile errors that contain this error and warning from clang-cl:

```
clang-cl: error: cannot specify '/Foobj/src/fonts/fontmgr_win.SkFontMgr_  
clang-cl: warning: Hausmann/.cargo/registry/src/index.crates.io-6f17d22b
```

The Skia sources are checked out in a path that's managed by Cargo, the Rust package manager. The error happens when that path contains spaces. By default that's in `%HOMEPATH%\.cargo`, which contains spaces if the login name contains spaces. To resolve this issue, set the `CARGO_HOME` environment variable to a path without spaces, such as `c:\cargo_home`.

- Compilation error when compiling for ARMv7 with hardware floating-point support

You may see compiler errors that contain this message:

```
Unable to generate bindings: ClangDiagnostic("/home/runner/work/slint/yo
```

The Skia build invokes clang in multiple occasions and is sensitive to compiler flags that affect the floating point abi (such as `-mfloat-abi=hard`), as they affect header file lookups.

To solve this, set the `BINDGEN_EXTRA_CLANG_ARGS` environment variable to contain the same flags that your build environment also passes to the C++ compiler.

For example, if you're building against a Yocto SDK, then you can find these flags in the `OECORE_TUNE_CCARGS` environment variable.

- Compilation error when linking on Windows

You may see compiler errors that contain this message:

```
error: linking with `link.exe` failed: exit code: 1120  
|  
...
```

```
= note: skunicode.lib(icu.SkLoadICU.obj) : error LNK2019: unresolved exte  
...  
skia.lib(skia.SkNWayCanvas.obj) : error LNK2001: unresolved external sy
```

The Skia build requires the use of Microsoft Visual Studio 2022 as compiler. Make sure to have the latest patches to the compiler installed.

- Compilation error on macOS:

The build fails and somewhere in the log output you see this message:

```
cargo:warning=xcrun: error: unable to lookup item 'PlatformVersion' from comma plaintext  
cargo:warning=xcrun: error: unable to lookup item 'PlatformVersion' in SDK '/L
```

This is due the build process calling `xcrun --show-sdk-platform-version` to determine the SDK version, and that's unfortunately not supported by the Xcode command line tools. To solve this issue, run the following command once:

```
sudo xcode-select -switch /Applications/Xcode.app/Contents/Developer plaintext
```

- Compilation error when cross-compiling with Yocto:

The build fails and towards the end of the log message you see this message:

```
error occurred: unknown target `arm-org-linux-gnueabi` plaintext
```

This is caused by an unfortunate combination of updates of the `cc` crate and the way the Skia Rust bindings use it. We anticipate solving this in a future release. In the meantime, you can work around this by downgrading the `cc` crate in your `Cargo.lock` file:

```
cargo update -p cc --precise 1.1.31 plaintext
```

Previous

Next

Other Platforms

LinuxKMS Backend

© 2025 SixtyFPS GmbH

LinuxKMS Backend

The LinuxKMS backend runs only on Linux and eliminates the need for a windowing system such as Wayland or X11. Instead it uses the following libraries and interface to render directly to the screen and react to touch, mouse, and keyboard input.

- OpenGL via KMS/DRI.
- Vulkan via the Vulkan KHR Display Extension.
- DRM dumb buffers for software rendering, as well as legacy LinuxFB rendering.
- libinput/libudev for input event handling from mice, touch screens, or keyboards.
- libseat for GPU and input device access without requiring root access. (optional)

Dependencies

For compilation, pkg-config is used to determine the location of the following required system libraries:

pkg-config package name	Package name on Debian based distros
gbm	libgbm-dev
xkbcommon	libxkbcommon-dev
libudev	libudev-dev
libseat	libseat-dev

Note

If you don't have `libseat` available on your target system, then instead of selecting `backend-linuxkms`, select `backend-linuxkms-noseat`. This variant of the LinuxKMS backend eliminates the need to have `libseat` installed, but in exchange requires running the application as a user that's privileged to access all input and DRM/KMS device files; typically that's the root user.

Renderers

The LinuxKMS backend supports different renderers. They can be explicitly selected for use through the `SLINT_BACKEND` environment variable.

Renderer name	Required Graphics APIs	<code>SLINT_BACKEND</code> value to select renderer
FemtoVG	OpenGL ES 2.0	<code>linuxkms-femtovg</code>
Skia	OpenGL ES 2.0, Vulkan	<code>linuxkms-skia-opengl</code> , <code>linuxkms-skia-vulkan</code> , or <code>linuxkms-skia-software</code>
Software	None	<code>linuxkms-software</code>

Note

This backend is still experimental. The backend has not undergone a great variety of testing on different devices and there are [known issues ↗](#).

Note

A mouse is supported as input device, but rendering of the mouse cursor only works with the Skia and FemtoVG renderers, not with the Slint software renderer.

Display Selection with OpenGL or Skia Software

FemtoVG uses OpenGL, and Skia - unless Vulkan is enabled - uses OpenGL, too. Linux's direct rendering manager (DRM) subsystem is used to configure display outputs. Slint defaults to selecting the first connected display and configures it at either its preferred resolution (if available) or its highest. Set the `SLINT_DRM_OUTPUT` environment variable to select a specific display. To get a list of available outputs, set `SLINT_DRM_OUTPUT` to `list`, run your program, and observe the output.

For example, the output may look like this on a laptop with a built-in screen (eDP-1) and an externally connected monitor (DP-3):

```
DRM Output List Requested:  
eDP-1 (connected: true)  
DP-1 (connected: false)  
DP-2 (connected: false)  
DP-3 (connected: true)  
DP-4 (connected: false)
```

plaintext

Setting `SLINT_DRM_OUTPUT` to `DP-3` will render on the second monitor.

To select a specific resolution and refresh rate (mode), set the `SLINT_DRM_MODE` variable. Set it to `list` and run your program to get a list of available modes. For example the program output could look like this:

```
DRM Mode List Requested:  
Index: 0 Width: 3840 Height: 2160 Refresh Rate: 60  
Index: 1 Width: 3840 Height: 2160 Refresh Rate: 50  
Index: 2 Width: 3840 Height: 2160 Refresh Rate: 30  
Index: 3 Width: 2560 Height: 1440 Refresh Rate: 59  
Index: 4 Width: 1920 Height: 1080 Refresh Rate: 60  
Index: 5 Width: 1680 Height: 1050 Refresh Rate: 59  
...
```

plaintext

Set `SLINT_DRM_MODE` to `4` to select 1920x1080@60.

Display Selection with Vulkan

When Skia's Vulkan feature is enabled, Skia will attempt use Vulkan's KHR Display extension to render directly to a connected screen. Slint defaults to selecting the first connected display and configures it at its highest available resolution and refresh rate. Set the `SLINT_VULKAN_DISPLAY` environment variable to select a specific display. To get a list of available outputs, set `SLINT_VULKAN_DISPLAY` to `list`, run your program, and observe the output.

For example, the output may look like this on a laptop with a built-in screen (index 0) and an externally connected monitor (index 1):

```
Vulkan Display List Requested:  
Index: 0 Name: monitor  
Index: 1 Name: monitor
```

plaintext

Setting `SLINT_VULKAN_DISPLAY` to `1` will render on the second monitor.

To select a specific resolution and refresh rate (mode), set the `SLINT_VULKAN_MODE` variable. Set it to `list` and run your program to get a list of available modes. For example the program output could look like this:

```
Vulkan Mode List Requested:  
Index: 0 Width: 3840 Height: 2160 Refresh Rate: 60  
Index: 1 Width: 3840 Height: 2160 Refresh Rate: 50  
Index: 2 Width: 3840 Height: 2160 Refresh Rate: 30  
Index: 3 Width: 2560 Height: 1440 Refresh Rate: 59  
Index: 4 Width: 1920 Height: 1080 Refresh Rate: 60  
Index: 5 Width: 1680 Height: 1050 Refresh Rate: 59  
...
```

plaintext

Set `SLINT_VULKAN_MODE` to `4` to select 1920x1080@60.

Configuring the Keyboard

By default the keyboard layout and model is assumed to be a US model and layout. Set the following environment variables to configure support for different keyboards:

- XKB_DEFAULT_LAYOUT : A comma separated list of layouts (languages) to include in the keymap. See the layouts section in [xkeyboard-config\(7\)](#) for a list of accepted language codes. for a list of supported layouts.
- XKB_DEFAULT_MODEL : The keyboard model by which to interpreter keys. See the models section in [xkeyboard-config\(7\)](#) for a list of accepted model codes.
- XKB_DEFAULT_VARIANT : A comma separated list of variants, one per layout, which configures layout specific variants. See the values in parentheses in the layouts section in [xkeyboard-config\(7\)](#) for a list of accepted variant codes.
- XKB_DEFAULT_OPTIONS : A comma separated list of options to configure layout-independent key combinations. See the options section in [xkeyboard-config\(7\)](#) for a list of accepted option codes.

Display Rotation

If your display's default orientation does not match the desired orientation of your user interface, then you can set the `SLINT_KMS_ROTATION` environment variable to instruct Slint to rotate at rendering time. Supported values are the rotation in degrees: `0` , `90` , `180` , and `270` .

Note that this variable merely rotates the rendering output. If you're using a touch screen attached to the same display, then you may need to configure it to also apply a rotation on the touch events generated. For configuring libinput's `LIBINPUT_CALIBRATION_MATRIX` see the [libinput Documentation](#) for a list of valid values. Values can typically be set by writing them into a rules file under `/etc/udev/rules.d` .

The following example configures libinput to apply a 90 degree clockwise rotation for any attached touch screen:

```
echo 'ENV{LIBINPUT_CALIBRATION_MATRIX}="0 -1 1 1 0 0"' > /etc/udev/rules.d/lib
udevadm control --reload-rules
udevadm trigger
```

Legacy LinuxFB Interface

For software rendering, DRM dumb buffers are the preferred default way of posting frame buffers to the display. If DRM dumb buffers are not supported, the LinuxKMS backend falls back to using the Linux legacy framebuffer interface (`/dev/fbX`).

To override this default and use only the legacy framebuffer interface, set the `SLINT_BACKEND_LINUXFB=1` environment variable.

Previous

Overview

Next

Qt Backend

© 2025 SixtyFPS GmbH

Qt Backend

The Qt backend uses the [Qt](#) library to interact with the windowing system, for rendering, as well as widget style for a native look and feel.

The Qt backend supports practically all relevant operating systems and windowing systems, including macOS, Windows, Linux with Wayland and X11, and direct full-screen rendering via KMS or proprietary drivers.

The Qt backend only supports software rendering at the moment. That means it runs with any graphics driver, but it does not utilize GPU hardware acceleration.

The compilation step will detect whether Qt is installed or not using the `qttypes` crate. See the instructions in the [qttypes documentation](#) on how to set environment variables to point to the Qt installation.

If Qt is not installed, the backend will be disabled, and Slint will fallback to another backend, usually the [Winit backend](#).

Configuration Options

The Qt backend reads and interprets the following environment variables:

Name	Accepted Values	Description
<code>SLINT_FULLSCREEN</code>	any value	If this variable is set, every window is shown in fullscreen mode.

Previous

LinuxKMS Backend

Next

Winit Backend

© 2025 SixtyFPS GmbH

Winit Backend

The Winit backend uses the [winit](#) library to interact with the windowing system.

The Winit backend supports practically all relevant operating systems and windowing systems, including macOS, Windows, Linux with Wayland and X11.

The Winit backend supports different renderers. They can be explicitly selected for use through the `SLINT_BACKEND` environment variable.

Renderer name	Supported/Required Graphics APIs	<code>SLINT_BACKEND</code> value to select renderer
FemtoVG	OpenGL	winit-femtovg
FemtoVG (WGPU)	Metal, Direct3D, Vulkan with (http://wgpu.rs)	winit-femtovg-wgpu
Skia	OpenGL, Metal, Direct3D, Software-rendering	winit-skia
Skia Software	Software-only rendering with Skia	winit-skia-software
Skia OpenGL	OpenGL rendering with Skia	winit-skia-opengl
software	Software-rendering, no GPU required	winit-software

If no renderer is explicitly set, the backend will first try to use the Skia renderer, if it was enabled at compile time. If that fails, it will fall back to the FemtoVG renderer, and if that also fails, it will use the software renderer.

Configuration Options

The Winit backend reads and interprets the following environment variables:

Name	Accepted Values	Description
SLINT_FULLSCREEN	any value N	If this variable is set, every window is shown in fullscreen mode.

Linux Dependencies

On Linux, the Winit backend requires either X11 or Wayland to be available. Support of either can be enabled or disabled at compile time by setting the `backend-winit-x11` or `backend-winit-wayland` features (instead of `backend-winit`).

For X11 the following runtime dependencies are required: `libx11-xcb`, `xinput`, `libxcursor`, `libxkbcommon-x11`, `libx11`. On Debian-based systems, these can be installed with:

```
sudo apt install libx11-xcb-dev xinput libxcursor-dev libxkbcommon-x11-dev lib  
bash
```

Previous

[Qt Backend](#)

© 2025 SixtyFPS GmbH

Types

Slint is a statically typed language and offers a rich range of primitive types.

Primitive Types

bool

`bool` default: false

boolean whose value can be either true or false .

string

`string` default: ""

Any sequence of utf-8 encoded characters surrounded by quotes is a string : "foo" .

```
export component Example inherits Text {  
    text: "hello";  
}
```

slint

Escape sequences may be embedded into strings to insert characters that would be hard to insert otherwise:

Escape	Result
\"	"
\\	\

\n	new line
\u{x}	where <code>x</code> is a hexadecimal number, expands to the unicode code point represented by this number
\{expression}	the result of evaluating the expression

Anything else following an unescaped `\` is an error.

Note

The `\{...\}` syntax is not valid within the `slint!` macro in Rust.

`is-empty` property is true when `string` doesn't contain anything.

```
export component LengthOfString {
    property<bool> empty: "".is-empty; // true
    property<bool> not-empty: "hello".is-empty; // false
}
```

slint

`character-count` property returns the number of [grapheme clusters](#).

```
export component CharacterCountOfString {
    property<int> empty: "".character-count; // 0
    property<int> hello: "hello".character-count; // 5
    property<int> hiragana: "あいうえお".character-count; // 5
    property<int> surrogate-pair: "😊𩫱".character-count; // 2
    property<int> variation-selectors: "👍".character-count; // 1
    property<int> combining-character: "ໄໝ".character-count; // 1
    property<int> zero-width-joiner: "𠂊".character-count; // 1
    property<int> region-indicator-character: "🇿🇼".character-count; // 2
    property<int> emoji-tag-sequences: "🇪🇸".character-count; // 1
}
```

slint

The `to-lowercase` and `to-uppercase` methods convert `string` to lowercase or uppercase according to the [Unicode Character Property ↗](#).

```
export component ChangeCaseOfString {  
    property<string> hello: "HELL0".to-lowercase(); // "hello"  
    property<string> bye: "tschüß".to-uppercase(); // "TSCHÜSS"  
    property<string> odysseus: "ὈΔΥΣΣΕΥΣ".to-lowercase(); // "οδυσσεύς"  
    property<string> new_year: "农历新年".to-uppercase(); // "农历新年"  
}
```

slint

Numeric Types

angle

`angle` default: `0deg`

Angle measurement, corresponds to a literal like `90deg` , `1.2rad` , `0 25turn`

duration

`duration` default: `0ms`

Type for the duration of animations. A suffix like `ms` (millisecond) or `s` (second) is used to indicate the precision.

float

`float` default: `0`

Signed, 32-bit floating point number. Numbers with a `%` suffix are automatically divided by 100, so for example `30%` is the same as `0.30` .

int

`int` default: `0`

Signed integral number.

length

`length` default: `0px`

The type used for `x`, `y`, `width` and `height` coordinates. Corresponds to a literal like `1px`, `1pt`, `1in`, `1mm`, or `1cm`. It can be converted to and from length provided the binding is run in a context where there is an access to the device pixel ratio.

percent

`percent` default: `0%`

Signed, 32-bit floating point number that is interpreted as percentage. Literal number assigned to properties of this type must have a `%` suffix.

physical-length

`physical-length` default: `0phx`

This is an amount of physical pixels. To convert from an integer to a length unit, one can simply multiply by `1px`. Or to convert from a length to a float, one can divide by `1phx`.

relative-font-size

`relative-font-size` default: `0rem`

Relative font size factor that is multiplied with the `Window.default-font-size` and can be converted to a `length`.

Please see the language specific API references how these types are mapped to the APIs of the different programming languages.

Color and Brush Types

brush

`brush` default: `transparent`

A brush is a special type that can be either initialized from a `color` or a `gradient`. See [Colors & Brushes](#).

color

`color` default: transparent

RGB color with an alpha channel, with 8 bit precision for each channel. CSS color names as well as the hexadecimal color encodings are supported, such as #RRGGBBAA or #RGB. See [Colors & Brushes](#).

Images

image

`image` default: empty image

The `image` type is a reference to an image.

In Slint, an image can be loaded from a file with the `@image-url("...")` construct. The address within the `@image-url` function must be a string literal and the image is resolved at compile time.

Slint looks for images in the following places:

1. The absolute path or the path relative to the current `.slint` file.
2. The include path used by the compiler to look up `.slint` files.

Loading image from `http` is only supported in [SlintPad](#).

Supported formats are SVG, and formats supported by the [image crate](#): AVIF, BMP, DDS, Farbfeld, GIF, HDR, ICO, JPEG, EXR, PNG, PNM, QOI, TGA, TIFF, WebP.

For Rust applications, not all formats are enabled by default. Enable them with the `image-default-formats` Cargo feature.

C++

Rust

NodeJS

Python

In Rust, properties or struct fields of the image type are mapped to [`slint::Image`](#).

Note

Some image formats can be disabled using cargo features to reduce binary size and speed up compilation.

Access an `image`'s dimension using its `width` and `height` properties.

```
export component Example inherits Window {  
    preferred-width: 150px;  
    preferred-height: 50px;  
  
    // Note: http URL only work on the web version.  
    in property <image> some_image: @image-url("https://slint.dev/logo/slnt-l  
  
    HorizontalLayout {  
        Text {  
            text: "The image is " + some_image.width + "x" + some_image.height  
        }  
  
        // Check the size to find out if the image is empty.  
        if some_image.width > 0 : Image {  
            source: some_image;  
        }  
    }  
}
```

It is also possible to load images supporting [9 slice scaling](#) (also called nine patch or border images) by adding a `nine-slice(...)` argument. The argument can have either one, two, or four numbers that specifies the size of the edges. The numbers are either `top right bottom left` or `vertical horizontal`, or one number for everything

```
// nine-slice scaling  
export component Example inherits Window {  
    width: 100px;  
    height: 150px;  
    VerticalLayout {  
        Image {
```

```
        source: @image-url("https://interactive-examples.mdn.mozilla.net/m
```

```
    }
```

```
}
```

```
}
```

See also the [Image element](#).

Animation

easing

`easing` default: linear

The `easing` type allows defining an easing curve for animations.

To specify an easing curve, use the values from the `Easing` namespace. For example you can use `Easing.ease-out` or `Easing.ease-in-quad`. The namespace consists of the following names (see [easings.net ↗](#) for a visual reference):

- `linear`
- `ease-in-quad`
- `ease-out-quad`
- `ease-in-out-quad`
- `ease`
- `ease-in`
- `ease-out`
- `ease-in-out`
- `ease-in-quart`
- `ease-out-quart`
- `ease-in-out-quart`
- `ease-in-quint`
- `ease-out-quint`
- `ease-in-out-quint`
- `ease-in-expo`

- ease-out-expo
- ease-in-out-expo
- ease-in-sine
- ease-out-sine
- ease-in-out-sine
- ease-in-back
- ease-out-back
- ease-in-out-back
- ease-in-circ
- ease-out-circ
- ease-in-out-circ
- ease-in-elastic
- ease-out-elastic
- ease-in-out-elastic
- ease-in-bounce
- ease-out-bounce
- ease-in-out-bounce
- cubic-bezier(a, b, c, d) as in CSS

Additionally, in expressions of type `easing`, those names are available directly.

```
struct AnimationData {
    curve: easing,
}

component Custom inherits Rectangle {
    property<AnimationData> animation: {
        // Using the Easing namespace.
        curve: Easing.ease-in-circ,
    };
}

animate x {
    // In easing expressions the names are available via global scope.
```

slint

```
    easing: ease-out-bounce;  
}  
}
```

Type Conversions

Slint supports conversions between different types. Explicit conversions are required to make the UI description more robust, but implicit conversions are allowed between some types for convenience.

The following conversions are possible:

- `int` can be converted implicitly to `float` and vice-versa. When converting from `float` to `int`, the value is truncated.
- `int` and `float` can be converted implicitly to `string`
- `physical-length`, `relative-font-size`, and `length` can be converted implicitly to each other only in context where the pixel ratio is known.
- the units type (`length`, `physical-length`, `duration`, ...) can't be converted to numbers (`float` or `int`) but they can be divided by themselves to result in a number. Similarly, a number can be multiplied by one of these unit. The idea is that one would multiply by `1px` or divide by `1px` to do such conversions
- The literal `0` can be converted to any of these types that have associated unit.
- Struct types convert with another struct type if they have the same property names and their types can be converted. The source struct can have either missing properties, or extra properties. But not both.
- Arrays generally don't convert between each other. Array literals can be converted if the element types are convertible.
- String can be converted to float by using the `to-float` function. That function returns `0` if the string isn't a valid number. You can check with `is-float()` if the string contains a valid number
- `float` can be converted to a formatted `string` using `to-fixed` and `to-precision` which can be passed the number of digits after the decimal point and the number

of significant digits respectively. They behave like their JavaScript counterparts [toFixed\(\)](#) and [toPrecision\(\)](#).

```
export component Example {  
    // OK: int converts to string  
    property<{a: string, b: int}> prop1: {a: 12, b: 12 };  
    // OK: even if a is missing, it will just have the default value ("")  
    property<{a: string, b: int}> prop2: { b: 12 };  
    // OK: even if c is too many, it will be discarded  
    property<{a: string, b: int}> prop3: { a: "x", b: 12, c: 42 };  
    // ERROR: b is missing and c is extra, this doesn't compile, because it co  
    // property<{a: string, b: int}> prop4: { a: "x", c: 42 };  
  
    property<string> xxx: "42.1";  
    property<float> xxx1: xxx.to-float(); // 42.1  
    property<bool> xxx2: xxx.is-float(); // true  
    property<int> xxx3: 45.8; // 45  
}
```

slint

Previous
Overview

Next
**Common Properties &
Callbacks**

Getting started

This tutorial shows you how to use the languages that Slint supports as the host programming language.

We recommend using [our editor integrations for Slint↗](#) for following this tutorial.

Slint has application templates you can use to create a project with dependencies already set up that follows recommended best practices.

Prerequisites

C++ NodeJS **Rust** Python

We recommend using [rust-analyzer↗](#) and [our editor integrations for Slint↗](#) for following this tutorial.

1. Download and extract the [ZIP archive↗](#) of the [Rust Template↗](#).
2. Rename the extracted directory and change into it:

```
mv slint-rust-template-main memory  
cd memory
```

bash

Configure the project

Replace the contents of `src/main.rs` with the following:

```
slint::include_modules!();
```

rust

```
fn main() -> Result<(), slint::PlatformError> {
    let main_window = MainWindow::new()?;
    main_window.run();
}
```

Replace the contents of `ui/app-window.slint` with the following:

```
export component MainWindow inherits Window {
    Text {
        text: "hello world";
        color: green;
    }
}
```

slint

Run the application

Run the example with `cargo run` and a window appears with the green “Hello World” greeting.

Previous

Introduction

Next

Memory Tile

Common Properties & Callbacks

The Slint elements have many common properties, callbacks and behavior. This page describes these properties and their usage.

Common Visual Properties

These properties are valid on all visual items. For example `Rectangle` , `Text` , and `layouts` . Non visual items such as `Timer` don't have these properties.

x, y

`length` default: `0px`

The position of the element relative to its parent.

z

`float` default: `0.0`

Allows to specify a different order to stack the items with its siblings. The value must be a compile time constant.

Note

Currently the `z` value is a compile time constant and cannot be changed at runtime.

width, height

`length` default: `0px`

The width and height of the element. When set, this overrides the default size.

opacity

```
component ImageInfo inherits Rectangle {  
    in property <float> img-opacity: 1.0;  
    background: transparent;  
    VerticalLayout {  
        spacing: 5px;  
        Image {  
            source: @image-url("elements/slnt-logo.png");  
            opacity: img-opacity;  
        }  
        Text {  
            text: "opacity: " + img-opacity;  
            color: white;  
            horizontal-alignment: center;  
        }  
    }  
}  
  
export component Example inherits Window {  
    width: 100px;  
    height: 310px;  
    background: transparent;  
    Rectangle {  
        background: #141414df;  
        border-radius: 10px;  
    }  
    VerticalLayout {  
        spacing: 15px;  
        padding-top: 10px;  
        padding-bottom: 10px;  
        ImageInfo {  
            img-opacity: 1.0;  
        }  
    }  
}
```

```
ImageInfo {  
    img-opacity: 0.6;  
}  
  
ImageInfo {  
    img-opacity: 0.3;  
}  
}  
}
```

float default: 1

A value between 0 and 1 (or a percentage) that is used to draw the element and its children with transparency. 0 is fully transparent (invisible), and 1 is fully opaque. The opacity is applied to the tree of child elements as if they were first drawn into an intermediate layer, and then the whole layer is rendered with this opacity.

The following example demonstrates the opacity property with children. Note the software renderer does not support layer opacity and this will result in a different end result as shown.

```
Rectangle {  
    opacity: 50%;  
}  
  
Rectangle {  
    x: 0;  
    y: 0;  
    width: 100px;  
    height: 100px;  
    background: blue;  
}
```

slint

```
Rectangle {  
    x: 50px;  
    y: 50px;  
    width: 100px;  
    height: 100px;  
    background: green;  
}
```

```
}
```

visible

`bool` default: true

When set to `false`, the element and all his children won't be drawn and not react to mouse input. The element will still take up layout space within any layout container.

absolute-position

`struct Point` (out) default: a struct with all default values

A common issue is that in a UI with many nested components it's useful to know their (x,y)position relative to the main window or screen. This convenience property gives easy read only access to that value.

It represents a point specifying the absolute position within the enclosing [Window](#) or [PopupWindow](#). It defines coordinates (x,y) relative to the enclosing Window or PopupWindow, but the reference frame is unspecified (could be screen, window, or popup coordinates).

Miscellaneous

cache-rendering-hint

`bool` default: false

When set to `true`, this provides a hint to the renderer to cache the contents of the element and all the children into an intermediate cached layer. For complex sub-trees that rarely change this may speed up the rendering, at the expense of increased memory consumption. Not all rendering backends support this, so this is merely a hint.

dialog-button-role

`enum DialogButtonRole` default: none

Specify that this is a button in a `Dialog`.

Common Callbacks

init()

Every element implicitly declares an `init` callback. You can assign a code block to it that will be invoked when the element is instantiated and after all properties are initialized with the value of their final binding. The order of invocation is from inside to outside. The following example will print "first", then "second", and then "third":

```
component MyButton inherits Rectangle {  
    in-out property <string> text: "Initial";  
    init => {  
        // If `text` is queried here, it will have the value "Hello".  
        debug("first");  
    }  
}  
  
component MyCheckBox inherits Rectangle {  
    init => { debug("second"); }  
}  
  
export component MyWindow inherits Window {  
    MyButton {  
        text: "Hello";  
        init => { debug("third"); }  
    }  
    MyCheckBox {  
    }  
}
```

slint

Don't use this callback to initialize properties, because this violates the declarative principle.

Even though the `init` callback exists on all components, it cannot be set from application code, i.e. an `on_init` function does not exist in the generated code. This is because the callback is invoked during the creation of the component, before you could call `on_init` to actually set it.

While the `init` callback can invoke other callbacks, e.g. one defined in a `global` section, and you *can* bind these in the backend, this doesn't work for statically-created components, including the window itself, because you need an instance to set the globals binding. But it is possible to use this for dynamically created components (for example ones behind an `if`):

```
export global SystemService {  
    // This callback can be implemented in native code using the Slint API  
    callback ensure_service_running();  
}  
  
component MySystemButton inherits Rectangle {  
    init => {  
        SystemService.ensure_service_running();  
    }  
    // ...  
}  
  
export component AppWindow inherits Window {  
    in property <bool> show-button: false;  
  
    // MySystemButton isn't initialized at first, only when show-button is set  
    // At that point, its init callback will call ensure_service_running()  
    if show-button : MySystemButton {}  
}
```

Accessibility Properties

Use the following `accessible-` properties to make your items interact well with software like screen readers, braille terminals and other software to make your application accessible. `accessible-role` must be set in order to be able to set any other accessible property or callback.

accessible-role

`enum AccessibleRole` default: the first enum value

The role of the element. This property is mandatory to be able to use any other accessible properties. It should be set to a constant value. (default value: `none` for most elements, but `text` for the Text element)

accessible-checkable

`bool` default: `false`

Whether the element is can be checked or not.

accessible-checked

`bool` default: `false`

Whether the element is checked or not. This maps to the "checked" state of checkboxes, radio buttons, and other widgets.

accessible-description

`string` default: `""`

The description for the current element.

accessible-enabled

`bool` default: `false`

Whether the element is enabled or not. This maps to the "enabled" state of most widgets. (default value: `true`)

accessible-expandable

`bool` default: `false`

Whether the element can be expanded or not. For example, a `ComboBox` widget should set this to true, as its selection can be changed via an expandable popup.

accessible-expanded

`bool` default: `false`

Whether the element is expanded or not. Applies to combo boxes, menu items, tree view items and other widgets.

accessible-label

`string` default: ""

The label for an interactive element. (default value: empty for most elements, or the value of the `text` property for Text elements)

accessible-value-maximum

`float` default: 0.0

The maximum value of the item. This is used for example by spin boxes.

accessible-value-minimum

`float` default: 0.0

The minimum value of the item.

accessible-value-step

`float` default: 0.0

The smallest increment or decrement by which the current value can change. This corresponds to the step by which a handle on a slider can be dragged.

accessible-value

`string` default: ""

The current value of the item.

accessible-placeholder-text

`string` default: ""

A placeholder text to use when the item's value is empty. Applies to text elements.

accessible-read-only

`bool` default: false

Whether the element's content can be edited. This maps to the "read-only" state of line edit and text edit widgets.

accessible-item-selectable

`bool` default: false

Whether the element can be selected or not.

accessible-item-selected

`bool` default: false

Whether the element is selected or not. This maps to the "is-selected" state of listview items.

accessible-item-index

`int` default: 0

The index (starting from 0) of this element in a group of similar elements. Applies to list items, radio buttons and other elements.

accessible-item-count

`int` default: 0

The total number of elements in a group. Applies to the parent container of a group of element such as list views, radio button groups or other grouping elements.

Accessibility Callbacks

You can also use the following callbacks that are going to be called by the accessibility framework:

accessible-action-default()

Invoked when the default action for this widget is requested (eg: pressed for a button).

accessible-action-set-value(string)

Invoked when the user wants to change the accessible value.

accessible-action-increment()

Invoked when the user requests to increment the value.

accessible-action-decrement()

Invoked when the user requests to decrement the value.

accessible-action-expand()

Invoked when the user requests to expand the widget (eg: disclose the list of available choices for a combo box).

Previous

[Primitive Types](#)

Next

[Colors & Brushes](#)

Colors & Brushes

Color literals follow the syntax of CSS:

```
export component Example inherits Window {  
    background: blue;  
    property<color> c1: #ffaaff;  
    property<brush> b2: Colors.red;  
}
```

slint

In addition to plain colors, many elements have properties that are of type `brush` instead of `color`. A brush is a type that can be either a color or gradient. The brush is then used to fill an element or draw the outline.

CSS Color names are only in scope in expressions of type `color` or `brush`. Otherwise, access colors from the `Colors` namespace.

Color Properties

The following properties are exposed:

red

green

blue

alpha

These properties are in the range 0-255.

Use the colors namespace to select colors by their name. For example you can use `Colors.aquamarine` or `Colors.bisque`. The entire list of names is very long. You can find a complete list in the [CSS Specification](#).

These color names are available in scope of `color` and `brush` expressions, as well as in the `Colors` namespace.

```
// Using the Colors namespace
background: Colors.aquamarine;

// Using the functions via global scope.
background: aquamarine;
```

slint

Global Color Functions

rgb(int, int, int) -> color

rgba(int, int, int, float) -> color

Return the color as in CSS. Like in CSS, these two functions are actually aliases that can take three or four parameters.

The first 3 parameters can be either number between 0 and 255, or a percentage with a `%` unit. The fourth value, if present, is an alpha value between 0 and 1.

Unlike in CSS, the commas are mandatory.

hsv(h: float, s: float, v: float) -> color

hsv(h: float, s: float, v: float, a: float) -> color

Returns a color using HSV (Hue, Saturation, Value) coordinates. The hue parameter is a float representing degrees (0-360) and wraps around (e.g., 480 becomes 120). The saturation, value, and optional alpha parameter are expected to be within the range of 0 and 1.

Color Methods

All colors and brushes define the following methods:

brighter(factor: float) -> brush

Returns a new color derived from this color but has its brightness increased by the specified factor. This is done by converting the color to the HSV color space and multiplying the brightness (value) with $(1 + \text{factor})$. For example if the factor is 0.5 (or for example 50%) the returned color is 50% brighter. Negative factors decrease the brightness.

darker(factor: float) -> brush

Returns a new color derived from this color but has its brightness decreased by the specified factor. This is done by converting the color to the HSV color space and dividing the brightness (value) by $(1 + \text{factor})$. For example if the factor is .5 (or for example 50%) the returned color is 50% darker. Negative factors increase the brightness.

mix(other: brush, factor: float) -> brush

Returns a new color that is a mix of this color and `other`. The specified factor is clamped to be between `0.0` and `1.0` and then applied to this color, while `1.0 - factor` is applied to `other`. For example `red.mix(green, 70%)` will have a stronger tone of red, while `red.mix(green, 30%)` will have a stronger tone of green.

transparentize(factor: float) -> brush

Returns a new color with the opacity decreased by `factor`. The transparency is obtained by multiplying the alpha channel by `(1 - factor)`.

with-alpha(alpha: float) -> brush

Returns a new color with the alpha value set to `alpha` (between 0 and 1)

to-hsv() -> { hue: float, saturation: float, value: float, alpha: float }

Converts this color to the HSV color space and returns a struct with the `hue`, `saturation`, `value`, and `alpha` fields. `hue` is between 0 and 360 while `saturation`, `value`, and `alpha` are between 0 and 1.

Linear Gradients

Linear gradients describe smooth, colorful surfaces. They're specified using an angle and a series of color stops. The colors will be linearly interpolated between the stops, aligned to an imaginary line that is rotated by the specified angle. This is called a linear gradient and is specified using the `@linear-gradient` macro with the following signature:

`@linear-gradient(angle, color percentage, color percentage, ...)`

The first parameter to the macro is an angle (see [Types](#)). The gradient line's starting point will be rotated by the specified value.

Following the initial angle is one or multiple color stops, described as a space separated pair of a `color` value and a `percentage`. The color specifies which value the linear color interpolation should reach at the specified percentage along the axis of the gradient.

The following example shows a rectangle that's filled with a linear gradient that starts with a light blue color, interpolates to a very light shade in the center and finishes with an orange tone:

```
export component Example inherits Window {  
  preferred-width: 100px;  
  preferred-height: 100px;  
  
  Rectangle {  
    background: @linear-gradient(90deg, #3f87a6 0%, #ebf  
  }  
}
```

slint

Radial Gradients

Radial gradients are like linear gradients but the colors are interpolated circularly instead of along a line. To describe a radial gradient, use the `@radial-gradient` macro with the following signature:

`@radial-gradient(circle, color percentage, color percentage, ...)`

The first parameter to the macro is always `circle` because only circular gradients are supported. The syntax is otherwise based on the CSS `radial-gradient` function.

Example:

```
export component Example inherits Window {  
  preferred-width: 100px;  
  preferred-height: 100px;  
  Rectangle {  
    background: @radial-gradient(circle, #f00 0%, #0f0 5  
  }  
}
```

slint

Conic Gradients

Conic gradients are gradients where the color transitions rotate around a center point (like the angle on a color wheel). To describe a conic gradient, use the `@conic-gradient` macro with the following signature:

`@conic-gradient(color angle, color angle, ...)`

The conic gradient is described by a series of color stops, each consisting of a color and an angle. The angle specifies where the color is placed along the circular sweep (0deg to 360deg). Colors are interpolated between the stops along the circular path.

Example:

```
export component Example inherits Window {  
    preferred-width: 100px;  
    preferred-height: 100px;  
    Rectangle {  
        background: @conic-gradient(#f00 0deg, #0f0 120deg,  
    }  
}
```

slint

This creates a color wheel effect with red at the top (0deg/360deg), green at 120 degrees, and blue at 240 degrees.

Known Limitation

Negative angles cannot be used directly in conic gradients (e.g., `#ff0000 -90deg`).

Instead, use one of these workarounds:

- Convert to positive angles: `-90deg` → `270deg`
- Use variables: property `<angle>` `start: -90deg;` then use `start` in the gradient
- Use explicit subtraction: `#ff0000 0deg - 90deg`

Previous

**Common Properties &
Callbacks**

Next

Timer

Timer

Note

Timer is not an actual element visible in the tree, therefore it doesn't have the common properties such as `x`, `y`, `width`, `height`, etc. It also doesn't take room in a layout and cannot have any children or be inherited from.

This example shows a timer that counts down from 10 to 0 every second:

```
import { Button } from "std-widgets.slint";  
export component Example inherits Window {  
    property <int> value: 10;  
    timer := Timer {  
        interval: 1s;  
        running: true;  
        triggered() => {  
            value -= 1;  
            if (value == 0) {  
                self.running = false;  
            }  
        }  
    }  
    HorizontalLayout {  
        Text { text: value; }  
        Button {  
            text: "Reset";  
            clicked() => { value = 10; timer.running = true; }  
        }  
    }  
}
```

```
}
```

Use the `Timer` pseudo-element to schedule a callback at a given interval. The timer is only running when the `running` property is set to `true`. To stop or start the timer, set that property to `true` or `false`. It can be also set to a binding expression. When already running, the timer will be restarted if the `interval` property is changed.

Caution

By default the `Timer` is always running `running: true`. This can result in constant CPU usage and power usage so ensure that you set `running` to `false` when you don't want the timer to run.

```
property <int> count: 0;                                         slint
Timer {
    interval: 8s; // every 8 seconds the timer will activate (tick)
    triggered() => { // The triggered callback activates every time the timer
        if count >= 5 {
            self.running = false; // stop the timer after 5 ticks
        }
        count += 1;
    }
}
```

Properties

interval

`duration` default: 0ms

The interval between timer ticks. This property is mandatory.

```
Timer {
    property <int> count: 0;
```

slint

```
interval: 250ms;  
triggered() => {  
    debug("count is:", count);  
    count += 1;  
}  
}
```

running

`bool` default: true

true if the timer is running.

```
Timer {  
    property <int> count: 0;  
    interval: 250ms;  
    running: false; // timer is not running  
    triggered() => {  
        debug("count is:", count);  
    }  
}
```

lint

Callbacks

triggered()

Invoked every time the timer ticks (every `interval`).

```
Timer {  
    property <int> count: 0;  
    interval: 250ms;  
    triggered() => {  
        debug("count is:", count);  
    }  
}
```

lint

Functions

start()

Start the timer (equivalent to setting `running` to true).

stop()

Stop the timer (equivalent to setting `running` to false).

restart()

Restarts the timer if it was previously started.

Previous

Colors & Brushes

Next

Image

© 2025 SixtyFPS GmbH

Image

```
Image {  
  source: @image-url("mini-banner.png");  
}
```

slint

Use the `Image` element to display an [image](#).

Properties

colorize

`brush` default: a transparent brush

When set, the image is used as an alpha mask and is drawn in the given color (or with the gradient).

```
Image {  
  source: @image-url("slint-logo-simple-dark.png");  
  colorize: darkorange;  
}
```

slint

horizontal-alignment

`enum ImageHorizontalAlignment` default: center

vertical-alignment

`enum ImageVerticalAlignment` default: center

Image Tiling

horizontal-tiling

```
enum ImageTiling default: none
```

vertical-tiling

```
enum ImageTiling default: none
```

```
Image {  
  width: 400px;  
  height: 400px;  
  source: @image-url("slint-logo.png");  
  horizontal-tiling: repeat;  
}
```

slint

```
Image {  
  width: 400px;  
  height: 400px;  
  source: @image-url("slint-logo.png");  
  horizontal-tiling: round;  
}
```

slint

```
Image {  
  width: 400px;  
  height: 400px;  
  source: @image-url("slint-logo.png");  
  vertical-tiling: repeat;  
}
```

slint

```
Image {  
  width: 400px;  
  height: 400px;  
  source: @image-url("slint-logo.png");  
  vertical-tiling: round;  
}
```

slint

```
}
```

```
Image {  
    width: 400px;  
    height: 400px;  
    source: @image-url("slint-logo.png");  
    vertical-tiling: round;  
    horizontal-tiling: round;  
}
```

slint

image-fit

enum ImageFit

default: `contain` when the `Image` element is part of a layout, `fill` otherwise

```
Image {  
    width: 200px; height: 50px;  
    source: @image-url("mini-banner.png");  
    image-fit: fill;  
}
```

slint

```
Image {  
    width: 250px; height: 40px;  
    source: @image-url("mini-banner.png");  
    image-fit: contain;  
}
```

slint

```
Image {  
    width: 250px; height: 250px;  
    source: @image-url("mini-banner.png");  
    image-fit: cover;  
}
```

slint

```
Image {
```

slint

```
    width: 400px; height: 400px;  
    source: @image-url("mini-banner.png");  
    image-fit: preserve;  
}
```

image-rendering

enum `ImageRendering` default: smooth

```
Image {  
    width: 800px;  
    source: @image-url("mini-banner.png");  
    image-rendering: smooth;  
}
```

slint

```
Image {  
    width: 800px;  
    source: @image-url("mini-banner.png");  
    image-rendering: pixelated;  
}
```

slint

Rotation

Rotates the text by the given angle around the specified origin point. The default origin point is the center of the element. When these properties are set, the `Image` can't have children.

```
Image {  
    x: 0;  
    y: 0;  
    source: @image-url("images/slint-logo.svg");  
    rotation-angle: 45deg;  
    rotation-origin-x: 0;  
    rotation-origin-y: 0;  
}
```

slint

rotation-angle

`angle` default: 0deg

rotation-origin-x

`length` default: self.width / 2

rotation-origin-y

`length` default: self.height / 2

Source Properties

source

`image` default: the empty image

The `image` type is a reference to an image. It's defined using the `@image-url("...")` construct. The address within the `@image-url` function must be known at compile time.

Slint looks for images in the following places:

1. The absolute path or the path relative to the current `.slint` file.
2. The include path used by the compiler to look up `.slint` files.

Access an `image`'s source dimension using its `source.width` and `source.height` properties.

```
export component Example inherits Window {  
  preferred-width: 150px;  
  preferred-height: 50px;  
  
  in property <image> some_image: @image-url("https://slint.dev/logo/slnt-l  
  Text {  
    text: "The image is " + some_image.width + "x" + some_image.height;  
  }  
}
```

slint

```
// nine-slice scaling
export component Example inherits Window {
    width: 100px;
    height: 150px;
    VerticalLayout {
        Image {
            source: @image-url("https://interactive-examples.mdn.mozilla.net/mozilla-test-image.png");
        }
    }
}
```

slint

Use the `@image-url` macro to specify the image's path.

source-clip-x

`int` default: 0

source-clip-y

`int` default: 0

source-clip-width

`int` default: `source.width - source.clip-x`

source-clip-height

`int` default: `source.height - source.clip-y`

Properties in source image coordinates that define the region of the source image that is rendered. By default the entire source image is visible:

Accessibility

Alternative text

Consider giving an alternative text description of your image by setting the `accessible-label` property:

```
Image {  
    width: 100px;  
    height: 100px;  
    source: @image-url("slint-logo.png");  
    accessible-label: "Slint logo";  
}
```

slint

Filtering out images for users of assistive technologies

By default, images have the `accessible-role` property set to `image`. If your image is purely decorative and doesn't convey any information, consider removing it from the accessibility tree:

```
Image {  
    source: @image-url("mini-banner.png");  
    accessible-role: none;  
}
```

slint

Previous
Timer

Next
Path

Path

The `Path` element allows rendering a generic shape, composed of different geometric commands. A path shape can be filled and outlined.

When not part of a layout, its width or height defaults to 100% of the parent element when not specified.

A path can be defined in two different ways:

- Using SVG path commands as a string
- Using path command elements in `.slint` markup.

The coordinates used in the geometric commands are within the imaginary coordinate system of the path. When rendering on the screen, the shape is drawn relative to the `x` and `y` properties. If the `width` and `height` properties are non-zero, then the entire shape is fit into these bounds - by scaling accordingly.

Properties

fill

`brush` default: a transparent brush

The color for filling the shape of the path.

fill-rule

`enum FillRule` default: nonzero

The fill rule to use for the path.

stroke

`brush` default: a transparent brush

The color for drawing the outline of the path.

stroke-width

`length` default: 0px

The width of the outline.

stroke-line-cap

`enum LineCap` default: butt

The appearance of the ends of the path's outline.

width

`length` default: 0px

If non-zero, the path will be scaled to fit into the specified width.

height

`length` default: 0px

If non-zero, the path will be scaled to fit into the specified height.

Viewbox Properties

viewbox-x

`float` default: 0.0

viewbox-y

`float` default: 0.0

viewbox-width

`float` default: 0.0

viewbox-height

`float` default: 0.0

These four properties allow defining the position and size of the viewport of the path in path coordinates.

If the `viewbox-width` or `viewbox-height` is less or equal than zero, the `viewbox` properties are ignored and instead the bounding rectangle of all path elements is used to define the view port.

clip

`bool` default: false

By default, when a path has a view box defined and the elements render outside of it, they are still rendered. When this property is set to `true`, then rendering will be clipped at the boundaries of the view box.

anti-alias

`bool` default: true

By default, the fill and stroke of a path is rendered with anti-aliasing, for best quality. Some GPUs have performance issues when rendering with anti-aliasing and animation. Setting the value to `false` might improve the frame-rate at the expense of a smoother looking path.

Path Using SVG Commands

SVG is a popular file format for defining scalable graphics, which are often composed of paths. In SVG paths are composed using [commands](#), which in turn are written in a string. In `.slint` the path commands are provided to the `commands` property. The following example renders a shape consists of an arc and a rectangle, composed of `line-to`, `move-to` and `arc` commands:

```
export component Example inherits Path {
```

slint

```
    width: 100px;  
    height: 100px;  
    commands: "M 0 0 L 0 100 A 1 1 0 0 0 100 100 L 100 0 Z";  
    stroke: red;  
    stroke-width: 1px;  
}
```

The commands are provided in a property:

Commands

`string` default: ""

A string providing the commands according to the SVG path specification. This property can only be set in a binding and cannot be accessed in an expression.

Path Using SVG Path Elements

The shape of the path can also be described using elements that resemble the SVG path commands but use the `.slint` markup syntax. The earlier example using SVG commands can also be written like that:

```
export component Example inherits Path {  
  width: 100px;  
  height: 100px;  
  stroke: blue;  
  stroke-width: 1px;  
  
  MoveTo {  
    x: 0;  
    y: 0;  
  }  
  LineTo {  
    x: 0;  
    y: 100;  
  }  
}
```

slint

```
ArcTo {  
    radius-x: 1;  
    radius-y: 1;  
    x: 100;  
    y: 100;  
}  
LineTo {  
    x: 100;  
    y: 0;  
}  
Close {  
}  
}
```

Note how the coordinates of the path elements don't use units - they operate within the imaginary coordinate system of the scalable path.

MoveTo Sub-Element for Path

The `MoveTo` sub-element closes the current sub-path, if present, and moves the current point to the location specified by the `x` and `y` properties. Subsequent elements such as `LineTo` will use this new position as their starting point, therefore this starts a new sub-path.

x

`float` default: `0.0`

The x position of the new current point.

y

`float` default: `0.0`

The y position of the new current point.

LineTo Sub-Element for Path

The `LineTo` sub-element describes a line from the path's current position to the location specified by the `x` and `y` properties.

x

`float` default: `0.0`

The target x position of the line.

y

`float` default: `0.0`

The target y position of the line.

ArcTo Sub-Element for Path

The `ArcTo` sub-element describes the portion of an ellipse. The arc is drawn from the path's current position to the location specified by the `x` and `y` properties. The remaining properties are modelled after the SVG specification and allow tuning visual features such as the direction or angle.

large-arc

`bool` default: `false`

Out of the two arcs of a closed ellipse, this flag selects that the larger arc is to be rendered. If the property is `false`, the shorter arc is rendered instead.

radius-x

`float` default: `0.0`

The x-radius of the ellipse.

radius-y

float default: 0.0

The y-radius of the ellipse.

sweep

bool default: false

If the property is true , the arc will be drawn as a clockwise turning arc; anti-clockwise otherwise.

x-rotation

float default: 0.0

The x-axis of the ellipse will be rotated by the value of this properties, specified in as angle in degrees from 0 to 360.

x

float default: 0.0

The target x position of the line.

y

float default: 0.0

The target y position of the line.

CubicTo Sub-Element for Path

The CubicTo sub-element describes a smooth Bézier from the path's current position to the location specified by the x and y properties, using two control points specified by their respective properties.

control-1-x

float default: 0.0

The x coordinate of the curve's first control point.

control-1-x

`float` default: `0.0`

The x coordinate of the curve's first control point.

control-1-y

`float` default: `0.0`

The y coordinate of the curve's first control point.

control-2-x

`float` default: `0.0`

The x coordinate of the curve's second control point.

x

`float` default: `0.0`

The target x position of the curve.

y

`float` default: `0.0`

The target y position of the curve.

QuadraticTo Sub-Element for Path

The QuadraticTo sub-element describes a smooth Bézier from the path's current position to the location specified by the `x` and `y` properties, using the control points specified by the `control-x` and `control-y` properties.

control-x

`float` default: `0.0`

The x coordinate of the curve's control point.

control-y

`float` default: `0.0`

The y coordinate of the curve's control point.

x

`float` default: `0.0`

The target x position of the curve.

y

`float` default: `0.0`

The target y position of the curve.

Close Sub-Element for Path

The `close` element closes the current sub-path and draws a straight line from the current position to the beginning of the path.

Previous

Image

Next

Rectangle

Rectangle

By default, a `Rectangle` is just an empty item that shows nothing. By setting a color or configuring a border, it's then possible to draw a rectangle on the screen.

When not part of a layout, its width and height default to 100% of the parent element.

```
export component ExampleRectangle inherits Window {  
    width: 200px; height: 800px; background: transparent;  
  
    Rectangle {  
        x: 10px; y: 10px;  
        width: 180px;  
        height: 180px;  
        background: #315afdf;  
    }  
  
    // Rectangle with a border  
    Rectangle {  
        x: 10px; y: 210px;  
        width: 180px;  
        height: 180px;  
        background: green;  
        border-width: 2px;  
        border-color: red;  
    }  
  
    // Transparent Rectangle with a border and a radius  
    Rectangle {  
        x: 10px; y: 410px;
```

```

        width: 180px;
        height: 180px;
        border-width: 4px;
        border-color: black;
        border-radius: 30px;
    }

// A radius of width/2 makes it a circle
Rectangle {
    x: 10px; y: 610px;
    width: 180px;
    height: 180px;
    background: yellow;
    border-width: 2px;
    border-color: blue;
    border-radius: self.width/2;
}
}

```

Properties

background

brush default: transparent

The background brush of this Rectangle .

```

property <brush> rainbow-gradient: @linear-gradient(40deg, r
slint

Rectangle {
    x: 10px;
    y: 10px;
    width: 180px;
    height: 180px;
    background: #315afad;
}

```

```
Rectangle {  
    x: 10px;  
    y: 210px;  
    width: 180px;  
    height: 180px;  
    background: rainbow-gradient;  
}
```

border-color

brush default: transparent

```
Rectangle {  
    width: 200px;  
    height: 200px;  
    border-width: 10px;  
    border-color: lightslategray;  
}
```

slint

The color of the border.

Caution

The default `border-width` is `0px`, so the border is invisible. After setting a color also ensure that the `border-width` is set to a non-zero value.

border-width

length default: 0

```
Rectangle {  
    width: 200px;  
    height: 200px;  
    border-width: 30px;
```

slint

```
    border-color: lightslategray;  
}
```

The width of the border.

clip

bool default: false

```
// clip: false; the default  
slint  
Rectangle {  
  x: 50px; y: 50px;  
  width: 150px;  
  height: 150px;  
  background: darkslategray;  
  Rectangle {  
    x: -40px; y: -40px;  
    width: 100px;  
    height: 100px;  
    background: lightslategray;  
  }  
}  
  
// clip: true; Clips the children of this Rectangle  
Rectangle {  
  x: 50px; y: 250px;  
  width: 150px;  
  height: 150px;  
  background: darkslategray;  
  clip: true;  
  Rectangle {  
    x: -40px; y: -40px;  
    width: 100px;  
    height: 100px;  
    background: lightslategray;  
  }  
}
```

```
}
```

By default, when child elements are outside the bounds of a parent, they are still shown.

When this property is set to `true`, the children of this `Rectangle` are clipped and only the contents inside the elements bounds are shown.

Border Radius Properties

`border-radius`

`length` default: 0

The size of the radius. This single value is applied to all four corners.

To target specific corners with different values use the following properties:

`border-top-left-radius`

`length` default: 0px

`border-top-right-radius`

`length` default: 0px

`border-bottom-left-radius`

`length` default: 0px

`border-bottom-right-radius`

Drop Shadows

To achieve the graphical effect of a visually elevated shape that shows a shadow effect underneath the frame of an element, it's possible to set the following `drop-shadow` properties:

`drop-shadow-blur`

`length` default: 0px

The radius of the shadow that also describes the level of blur applied to the shadow. Negative values are ignored and zero means no blur.

drop-shadow-color

`color` default: a transparent color

The base color of the shadow to use. Typically that color is the starting color of a gradient that fades into transparency.

drop-shadow-offset-x

`length` default: 0px

The horizontal distance of the shadow from the element's frame.

drop-shadow-offset-y

`length` default: 0px

The vertical distance of the shadow from the element's frame.

Previous

Path

Next

Text

Text

```
// text-example.slint                                         slint
export component TextExample inherits Window {
    // Text colored red.
    Text {
        x:0; y:0;
        text: "Hello World";
        color: red;
    }

    // This paragraph breaks into multiple lines of text.
    Text {
        x:0; y: 30px;
        text: "This paragraph breaks into multiple lines of text";
        wrap: word-wrap;
        width: 150px;
        height: 100%;
    }
}
```

A `Text` element for displaying text.

By default, the `min-width`, `min-height`, `preferred-width`, and `preferred-height` of a `Text` element are set to fit the full text as if it were displayed on a single line (unless the text contains explicit line breaks). However, if the `wrap` property is set to `word-wrap`, and/or if the `overflow` property is set to `elide`, the `min-width` is reduced to zero, allowing the text to wrap or be elided, while the `preferred-width` and `preferred-height` remain unchanged.

Properties

color

`brush` default: <depends on theme>

The color of the text.

```
Text {  
  text: "Hello";  
  color: #3586f4;  
  font-size: 40pt;  
}
```

slint

font-family

`string` default: ""

The name of the font family selected for rendering the text.

```
Text {  
  text: "CoMiC!";  
  color: black;  
  font-size: 40pt;  
  font-family: "Comic Sans MS";  
}
```

slint

Note

Make sure the font is loaded before using it in a `Text` element. See [FontHandling](#) for more.

font-size

`length` default: 0px

The font size of the text.

```
Text {  
  text: "Big";  
  color: black;  
  font-size: 70pt;  
}
```

slint

font-weight

`int` default: 0

The weight of the font. The values range from 100 (lightest) to 900 (thickest). 400 is the normal weight.

```
Text {  
  text: "BOLD";  
  color: black;  
  font-size: 30pt;  
  font-weight: 800;  
}
```

slint

font-italic

`bool` default: false

Whether or not the font face should be drawn italicized or not.

```
Text {  
  text: "Italic";  
  color: black;  
  font-italic: true;  
  font-size: 40pt;  
}
```

slint

font-metrics

`struct FontMetrics` default: a struct with all default values

The design metrics of the font scaled to the font pixel size used by the element.

horizontal-alignment

`enum TextHorizontalAlignment` default: the first enum value

```
Text {  
  x: 0;  
  text: "Hello";  
  color: black;  
  font-size: 40pt;  
  horizontal-alignment: left;  
}
```

slint

letter-spacing

`length` default: 0px

The letter spacing allows changing the spacing between the glyphs. A positive value increases the spacing and a negative value decreases the distance.

```
Text {  
  text: "Spaced!";  
  color: black;  
  font-size: 30pt;  
  letter-spacing: 4px;  
}
```

slint

overflow

`enum TextOverflow` default: the first enum value

text

`string` default: ""

The text rendered.

vertical-alignment

`enum TextVerticalAlignment` default: the first enum value

wrap

`enum TextWrap` default: the first enum value

```
Text {  
    text: "This paragraph breaks into multiple lines of text  
    font-size: 20pt;  
    wrap: word-wrap;  
    width: 180px;  
}
```

slint

stroke

`brush` default: a transparent brush

The brush used for the text outline.

```
Text {  
    text: "Stroke";  
    stroke-width: 2px;  
    stroke: darkblue;  
    stroke-style: center;  
    font-size: 80px;  
    color: lightblue;  
}
```

slint

stroke-width

`length` default: 0px

The width of the text outline. If the width is zero, then a hairline stroke (1 physical pixel) will be rendered.

stroke-style

enum `TextStrokeStyle` default: the first enum value

```
Text {  
  text: "Style";  
  stroke-width: 2px;  
  stroke: #3586f4;  
  stroke-style: center; // slint  
  font-size: 60px;  
  color: white;  
}
```

slint

Rotation

Rotates the text by the given angle around the specified origin point. The default origin point is the center of the element.

Caution

When these properties are set, the `Text` can't have children.

```
Text {  
  text: "I'm dizzy";  
  rotation-angle: 45deg;  
  rotation-origin-x: self.width / 2;  
  rotation-origin-y: self.height / 2;  
  font-size: 30pt;  
}
```

slint

rotation-angle

angle default: 0deg

rotation-origin-x

`length` default: `self.width / 2`

rotation-origin-y

`length` default: `self.height / 2`

Accessibility

By default, `Text` elements have the following accessibility properties set:

- `accessible-role: text;`
- `accessible-label: text;`

Previous

Rectangle

Next

Flickable

© 2025 SixtyFPS GmbH

Flickable

```
export component Example inherits Window {  
  width: 270px;  
  height: 100px;  
  
  Flickable {  
    viewport-height: 300px;  
    Text {  
      x: 0;  
      y: 150px;  
      text: "This is some text that you have to scroll to see";  
    }  
  }  
}
```

slint

The `Flickable` is a low-level element that is the base for scrollable widgets, such as the `ScrollView`. When the `viewport-width` or the `viewport-height` is greater than the parent's `width` or `height` respectively, the element becomes scrollable. Note that the `Flickable` doesn't create a scrollbar. When unset, the `viewport-width` and `viewport-height` are calculated automatically based on the `Flickable`'s children. This isn't the case when using a `for` loop to populate the elements. This is a bug tracked in issue [#407 ↗](#). The maximum and preferred size of the `Flickable` are based on the viewport.

When not part of a layout, its width or height defaults to 100% of the parent element when not specified.

Pointer Event Interaction

If the `Flickable`'s area contains elements that use `TouchArea` to act on clicking, such as `Button` widgets, then the following algorithm is used to distinguish between the user's intent of scrolling or interacting with `TouchArea` elements:

1. If the `Flickable`'s `interactive` property is `false`, all events are forwarded to elements underneath.
2. If a press event is received where the event's coordinates interact with a `TouchArea`, the event is stored and any subsequent move and release events are handled as follows:
 1. If 100ms elapse without any events, the stored press event is delivered to the `TouchArea`.
 2. If a release event is received before 100ms have elapsed, the stored press event as well as the release event are immediately delivered to the `TouchArea` and the algorithm resets.
 3. Any move events received will start a flicking operation on the `Flickable` if all of the following conditions are met:
 1. The event is received before 500ms have elapsed since receiving the press event.
 2. The distance to the press event exceeds 8 logical pixels in an orientation in which we are allowed to move. If `Flickable` decides to flick, any press event sent previously to a `TouchArea`, is followed up by an exit event. During the phase of receiving move events, the flickable follows the coordinates.
 3. If the interaction of press, move, and release events begins at coordinates that do not intersect with a `TouchArea`, then `Flickable` will flick immediately on pointer move events when the euclidean distance to the coordinates of the press event exceeds 8 logical pixels.

All pointer and mouse events that occur within the bounds of a `Flickable` are intercepted by the `Flickable` itself and are not propagated to any elements underneath it.

Wheel/Scroll Event Interaction

The `Flickable` also supports scrolling with the mouse wheel and touchpad scroll gestures. It will scroll regardless of the `interactive` property. If the `Flickable` can scroll in the event's direction, the event will be intercepted. If the `Flickable` can't scroll in the direction of the event, the event will be forwarded to the parent.

Properties

interactive

`bool` default: `true`

```
Flickable {  
    interactive: false;  
}
```

slint

When true, the viewport can be scrolled by clicking on it and dragging it with the cursor.

viewport-width

`length` default: `0px`

The total width of the scrollable element.

viewport-height

`length` default: `0px`

The total height of the scrollable element.

viewport-x

`length` default: `0px`

The position of the scrollable element relative to the `Flickable`. This is usually a negative value.

viewport-y

`length` default: `0px`

The position of the scrollable element relative to the `Flickable`. This is usually a negative value.

Callbacks

flicked()

Invoked when `viewport-x` or `viewport-y` is changed by a user action (dragging, scrolling).

Previous

Text

Next

SwipeGestureHandler

© 2025 SixtyFPS GmbH

SwipeGestureHandler

Use the `SwipeGestureHandler` to handle swipe gesture in some particular direction.
Recognition is limited to the element's geometry.

```
export component Example inherits Window {  
  width: 270px;  
  height: 100px;  
  
  property <int> current-page: 0;  
  
  sgr := SwipeGestureHandler {  
    handle-swipe-right: current-page > 0;  
    handle-swipe-left: current-page < 5;  
    swiped => {  
      if self.current-position.x > self.pressed-position.x + self.width  
        current-page -= 1;  
      } else if self.current-position.x < self.pressed-position.x - self.width  
        current-page += 1;  
    }  
  }  
  
  HorizontalLayout {  
    property <length> position: - current-page * root.width;  
    animate position { duration: 200ms; easing: ease-in-out; }  
    property <length> swipe-offset;  
    x: position + swipe-offset;  
    states [  
      swiping when sgr.swiping : {  
    }  
  }  
}
```

```

        swipe-offset: sgr.current-position.x - sgr.pressed-position.x
    out { animate swipe-offset { duration: 200ms; easing: ease-in-out; } }
]

Rectangle { width: root.width; background: green; }
Rectangle { width: root.width; background: limegreen; }
Rectangle { width: root.width; background: yellow; }
Rectangle { width: root.width; background: orange; }
Rectangle { width: root.width; background: red; }
Rectangle { width: root.width; background: violet; }
}
}
}

```

Specify the different swipe directions you'd like to handle by setting the `handle-swipe-left/right/up/down` properties and react to the gesture in the `swiped` callback.

Pointer press events on the recognizer's area are forwarded to the children with a small delay. If the pointer moves by more than 8 logical pixels in one of the enabled swipe directions, the gesture is recognized, and events are no longer forwarded to the children.

Properties

enabled

`bool` default: true

When disabled, the `SwipeGestureHandler` doesn't recognize any gestures.

pressed-position

`struct Point` (out) default: a struct with all default values

The position of the pointer when the swipe started.

current-position

struct Point (out) default: a struct with all default values

The current pointer position.

swiping

bool (out) default: false

true while the gesture is recognized, false otherwise.

Handle swipe directions properties

handle-swipe-left

bool default: false

handle-swipe-right

bool default: false

handle-swipe-up

bool default: false

handle-swipe-down

bool default: false

Callbacks

- **moved()** : Invoked when the pointer is moved.
- **swiped()** : Invoked after the swipe gesture was recognized and the pointer was released.
- **cancelled()** : Invoked when the swipe is cancelled programmatically or if the window loses focus.

Functions

- **cancel()** : Cancel any on-going swipe gesture recognition.

Previous

Flickable

Next

TouchArea

© 2025 SixtyFPS GmbH

TouchArea

Use `TouchArea` to control what happens when the region it covers is touched or interacted with using the mouse.

When not part of a layout, its width or height default to 100% of the parent element.

```
export component Example inherits Window {  
    width: 200px;  
    height: 100px;  
    area := TouchArea {  
        width: parent.width;  
        height: parent.height;  
        clicked => {  
            rect2.background = #ff0;  
        }  
    }  
    Rectangle {  
        x:0;  
        width: parent.width / 2;  
        height: parent.height;  
        background: area.pressed ? blue: red;  
    }  
    rect2 := Rectangle {  
        x: parent.width / 2;  
        width: parent.width / 2;  
        height: parent.height;  
    }  
}
```

slint

Properties

enabled

`bool` default: true

When disabled, the `TouchArea` doesn't recognize any touch or mouse events and they are passed through to elements underneath.

```
import { Button, CheckBox } from "std-widgets.slint";    slint

export component Example inherits Window {
    width: 200px; height: 100px;

    VerticalLayout {
        Rectangle {
            Button {
                text: "Try to press me";
            }
            TouchArea {
                enabled: event-blocker.checked;
            }
        }
        event-blocker := CheckBox {
            text: "Block Access";
        }
    }
}
```

Note

When `enabled` is set to false while the `TouchArea` is pressed, `pointer-event` will be invoked with `PointerEventKind.Cancel`, and the `pressed` and `has-hover` properties will be reset to `false`.

has-hover

`bool` (out) default: false

Set to true when the mouse is over the `TouchArea` area.

mouse-cursor

`enum MouseCursor` default: the first enum value

The mouse cursor type when the mouse is hovering the `TouchArea`.

mouse-x

`length` (out) default: 0px

Set by the `TouchArea` to the position of the mouse within it.

mouse-y

`length` (out) default: 0px

Set by the `TouchArea` to the position of the mouse within it.

pressed-x

`length` (out) default: 0px

Set by the `TouchArea` to the position of the mouse at the moment it was last pressed.

pressed-y

`length` (out) default: 0px

Set by the `TouchArea` to the position of the mouse at the moment it was last pressed.

pressed

`bool` (out) default: false

Set to true by the `TouchArea` when the mouse is pressed over it.

Callbacks

`clicked()`

Invoked when clicked: A finger or the left mouse button is pressed, then released on this element.

`double-clicked()`

Invoked when double-clicked. The left mouse button is pressed and released twice on this element in a short period of time, or the same is done with a finger. The `clicked()` callbacks will be triggered before the `double-clicked()` callback is triggered.

`moved()`

The mouse or finger has been moved. This will only be called if the mouse is also pressed or the finger continues to touch the display. See also **pointer-event(PointerEvent)**.

pointer-event(PointerEvent)

PointerEvent

Represents a Pointer event sent by the windowing system. This structure is passed to the `pointer-event` callback of the `TouchArea` element.

- **button** (*PointerEventButton*): The button that was pressed or released
- **kind** (*PointerEventKind*): The kind of the event
- **modifiers** (*KeyboardModifiers*): The keyboard modifiers pressed during the event

scroll-event(PointerScrollEvent) -> EventResult

Invoked when the mouse wheel was rotated or another scroll gesture was made. The `PointerScrollEvent` argument contains information about how much to scroll in what direction.

PointerScrollEvent

Represents a Pointer scroll (or wheel) event sent by the windowing system. This structure is passed to the `scroll-event` callback of the `TouchArea` element.

- **delta_x** (*length*): The amount of pixel in the horizontal direction
- **delta_y** (*length*): The amount of pixel in the vertical direction
- **modifiers** (*KeyboardModifiers*): The keyboard modifiers pressed during the event

The returned `EventResult` indicates whether to accept or ignore the event. Ignored events are forwarded to the parent element.

EventResult

This enum describes whether an event was rejected or accepted by an event handler.

- **reject** : The event is rejected by this event handler and may then be handled by the parent item
- **accept** : The event is accepted and won't be processed further

Previous

[SwipeGestureHandler](#)

Next

[Overview](#)

© 2025 SixtyFPS GmbH

Key Handling Overview

To handle keyboard input in Slint, use the `FocusScope` or individual `key-pressed` and `key-released` callbacks in various elements. Keyboard input is delivered via [KeyEvent](#) data structures. The primary field of this data structure is the `text` property, which holds all affected keys encoded in a string. Use the [Key namespace](#) to identify known named keys.

KeyEvent

This structure is generated and passed to the key press and release callbacks of the `FocusScope` element.

- `text (string)`: The unicode representation of the key pressed.
- `modifiers (KeyboardModifiers)`: The keyboard modifiers active at the time of the key press event.
- `repeat (bool)`: This field is set to true for key press events that are repeated, i.e. the key is held down. It's always false for key release events.

Key Namespace

Use the constants in the `Key` namespace to handle pressing of keys that don't have a printable character.

- `Backspace`
- `Tab`
- `Return`
- `Escape`

- **Backtab**
- **Delete**
- **Shift**
- **Control**
- **Alt**
- **AltGr**
- **CapsLock**
- **ShiftR**
- **ControlR**
- **Meta**
- **MetaR**
- **Space**
- **UpArrow**
- **DownArrow**
- **LeftArrow**
- **RightArrow**
- **F1**
- **F2**
- **F3**
- **F4**
- **F5**
- **F6**
- **F7**
- **F8**
- **F9**
- **F10**
- **F11**
- **F12**
- **F13**
- **F14**

- **F15**
- **F16**
- **F17**
- **F18**
- **F19**
- **F20**
- **F21**
- **F22**
- **F23**
- **F24**
- **Insert**
- **Home**
- **End**
- **PageUp**
- **PageDown**
- **ScrollLock**
- **Pause**
- **SysReq**
- **Stop**
- **Menu**

Previous

TouchArea

Next

FocusScope

FocusScope

```
export component Example inherits Window {  
    width: 100px;  
    height: 100px;  
    forward-focus: my-key-handler;  
    my-key-handler := FocusScope {  
        key-pressed(event) => {  
            debug(event.text);  
            if (event.modifiers.control) {  
                debug("control was pressed during this event");  
            }  
            if (event.text == Key.Escape) {  
                debug("Esc key was pressed")  
            }  
            accept  
        }  
    }  
}
```

slint

The `FocusScope` exposes callbacks to handle key events. Note that `FocusScope` will only invoke them when it `has-focus`.

The [KeyEvent](#) has a `text` property, which is a character of the key entered. When a non-printable key is pressed, the character will be either a control character, or it will be mapped to a private unicode character. The mapping of these non-printable, special characters is available in the [KeyEvent](#) namespace

Key Event Delivery

Key events are delivered to the element that `has-focus`.

Before attempting to deliver the `KeyEvent`, it is checked whether some other element wants to intercept the `KeyEvent`. Visiting all the elements starting at the `Window`, going down toward the focused element, `capture_key_pressed` or `capture_key_released` is called. If any of these returns `EventResult::accept`, then key event processing stops at this point. If `EventResult::reject` is returned, then event delivery continues.

If no element captures the `KeyEvent`, then the `KeyEvent` is delivered to the focused element by calling `key-pressed` or `key-released`. If these callbacks return `EventResult::accept`, then event delivery is finished and the event has been handled. Otherwise, (recursively) try to deliver the key event to the parent element.

Properties

has-focus

`bool` (out) default: false

Is `true` when the element has keyboard focus.

enabled

`bool` default: true

When `false`, the `FocusScope` will not accept focus, neither via click nor via tab focus traversal, not even programmatically.

A parent `FocusScope` will still receive key events from child `FocusScope`s that were rejected, even if `enabled` is set to `false`.

focus-on-click

`bool` default: true

When `true`, the `FocusScope` will make itself the focused element when clicked.

This property has no effect if the `enabled` property is set to false.

focus-on-tab-navigation

`bool` default: true

When true, the `FocusScope` will accept focus as part of the tab focus traversal.

This property has no effect if the `enabled` property is set to false.

Functions

focus()

Call this function to transfer keyboard focus to this `FocusScope`, to receive future [KeyEvent](#)s.

clear-focus()

Call this function to remove keyboard focus from this `FocusScope` if it currently has the focus. See also [FocusHandling](#).

Callbacks

capture-key-pressed(KeyEvent) -> EventResult

This function is called during key event handling, *before* `key-pressed` is called. Use this to intercept key press events. The returned [EventResult](#) indicates whether to accept or reject the event. Rejected events are forwarded to the parent element.

capture-key-released(KeyEvent) -> EventResult

This function is called during key event handling, *before* `key-released` is called. Use this to intercept key release events. The returned [EventResult](#) indicates whether to accept or reject the event. Rejected events are forwarded to the parent element.

key-pressed(KeyEvent) -> EventResult

Invoked when a key is pressed, the argument is a [KeyEvent](#) struct. The returned [EventResult](#) indicates whether to accept or reject the event. Rejected events are forwarded to the parent element.

key-released(KeyEvent) -> EventResult

Invoked when a key is released, the argument is a [KeyEvent](#) struct. The returned [EventResult](#) indicates whether to accept or reject the event. Rejected events are forwarded to the parent element.

focus-changed-event(FocusReason)

Invoked when the focus on the `FocusScope` has changed. The argument is a [FocusReason](#) enum containing the reason for focus change.

focus-gained(FocusReason)

Invoked when the `FocusScope` gains focus. The argument is a [FocusReason](#) enum containing the reason for focus gain.

focus-lost(FocusReason)

Invoked when the `FocusScope` loses focus. The argument is a [FocusReason](#) enum containing the reason for focus loss.

Previous

[Overview](#)

Next

[TextInput](#)

TextInput

The `TextInput` is a lower-level item that shows text and allows entering text. You should probably not use this directly, but instead use the [LineEdit](#) or [TextEdit](#) component.

When not part of a layout, its width and height defaults to 100% of the parent element.

The `TextInput` does not scroll automatically when the cursor is outside of the visible area. This is the responsibility of the enclosing widget to ensure using the `cursor-position-changed` callback.

Example

```
export component Example inherits Window {slint
    width: 270px;
    height: 40px;
    Rectangle {
        clip: true;

        TextInput {
            text: "Edit me";
            width: max(parent.width, self.preferred-width);
            vertical-alignment: center;

            private property <length> margin: 1rem;
            cursor-position-changed(cursor-position) => {
                if cursor-position.x + self.x < margin {
                    self.x = - cursor-position.x + margin;
                } else if cursor-position.x + self.x > parent.width - margin -

```

```
        self.x = parent.width - cursor-position.x - margin - self.  
    }  
}  
}  
}  
}  
}
```

Properties

color

`brush` default: depends on the style

The color of the text.

font-family

`string` default: ""

The name of the font family selected for rendering the text.

font-size

`length` default: 0px

The font size of the text.

font-weight

`int` default: 0

The weight of the font. The values range from 100 (lightest) to 900 (thickest). 400 is the normal weight.

font-italic

`bool` default: false

Whether or not the font face should be drawn italicized or not.

font-metrics

`struct FontMetrics` default: a struct with all default values

The design metrics of the font scaled to the font pixel size used by the element.

has-focus

`bool` (out) default: false

`TextInput` sets this to `true` when it's focused. Only then it receives [KeyEvents](#).

horizontal-alignment

`enum TextHorizontalAlignment` default: the first enum value

The horizontal alignment of the text.

input-type

`enum InputType` default: `text`

Use this to configure `TextInput` for editing special input, such as password fields.

letter-spacing

`length` default: 0

The letter spacing allows changing the spacing between the glyphs. A positive value increases the spacing and a negative value decreases the distance.

page-height

`length` default: 0px

The height of the page used to compute how much to scroll when the user presses page up or page down.

read-only

`bool` default: false

When set to `true`, text editing via keyboard and mouse is disabled but selecting text is still enabled as well as editing text programmatically.

selection-background-color

`color` default: a transparent color

The background color of the selection.

selection-foreground-color

`color` default: a transparent color

The foreground color of the selection.

single-line

`bool` default: `true`

When set to `true`, the text is always rendered as a single line, regardless of new line separators in the text.

text-cursor-width

`length` default: provided at run-time by the selected widget style

The width of the text cursor.

text

`string` default: `""`

The text rendered and editable by the user.

vertical-alignment

`enum TextVerticalAlignment` default: the first enum value

The vertical alignment of the text.

wrap

`enum TextWrap` default: no-wrap

The way the text input wraps. Only makes sense when `singleLine` is false.

Functions

focus()

Call this function to focus the text input and make it receive future keyboard events.

clear-focus()

Call this function to remove keyboard focus from this `TextInput` if it currently has the focus.

See also [FocusHandling](#).

set-selection-offsets(int, int)

Selects the text between two UTF-8 offsets.

select-all()

Selects all text.

clear-selection()

Clears the selection.

copy()

Copies the selected text to the clipboard.

cut()

Copies the selected text to the clipboard and removes it from the editable area.

paste()

Pastes the text content of the clipboard at the cursor position.

Callbacks

accepted()

Invoked when the enter key is pressed.

cursor-position-changed(Point)

The cursor was moved to the new (x, y) position described by the `Point` argument.

edited()

Invoked when the text has changed because the user modified it.

key-pressed(KeyEvent) -> EventResult

Invoked when a key is pressed, the argument is a `KeyEvent` struct. Use this callback to handle keys before `TextInput` does. Return `accept` to indicate that you've handled the event, or return `reject` to let `TextInput` handle it.

key-released(KeyEvent) -> EventResult

Invoked when a key is released, the argument is a `KeyEvent` struct. Use this callback to handle keys before `TextInput` does. Return `accept` to indicate that you've handled the event, or return `reject` to let `TextInput` handle it.

Accessibility

By default, `TextInput` elements have the following accessibility properties set:

- `accessible-role: text-input;`

- `accessible-value: text;`
- `accessible-enabled: enabled;`
- `accessible-read-only: read-only;`

Previous

FocusScope

Next

TextInputInterface

© 2025 SixtyFPS GmbH

TextInputInterface

```
import { LineEdit } from "std-widgets.slint";  
  
component VKB {  
    Rectangle { background: yellow; }  
}  
  
export component Example inherits Window {  
    width: 200px;  
    height: 100px;  
    VerticalLayout {  
        LineEdit {}  
        FocusScope {}  
        if TextInputInterface.text-input-focused: VKB {}  
    }  
}
```

slint

Properties

The `TextInputInterface.text-input-focused` property can be used to find out if a `TextInput` element has the focus. If you're implementing your own virtual keyboard, this property is an indicator whether the virtual keyboard should be shown or hidden.

text-input-focused

`bool` default: false

True if an `TextInput` element has the focus; false otherwise.

Previous

TextInput

Next

Common Properties

© 2025 SixtyFPS GmbH

Common Properties

Properties

These properties are valid on all visible items and can be used to specify constraints when used in layouts:

col, row

`int` default: 0

See [GridLayout](#).

colspan, rowspan

`int` default: 0

See [GridLayout](#).

horizontal-stretch, vertical-stretch

`float` (in-out) default: 0.0

Specify how much relative space these elements are stretching in a layout. When 0, this means that the elements won't be stretched unless all elements are 0. Built-in widgets have a value of either 0 or 1.

max-width, max-height

`length` default: 0px

The maximum size of an element.

min-width, min-height

`length` default: 0px

The minimum size of an element.

preferred-width, preferred-height

`length` default: 0px

The preferred size of an element.

Previous

[TextInputInterface](#)

Next

[GridLayout](#)

© 2025 SixtyFPS GmbH

GridLayout

GridLayout places elements on a grid.

GridLayout covers its entire surface with cells. Cells are not aligned. The elements constituting the cells will be stretched inside their allocated space, unless their size constraints—like, e.g., `min-height` or `max-width`—work against this.

```
// This example uses the `Row` element
export component Foo inherits Window {
    width: 200px;
    height: 200px;
    GridLayout {
        spacing: 5px;
        Row {
            Rectangle { background: red; }
            Rectangle { background: blue; }
        }
        Row {
            Rectangle { background: yellow; }
            Rectangle { background: green; }
        }
    }
}
```

slint

```
// This example uses the `col` and `row` properties
export component Foo inherits Window {
    width: 200px;
    height: 150px;
```

slint

```
GridLayout {  
    Rectangle { background: red; }  
    Rectangle { background: blue; }  
    Rectangle { background: yellow; row: 1; }  
    Rectangle { background: green; }  
    Rectangle { background: black; col: 2; row: 0; }  
}  
}
```

Spacing Properties

spacing

`length` default: 0px

The distance between the elements in the layout. This single value is applied to both horizontal and vertical spacing.

To target specific axis with different values use the following properties:

spacing-horizontal

`length` default: 0px

spacing-vertical

`length` default: 0px

Padding Properties

padding

`length` default: 0px

The padding around the grid structure as a whole. This single value is applied to all sides.

To target specific sides with different values use the following properties:

padding-left

`length` default: 0px

padding-right

`length` default: 0px

padding-top

`length` default: 0px

padding-bottom

`length` default: 0px

Cell elements

Cell elements inside a `GridLayout` obtain the following new properties. Any bindings to these properties must be compile-time constants:

row

`int` default: 0

The index of the element's row within the grid. Setting this property resets the element's column to zero, unless explicitly set.

col

`int` default: 0

The index of the element's column within the grid. Set this property to override the sequential column assignment (e.g., to skip a column).

rowspan

`int` default: 1

The number of rows this element should span.

colspan

`int` default: 1

The number of columns this element should span.

To implicitly sequentially assign row indices—just like with `col`—wrap cell elements in `Row` elements.

The following example creates a 2-by-2 grid with `Row` elements, omitting one cell:

```
import { Button } from "std-widgets.slint";  
export component Foo inherits Window {  
    width: 200px;  
    height: 100px;  
    GridLayout {  
        Row { // children implicitly on row 0  
            Button { col: 1; text: "Top Right"; } // implicit column after thi  
        }  
        Row { // children implicitly on row 1  
            Button { text: "Bottom Left"; } // implicitly in column 0...  
            Button { text: "Bottom Right"; } // ...and 1  
        }  
    }  
}
```

lint

The following example creates the same grid using the `row` property. Row indices must be taken care of manually:

```
import { Button } from "std-widgets.slint";  
export component Foo inherits Window {  
    width: 200px;  
    height: 100px;  
    GridLayout {  
        Button { row: 0; col: 1; text: "Top Right"; } // `row: 0;` could even  
        Button { row: 1; text: "Bottom Left"; } // new row, implicitly resets  
        Button { text: "Bottom Right"; } // same row, sequentially assigned co
```

lint

```
}
```

Previous

Common Properties

Next

HorizontalLayout

© 2025 SixtyFPS GmbH

HorizontalLayout

```
export component Foo inherits Window {  
  width: 200px;  
  height: 100px;  
  HorizontalLayout {  
    spacing: 5px;  
    Rectangle { background: red; width: 10px; }  
    Rectangle { background: blue; min-width: 10px; }  
    Rectangle { background: yellow; horizontal-stretch: 1; }  
    Rectangle { background: green; horizontal-stretch: 2; }  
  }  
}
```

slint

Places its children next to each other horizontally. The size of elements can either be fixed with the `width` or `height` property, or if they aren't set they will be computed by the layout respecting the minimum and maximum sizes and the stretch factor.

Spacing Properties

spacing

`length` default: 0px

The distance between the elements in the layout.

Padding Properties

padding

`length` default: 0px

The padding within the layout as a whole. This single value is applied to all sides.

To target specific sides with different values use the following properties:

padding-left

`length` default: 0px

padding-right

`length` default: 0px

padding-top

`length` default: 0px

padding-bottom

`length` default: 0px

Alignment Properties

alignment

`enum LayoutAlignment` default: the first enum value

Set the alignment. Matches the CSS flex box.

Previous

GridLayout

Next

VerticalLayout

VerticalLayout

```
export component Foo inherits Window {  
    width: 200px;  
    height: 100px;  
    VerticalLayout {  
        spacing: 5px;  
        Rectangle { background: red; width: 10px; }  
        Rectangle { background: blue; min-width: 10px; }  
        Rectangle { background: yellow; vertical-stretch: 1; }  
        Rectangle { background: green; vertical-stretch: 2; }  
    }  
}
```

slint

Places its children next to each other vertically. The size of elements can either be fixed with the `width` or `height` property, or if they aren't set they will be computed by the layout respecting the minimum and maximum sizes and the stretch factor.

Spacing Properties

spacing

`length` default: 0px

The distance between the elements in the layout.

Padding Properties

padding

`length` default: 0px

The padding within the layout as a whole. This single value is applied to all sides.

To target specific sides with different values use the following properties:

padding-left

`length` default: 0px

padding-right

`length` default: 0px

padding-top

`length` default: 0px

padding-bottom

`length` default: 0px

Alignment Properties

alignment

`enum LayoutAlignment` default: the first enum value

Set the alignment. Matches the CSS flex box.

Previous

[HorizontalLayout](#)

Next

[ContextMenuArea](#)

ContextMenuArea

Use the non-visual `ContextMenuArea` element to declare an area where the user can show a context menu.

The context menu is shown if the user right-clicks on the area covered by the `ContextMenuArea` element, or if the user presses the “Menu” key on their keyboard while a `FocusScope` within the `ContextMenuArea` has focus. On Android, the menu is shown with a long press. Call the `show()` function on the `ContextMenuArea` element to programmatically show the context menu.

One of the children of the `ContextMenuArea` must be a `Menu` element, which defines the menu to be shown. There can be at most one `Menu` child, all other children must be of a different type and will be shown as regular visual children. Define the structure of the menu by placing `MenuItem` or `Menu` elements inside that `Menu`.

Function

show(Point)

Call this function to programmatically show the context menu at the given position relative to the `ContextMenuArea` element.

close()

Close the context menu if it’s currently open.

enabled

`bool` default: true

When disabled, the `Menu` is not showing.

Menu

Place the `Menu` element in a [MenuBar](#), a `ContextMenuArea`, or within another `Menu`. Use `MenuItem` children of individual menu items, `Menu` children to create sub-menus, and `MenuSeparator` to create separators.

Properties of Menu

title

`string` default: ""

This is the label of the menu as written in the menu bar or in the parent menu.

enabled

`bool` default: true

When disabled, the `Menu` can be selected but not activated.

icon

`image` default: the empty image

The icon shown next to the title when in a parent menu.

MenuItem

A `MenuItem` represents a single menu entry. It must be a child of a `Menu` element.

Properties of MenuItem

title

`string` default: ""

The title shown for this menu item.

enabled

`bool` default: true

When disabled, the `MenuItem` can be selected but not activated.

checkable

`bool` default: true

When true, the `MenuItem` can be checked. The value of the `checked` property is toggled when the user activates the menu item.

checked

`bool` default: true

When true, a checkmark will be shown next to the title of the `MenuItem`.

icon

`image` default: the empty image

The icon shown next to the title.

Callbacks of `MenuItem`

`activated()`

Invoked when the menu entry is activated.

MenuSeparator

A `MenuSeparator` represents a separator in a menu. It cannot have children, and doesn't have properties or callbacks. `MenuSeparator` at the beginning or end of a menu will not be visible. Consecutive `MenuSeparator`s will be merged into one.

Example

```
export component Example {  
  ContextMenuArea {  
    Menu {  
      MenuItem {  
        title: @tr("Cut");  
        activated => { debug("Cut"); }  
      }  
      MenuItem {  
        title: @tr("Copy");  
        activated => { debug("Copy"); }  
      }  
      MenuItem {  
        title: @tr("Paste");  
        activated => { debug("Paste"); }  
      }  
      MenuSeparator {}  
      Menu {  
        title: @tr("Find");  
        MenuItem {  
          title: @tr("Find Next");  
        }  
        MenuItem {  
          title: @tr("Find Previous");  
        }  
      }  
    }  
  }  
}
```

Previous

Next

VBoxLayout

Dialog

© 2025 SixtyFPS GmbH

Dialog

```
import { StandardButton, Button } from "std-widgets.slint";
export component Example inherits Dialog {
    Text {
        text: "This is a dialog box";
    }
    StandardButton { kind: ok; }
    StandardButton { kind: cancel; }
    Button {
        text: "More Info";
        dialog-button-role: action;
    }
}
```

Dialog is like a window, but it has buttons that are automatically laid out.

A Dialog should have one main element as child, that isn't a button. The dialog can have any number of `StandardButton` widgets or other buttons with the `dialog-button-role` property. The buttons will be placed in an order that depends on the target platform at run-time.

The `kind` property of the `StandardButton`s and the `dialog-button-role` properties need to be set to a constant value, it can't be an arbitrary variable expression. There can't be several `StandardButton`s of the same kind.

A callback `<kind>_clicked` is automatically added for each `StandardButton` which doesn't have an explicit callback handler, so it can be handled from the native code: For example if there is a button of kind `cancel`, a `cancel_clicked` callback will be added. Each of these

automatically-generated callbacks is an alias for the `clicked` callback of the associated `StandardButton`.

Properties

icon

`image` default: the empty image

The window icon shown in the title bar or the task bar on window managers supporting it.

title

`string` default: ""

The window title that is shown in the title bar.

Previous

[ContextMenuArea](#)

Next

[MenuBar](#)

© 2025 SixtyFPS GmbH

MenuBar

Use the `MenuBar` element in a [Window](#) to declare the structure of a menu bar, including the actual menus and sub-menus.

Note

There can only be one `MenuBar` element in a [Window](#) and it must not be in a `for` or a `if`.

The `MenuBar` doesn't have properties, but it must contain [Menu](#) as children that represent top level entries in the menu bar.

Depending on the platform, the menu bar might be native or rendered by Slint. This means that for example, on macOS, the menu bar will be at the top of the screen. The `width` and `height` property of the [Window](#) define the client area, excluding the menu bar. The `x` and `y` properties of `Window` children are also relative to the client area.

Example

```
export component Example inherits Window {slint
  MenuBar {
    Menu {
      title: @tr("File");
      MenuItem {
        title: @tr("New");
        activated => { file-new(); }
      }
      MenuItem {
        title: @tr("Open");
        activated => { file-open(); }
      }
    }
  }
}
```

```
        title: @tr("Open");
        activated => { file-open(); }
    }
}

Menu {
    title: @tr("Edit");
    MenuItem {
        title: @tr("Copy");
    }
    MenuItem {
        title: @tr("Paste");
    }
    MenuSeparator {}
    Menu {
        title: @tr("Find");
        MenuItem {
            title: @tr("Find in document...");
        }
        MenuItem {
            title: @tr("Find Next");
        }
        MenuItem {
            title: @tr("Find Previous");
        }
    }
}

callback file-new();
callback file-open();

// ... actual window content goes here
}
```

Previous

Dialog

Next

PopupWindow

© 2025 SixtyFPS GmbH

PopupWindow

```
export component Example inherits Window {  
  width: 100px;  
  height: 100px;  
  
  popup := PopupWindow {  
    Rectangle { height:100%; width: 100%; background: yellow; }  
    x: 20px; y: 20px; height: 50px; width: 50px;  
  }  
  
  TouchArea {  
    height:100%; width: 100%;  
    clicked => { popup.show(); }  
  }  
}
```

slint

Use this element to show a popup window like a tooltip or a popup menu.

Note: It isn't allowed to access properties of elements within the popup from outside of the `PopupWindow`.

Properties

close-policy

`enum PopupClosePolicy` default: `close-on-click`

By default, a `PopupWindow` closes when the user clicks. Set this to false to prevent that behavior and close it manually using the `close()` function.

Functions

show()

Show the popup on the screen.

close()

Closes the popup. Use this if you set the `close-policy` property to `no-auto-close`.

Previous

MenuBar

Next

Window

© 2025 SixtyFPS GmbH

Window

`Window` is the root of the tree of elements that are visible on the screen.

The `Window` geometry will be restricted by its layout constraints: Setting the `width` will result in a fixed width, and the window manager will respect the `min-width` and `max-width` so the window can't be resized bigger or smaller. The initial width can be controlled with the `preferred-width` property. The same applies to the `Window`'s height.

Use the [MenuBar](#) element to declare a menu bar for the window.

Properties

always-on-top

`bool` default: `false`

Whether the window should be placed above all other windows on window managers supporting it.

full-screen

`bool` default: `true` if '`SLINT_FULLSCREEN`' environment variable is set, otherwise `false`

Whether to display the `Window` in full-screen mode. In full-screen mode the `Window` will occupy the entire screen, it will not be resizable, and it will not display the title bar.

background

`brush` default: depends on the style

The background brush of the `Window`.

default-font-family

`string` default: ""

The font family to use as default in text elements inside this window, that don't have their `font-family` property set.

default-font-size

`length` (in-out) default: 0

The font size to use as default in text elements inside this window, that don't have their `font-size` property set. The value of this property also forms the basis for relative font sizes.

default-font-weight

`int` default: 0

The font weight to use as default in text elements inside this window, that don't have their `font-weight` property set. The values range from 100 (lightest) to 900 (thickest). 400 is the normal weight.

icon

`image` default: the empty image

The window icon shown in the title bar or the task bar on window managers supporting it.

no-frame

`bool` default: false

Whether the window should be borderless/frameless or not.

title

`string` default: ""

The window title that is shown in the title bar.

Previous

[PopupWindow](#)

Next

[Global Structs and Enums](#)

© 2025 SixtyFPS GmbH

Global Structs and Enums

Structs

FontMetrics

`FontMetrics`

A structure to hold metrics of a font for a specified pixel size.

- **ascent** (*length*): The distance between the baseline and the top of the tallest glyph in the font.
- **descent** (*length*): The distance between the baseline and the bottom of the tallest glyph in the font. This is usually negative.
- **x_height** (*length*): The distance between the baseline and the horizontal midpoint of the tallest glyph in the font, or zero if not specified by the font.
- **cap_height** (*length*): The distance between the baseline and the top of a regular upper-case glyph in the font, or zero if not specified by the font.

KeyboardModifiers

`KeyboardModifiers`

The `KeyboardModifiers` struct provides booleans to indicate possible modifier keys on a keyboard, such as Shift, Control, etc. It is provided as part of `KeyEvent`'s `modifiers` field.

Keyboard shortcuts on Apple platforms typically use the Command key (⌘), such as Command+C for "Copy". On other platforms the same shortcut is typically represented using Control+C. To make it easier to develop cross-platform applications, on macOS, Slint maps the Command key to the control modifier, and the Control key to the meta modifier.

On Windows, the Windows key is mapped to the meta modifier.

- **alt** (*bool*): Indicates the Alt key on a keyboard.
- **control** (*bool*): Indicates the Control key on a keyboard, except on macOS, where it is the Command key (⌘).
- **shift** (*bool*): Indicates the Shift key on a keyboard.
- **meta** (*bool*): Indicates the Control key on macOS, and the Windows key on Windows.

KeyEvent

`KeyEvent`

This structure is generated and passed to the key press and release callbacks of the `FocusScope` element.

- **text** (*string*): The unicode representation of the key pressed.
- **modifiers** (*KeyboardModifiers*): The keyboard modifiers active at the time of the key press event.
- **repeat** (*bool*): This field is set to true for key press events that are repeated, i.e. the key is held down. It's always false for key release events.

Point

`Point`

This structure represents a point with x and y coordinate

- **x** (*length*):
- **y** (*length*):

PointerEvent

`PointerEvent`

Represents a Pointer event sent by the windowing system. This structure is passed to the `pointer-event` callback of the `TouchArea` element.

- **button** (*PointerEventButton*): The button that was pressed or released
- **kind** (*PointerEventKind*): The kind of the event
- **modifiers** (*KeyboardModifiers*): The keyboard modifiers pressed during the event

PointerScrollEvent

PointerScrollEvent

Represents a Pointer scroll (or wheel) event sent by the windowing system. This structure is passed to the `scroll-event` callback of the `TouchArea` element.

- **delta_x** (*length*): The amount of pixel in the horizontal direction
- **delta_y** (*length*): The amount of pixel in the vertical direction
- **modifiers** (*KeyboardModifiers*): The keyboard modifiers pressed during the event

StandardListViewItem

StandardItem

Represents an item in a StandardListView and a StandardTableView.

- **text** (*string*): The text content of the item

TableColumn

TableColumn

This is used to define the column and the column header of a TableView

- **title** (*string*): The title of the column header
- **min_width** (*length*): The minimum column width (logical length)
- **horizontal_stretch** (*float*): The horizontal column stretch
- **sort_order** (*SortOrder*): Sorts the column
- **width** (*length*): the actual width of the column (logical length)

Enums

AccessibleRole

AccessibleRole

This enum represents the different values for the `accessible-role` property, used to describe the role of an element in the context of assistive technology such as screen readers.

- **none** : The element isn't accessible.
- **button** : The element is a `Button` or behaves like one.
- **checkbox** : The element is a `CheckBox` or behaves like one.
- **combobox** : The element is a `ComboBox` or behaves like one.
- **groupbox** : The element is a `GroupBox` or behaves like one.
- **image** : The element is an `Image` or behaves like one. This is automatically applied to `Image` elements.
- **list** : The element is a `ListView` or behaves like one.
- **slider** : The element is a `Slider` or behaves like one.
- **spinbox** : The element is a `SpinBox` or behaves like one.
- **tab** : The element is a `Tab` or behaves like one.
- **tab-list** : The element is similar to the tab bar in a `TabWidget`.
- **tab-panel** : The element is a container for tab content.
- **text** : The role for a `Text` element. This is automatically applied to `Text` elements.
- **table** : The role for a `TableView` or behaves like one.
- **tree** : The role for a `TreeView` or behaves like one. (Not provided yet)
- **progress-indicator** : The element is a `ProgressIndicator` or behaves like one.
- **text-input** : The role for widget with editable text such as a `LineEdit` or a `TextEdit`. This is automatically applied to `TextInput` elements.
- **switch** : The element is a `Switch` or behaves like one.
- **list-item** : The element is an item in a `ListView`.

AnimationDirection

AnimationDirection

This enum describes the direction of an animation.

- **normal** : The ["normal" direction as defined in CSS↗](#).
- **reverse** : The ["reverse" direction as defined in CSS↗](#).
- **alternate** : The ["alternate" direction as defined in CSS↗](#).
- **alternate-reverse** : The ["alternate reverse" direction as defined in CSS↗](#).

ColorScheme

ColorScheme

This enum indicates the color scheme used by the widget style. Use this to explicitly switch between dark and light schemes, or choose Unknown to fall back to the system default.

- **unknown** : The scheme is not known and a system wide setting configures this. This could mean that the widgets are shown in a dark or light scheme, but it could also be a custom color scheme.
- **dark** : The style chooses light colors for the background and dark for the foreground.
- **light** : The style chooses dark colors for the background and light for the foreground.

DialogButtonRole

DialogButtonRole

This enum represents the value of the `dialog-button-role` property which can be added to any element within a `Dialog` to put that item in the button row, and its exact position depends on the role and the platform.

- **none** : This isn't a button meant to go into the bottom row
- **accept** : This is the role of the main button to click to accept the dialog. e.g. "Ok" or "Yes"
- **reject** : This is the role of the main button to click to reject the dialog. e.g. "Cancel" or "No"
- **apply** : This is the role of the "Apply" button

- **reset** : This is the role of the “Reset” button
- **help** : This is the role of the “Help” button
- **action** : This is the role of any other button that performs another action.

EventResult

EventResult

This enum describes whether an event was rejected or accepted by an event handler.

- **reject** : The event is rejected by this event handler and may then be handled by the parent item
- **accept** : The event is accepted and won’t be processed further

FillRule

FillRule

This enum describes the different ways of deciding what the inside of a shape described by a path shall be.

- **nonzero** : The [“nonzero” fill rule as defined in SVG↗](#).
- **evenodd** : The [“evenodd” fill rule as defined in SVG↗](#)

FocusReason

FocusReason

This enum describes the different reasons for a FocusEvent

- **programmatic** : A built-in function invocation caused the event (`.focus()` , `.clear-focus()`)
- **tab-navigation** : Keyboard navigation caused the event (tabbing)
- **pointer-click** : A mouse click caused the event
- **popup-activation** : A popup caused the event

- **window-activation** : The window manager changed the active window and caused the event

ImageFit

`ImageFit`

This enum defines how the source image shall fit into an `Image` element.

- **fill** : Scales and stretches the source image to fit the width and height of the `Image` element.
- **contain** : The source image is scaled to fit into the `Image` element's dimension while preserving the aspect ratio.
- **cover** : The source image is scaled to cover into the `Image` element's dimension while preserving the aspect ratio. If the aspect ratio of the source image doesn't match the element's one, then the image will be clipped to fit.
- **preserve** : Preserves the size of the source image in logical pixels. The source image will still be scaled by the scale factor that applies to all elements in the window. Any extra space will be left blank.

ImageHorizontalAlignment

`ImageHorizontalAlignment`

This enum specifies the horizontal alignment of the source image.

- **center** : Aligns the source image at the center of the `Image` element.
- **left** : Aligns the source image at the left of the `Image` element.
- **right** : Aligns the source image at the right of the `Image` element.

ImageRendering

`ImageRendering`

This enum specifies how the source image will be scaled.

- **smooth** : The image is scaled with a linear interpolation algorithm.

- **pixelated** : The image is scaled with the nearest neighbor algorithm.

ImageTiling

`ImageTiling`

This enum specifies how the source image will be tiled.

- **none** : The source image will not be tiled.
- **repeat** : The source image will be repeated to fill the `Image` element.
- **round** : The source image will be repeated and scaled to fill the `Image` element, ensuring an integer number of repetitions.

ImageVerticalAlignment

`ImageVerticalAlignment`

This enum specifies the vertical alignment of the source image.

- **center** : Aligns the source image at the center of the `Image` element.
- **top** : Aligns the source image at the top of the `Image` element.
- **bottom** : Aligns the source image at the bottom of the `Image` element.

InputType

`InputType`

This enum is used to define the type of the input field.

- **text** : The default value. This will render all characters normally
- **password** : This will render all characters with a character that defaults to “*”
- **number** : This will only accept and render number characters (0-9)
- **decimal** : This will accept and render characters if it's valid part of a decimal

LayoutAlignment

LayoutAlignment

Enum representing the `alignment` property of a `HorizontalBox`, a `VerticalBox`, a `HorizontalLayout`, or `VerticalLayout`.

- **stretch** : Use the minimum size of all elements in a layout, distribute remaining space based on `*-stretch` among all elements.
- **center** : Use the preferred size for all elements, distribute remaining space evenly before the first and after the last element.
- **start** : Use the preferred size for all elements, put remaining space after the last element.
- **end** : Use the preferred size for all elements, put remaining space before the first element.
- **space-between** : Use the preferred size for all elements, distribute remaining space evenly between elements.
- **space-around** : Use the preferred size for all elements, distribute remaining space evenly before the first element, after the last element and between elements.

MouseCursor

MouseCursor

This enum represents different types of mouse cursors. It's a subset of the mouse cursors available in CSS. For details and pictograms see the [MDN Documentation for cursor↗](#). Depending on the backend and used OS unidirectional resize cursors may be replaced with bidirectional ones.

- **default** : The systems default cursor.
- **none** : No cursor is displayed.
- **help** : A cursor indicating help information.
- **pointer** : A pointing hand indicating a link.
- **progress** : The program is busy but can still be interacted with.
- **wait** : The program is busy.
- **crosshair** : A crosshair.

- **text** : A cursor indicating selectable text.
- **alias** : An alias or shortcut is being created.
- **copy** : A copy is being created.
- **move** : Something is to be moved.
- **no-drop** : Something can't be dropped here.
- **not-allowed** : An action isn't allowed
- **grab** : Something is grabbable.
- **grabbing** : Something is being grabbed.
- **col-resize** : Indicating that a column is resizable horizontally.
- **row-resize** : Indicating that a row is resizable vertically.
- **n-resize** : Unidirectional resize north.
- **e-resize** : Unidirectional resize east.
- **s-resize** : Unidirectional resize south.
- **w-resize** : Unidirectional resize west.
- **ne-resize** : Unidirectional resize north-east.
- **nw-resize** : Unidirectional resize north-west.
- **se-resize** : Unidirectional resize south-east.
- **sw-resize** : Unidirectional resize south-west.
- **ew-resize** : Bidirectional resize east-west.
- **ns-resize** : Bidirectional resize north-south.
- **nesw-resize** : Bidirectional resize north-east-south-west.
- **nwse-resize** : Bidirectional resize north-west-south-east.

Orientation

Orientation

Represents the orientation of an element or widget such as the `Slider`.

- **horizontal** : Element is oriented horizontally.
- **vertical** : Element is oriented vertically.

PathEvent

PathEvent

PathEvent is a low-level data structure describing the composition of a path. Typically it is generated at compile time from a higher-level description, such as SVG commands.

- **begin** : The beginning of the path.
- **line** : A straight line on the path.
- **quadratic** : A quadratic bezier curve on the path.
- **cubic** : A cubic bezier curve on the path.
- **end-open** : The end of the path that remains open.
- **end-closed** : The end of a path that is closed.

PointerEventButton

PointerEventButton

This enum describes the different types of buttons for a pointer event, typically on a mouse or a pencil.

- **other** : A button that is none of left, right, middle, back or forward. For example, this is used for the task button on a mouse with many buttons.
- **left** : The left button.
- **right** : The right button.
- **middle** : The center button.
- **back** : The back button.
- **forward** : The forward button.

PointerEventKind

PointerEventKind

The enum reports what happened to the `PointerEventButton` in the event

- **cancel** : The action was cancelled.

- **down** : The button was pressed.
- **up** : The button was released.
- **move** : The pointer has moved,

PopupClosePolicy

PopupClosePolicy

- **close-on-click** : Closes the `PopupWindow` when user clicks or presses the escape key.
- **close-on-click-outside** : Closes the `PopupWindow` when user clicks outside of the popup or presses the escape key.
- **no-auto-close** : Does not close the `PopupWindow` automatically when user clicks.

ScrollBarPolicy

ScrollBarPolicy

This enum describes the scrollbar visibility

- **as-needed** : Scrollbar will be visible only when needed
- **always-off** : Scrollbar never shown
- **always-on** : Scrollbar always visible

SortOrder

SortOrder

This enum represents the different values of the `sort-order` property. It's used to sort a `StandardTableView` by a column.

- **unsorted** : The column is unsorted.
- **ascending** : The column is sorted in ascending order.
- **descending** : The column is sorted in descending order.

StandardButtonKind

StandardButtonKind

Use this enum to add standard buttons to a `Dialog`. The look and positioning of these `StandardButton`s depends on the environment (OS, UI environment, etc.) the application runs in.

- `ok` : A "OK" button that accepts a `Dialog`, closing it when clicked.
- `cancel` : A "Cancel" button that rejects a `Dialog`, closing it when clicked.
- `apply` : A "Apply" button that should accept values from a `Dialog` without closing it.
- `close` : A "Close" button, which should close a `Dialog` without looking at values.
- `reset` : A "Reset" button, which should reset the `Dialog` to its initial state.
- `help` : A "Help" button, which should bring up context related documentation when clicked.
- `yes` : A "Yes" button, used to confirm an action.
- `no` : A "No" button, used to deny an action.
- `abort` : A "Abort" button, used to abort an action.
- `retry` : A "Retry" button, used to retry a failed action.
- `ignore` : A "Ignore" button, used to ignore a failed action.

TextHorizontalAlignment

TextHorizontalAlignment

This enum describes the different types of alignment of text along the horizontal axis of a `Text` element.

- `left` : The text will be aligned with the left edge of the containing box.
- `center` : The text will be horizontally centered within the containing box.
- `right` : The text will be aligned to the right of the containing box.

TextOverflow

TextOverflow

This enum describes the how the text appear if it is too wide to fit in the `Text` width.

- **clip** : The text will simply be clipped.
- **elide** : The text will be elided with

TextStrokeStyle

TextStrokeStyle

This enum describes the positioning of a text stroke relative to the border of the glyphs in a `Text`.

- **outside** : The inside edge of the stroke is at the outer edge of the text.
- **center** : The center line of the stroke is at the outer edge of the text, like in Adobe Illustrator.

TextVerticalAlignment

TextVerticalAlignment

This enum describes the different types of alignment of text along the vertical axis of a `Text` element.

- **top** : The text will be aligned to the top of the containing box.
- **center** : The text will be vertically centered within the containing box.
- **bottom** : The text will be aligned to the bottom of the containing box.

TextWrap

TextWrap

This enum describes the how the text wrap if it is too wide to fit in the `Text` width.

- **no-wrap** : The text won't wrap, but instead will overflow.
- **word-wrap** : The text will be wrapped at word boundaries if possible, or at any location for very long words.
- **char-wrap** : The text will be wrapped at any character. Currently only supported by the Qt and Software renderers.

Previous

Window

Next

Math

© 2025 SixtyFPS GmbH

Math

The **Math** namespace contains functions that are available both in the global scope and in the `Math` namespace.

```
// Using the Math namespace
x: Math.abs(-10); // sets x to 10
```

slint

```
// Using the functions via global scope. No need for 'Math' prefix.
x: abs(-10); // sets x to 10
```

slint

Some of the math functions can be used in a postfix style which can make the code more readable.

```
// Using the postfix style.
x: (-10).abs(); // sets x to 10
```

slint

T type Many of the math functions can be used with any [numeric type](#) such as `angle`, `duration`, `float`, `int`, `length`, and `percent`. These are represented on this page as `T`.

General Math Functions

`abs(T) -> T`

Return the absolute value, where `T` is a numeric type.

```
Math.abs(-10); // returns 10
abs(-10px); // returns 10px
```

slint

```
(-10).abs(); // returns 10
```

ceil(float) -> int

Returns the value rounded up to the nearest integer.

```
Math.ceil(4); // returns 4  
Math.ceil(2.3); // returns 3  
Math.ceil(-1.5); // returns -1
```

slint

floor(float) -> int

Returns the value rounded down to the nearest integer.

```
Math.floor(4); // returns 4  
Math.floor(2.3); // returns 2  
Math.floor(-1.5); // returns -2
```

slint

round(float) -> int

Return the value rounded to the nearest integer

```
Math.round(4.5); // returns 5  
Math.round(4.4); // returns 4  
Math.round(-1.2); // returns -1
```

slint

clamp(T, T, T) -> T

Takes a `value` , `minimum` and `maximum` and returns `maximum` if `value > maximum` , `minimum` if `value < minimum` , or `value` in all other cases.

log(float, float) -> float

Return the log of the first value with a base of the second value

ln(float) -> float

Return the natural log of the value. Same as $\log(e, x)$

min(T, T) -> T

max(T, T) -> T

Return the arguments with the minimum (or maximum) value. All arguments must be of the same numeric type.

```
Math.min(1, 2); // returns 1
Math.min(2, 1); // returns 1
Math.max(1, 2); // returns 2
Math.max(2, 1); // returns 2
```

slint

mod(T, T) -> T

Perform a modulo operation, where T is some numeric type. Returns the remainder of the euclidean division of the arguments. This always returns a positive number between 0 and the absolute value of the second value.

sqrt(float) -> float

Square root

pow(float, float) -> float

Return the value of the first value raised to the second

exp(float, float) -> float

Return the value of the e raised to the x

Trigonometric Functions

acos(float) -> angle

Returns the arccosine, or inverse cosine, of a number. The arccosine is the angle whose cosine is number.

asin(float) -> angle

Returns the arcsine, or inverse sine, of a number. The arcsine is the angle whose sine is number.

atan(float) -> angle

Returns the arctangent, or inverse tangent, of a number.

atan2(float, float) -> angle

cos(angle) -> float

sin(angle) -> float

tan(angle) -> float

The trigonometry function. Note that the should be typed with `deg` or `rad` unit (for example `cos(90deg)` or `sin(slider.value * 1deg)`).

Previous

Global Structs and Enums

Next

animation-tick() / debug()

Builtin Functions

animation-tick() -> duration

This function returns a monotonically increasing time, which can be used for animations.

Calling this function from a binding will constantly re-evaluate the binding. It can be used like

```
so: x: 1000px + sin(animation-tick() / 1s * 360deg) * 100px; or y: 20px *  
mod(animation-tick(), 2s) / 2s
```

```
export component Example inherits Window {  
    preferred-width: 100px;  
    preferred-height: 100px;  
  
    Rectangle {  
        y: 0;  
        background: red;  
        height: 50px;  
        width: parent.width * mod(animation-tick(), 2s) / 2s;  
    }  
  
    Rectangle {  
        background: blue;  
        height: 50px;  
        y: 50px;  
        width: parent.width * abs(sin(360deg * animation-tick() / 3s));  
    }  
}
```

slint

debug(...)

The debug function can take one or multiple values as arguments, prints them, and returns nothing.

Previous

Math

Next

Platform Namespace

© 2025 SixtyFPS GmbH

Platform

The **Platform** namespace contains properties that help deal with platform specific differences.

Properties

os

`enum OperatingSystemType` default: the first enum value

This property holds the type of the operating system detected at run-time.

Note

When running in a web browser, the value of this property is computed at run-time by querying the web browser's navigator properties.

Note

When Slint is ported to new operating systems in the future, this table will be extended.

style-name

`string` default: ""

The name of the currently selected [widget style](#). Some widget styles have dark and light variant suffixes, such as `fluent-light`. This property contains the style name without the suffix. Use [Palette](#)'s `color-scheme` to determine the currently used scheme.

Previous

animation-tick() / debug()

Next

Overview

© 2025 SixtyFPS GmbH

Overview

```
import { Palette, HorizontalBox } from "std-widgets.slint";slint

export component MyCustomWidget {
    in property <string> text <=> label.text;

    Rectangle {
        background: Palette.control-background;

        HorizontalBox {
            label := Text {
                color: Palette.control-foreground;
            }
        }
    }
}
```

Slint provides a series of built-in widgets that can be imported from "std-widgets.slint".

The widget appearance depends on the selected style. See [Selecting a Widget Style](#) for details how to select the style. If no style is selected, `native` is the default. If `native` isn't available, `fluent` is the default.

All widgets support all [properties common to builtin elements](#).

Palette Properties

Use `Palette` to create custom widgets that match the colors of the selected style e.g. fluent, cupertino, material, or qt.

background

`brush` (out) default: a transparent brush

Defines the default background brush. Use this if none of the more specialized background brushes apply.

foreground

`brush` (out) default: a transparent brush

Defines the foreground brush that is used for content that is displayed on `background` brush.

alternate-background

`brush` (out) default: a transparent brush

Defines an alternate background brush that is used for example for text input controls or panels like a side bar.

alternate-foreground

`brush` (out) default: a transparent brush

Defines the foreground brush that is used for content that is displayed on `alternate-background` brush.

control-background

`brush` (out) default: a transparent brush

Defines the default background brush for controls, such as push buttons, combo boxes, etc.

control-foreground

`brush` (out) default: a transparent brush

Defines the foreground brush that is used for content that is displayed on `control-background` brush.

accent-background

`brush` (out) default: a transparent brush

Defines the background brush for highlighted controls such as primary buttons.

accent-foreground

`brush` (out) default: a transparent brush

Defines the foreground brush that is used for content that is displayed on `accent-background` brush.

selection-background

`brush` (out) default: a transparent brush

Defines the background brush that is used to highlight a selection such as a text selection.

selection-foreground

`brush` (out) default: a transparent brush

Defines the foreground brush that is used for content that is displayed on `selection-background` brush.

border

`brush` (out) default: a transparent brush

Defines the brush that is used for borders such as separators and widget borders.

color-scheme

`enum ColorScheme` (in-out) default: the first enum value

Read this property to determine the color scheme used by the palette. Set this property to force a dark or light color scheme. All styles except for the Qt style support setting a dark or

light color scheme.

StyleMetrics Properties

Use `StyleMetrics` to create custom widgets that match the layout settings of the selected style e.g. fluent, cupertino, material, or qt.

layout-spacing

`length` (out) default: 0px

Defines the default layout spacing. This spacing is also used by `VerticalBox`, `HorizontalBox` and `GridBox`.

layout-padding

`length` (out) default: 0px

Defines the default layout padding. This padding is also used by `VerticalBox`, `HorizontalBox` and `GridBox`.

Previous

[Platform Namespace](#)

Next

[Widget Styles](#)

© 2025 SixtyFPS GmbH

Widget Styles

You can modify the look of these widgets by choosing a style.

The styles available include:

Style	Light	Dark	Description
Name	Variant	Variant	
fluent	fluent-light	fluent-dark	These variants belong to the Fluent style, which is based on the Fluent Design System↗ .
material	material-light	material-dark	These variants are part of the Material style, which follows the Material Design↗ .
cupertino	cupertino-light	cupertino-dark	The Cupertino variants emulate the style used by macOS.
cosmic	cosmic-light	cosmic-dark	The Cosmic variants emulate the style used by Cosmic Desktop↗ .
qt			The Qt style uses Qt↗ to render widgets. This style requires Qt to be installed on your system.
native			This is an alias to one of the other styles depending on the platform. It is <code>cupertino</code> on macOS, <code>fluent</code> on Windows, <code>material</code> on Android, <code>qt</code> on linux if Qt is available, or <code>fluent</code> otherwise.

By default, the styles automatically adapt to the system's dark or light color setting. Select a `-light` or `-dark` variant to override the system setting and always show either dark or light colors.

The widget style is determined at your project's compile time. The method to select a style depends on how you use Slint.

If no style is selected, `native` is the default.

Rust

C++

NodeJS

Python

You can select the style before starting your compilation by setting the `SLINT_STYLE` environment variable to the name of your chosen style.

When using the `slint_build` API, call the [`slint_build::compile_with_config\(\)`](#) function.

When using the `slint_interpreter` API, call the

[`slint_interpreter::ComponentCompiler::set_style\(\)`](#) function.

Using Style Properties In Your Own Components

The global `Palette` and `StyleMetrics` properties can be accessed and will be set to the appropriate values of the current style.

```
import { Palette, StyleMetrics } from "std-widgets.slint";slint

export component Example inherits Window {
    Rectangle {
        border-radius: StyleMetrics.layout-padding;
        border-width: 2px;
        border-color: Palette.border;
        background: Palette.background;
    }
}
```

In situations where the specific property is not available you can detect the style via `Platform.style-name`.

```
import { Palette } from "std-widgets.slint";  
  
export component Example inherits Window {  
    Rectangle {  
        border-radius: Platform.style-name == "fluent" ? 4px : 2px;  
        border-width: Platform.style-name == "fluent" ? 2px : 1px;  
        border-color: Palette.border;  
        background: Palette.background;  
    }  
}
```

slint

Previewing Designs With `slint-viewer`

Select the style either by setting the `SLINT_STYLE` environment variable, or by passing the style name with the `--style` argument:

```
slint-viewer --style material /path/to/design.slint
```

Previewing Designs With The Slint Visual Studio Code Extension

To select the style, first open the Visual Studio Code settings editor:

Windows macOS Linux

File > Preferences > Settings

Then enter the style name in Extensions > Slint > Preview:Style

Previewing Designs With The Generic LSP Process

Choose the style by setting the `SLINT_STYLE` environment variable before launching the process. Alternatively, if your IDE integration allows for command line parameters, you can

specify the style using `--style` .

Previous

Overview

Next

Button

© 2025 SixtyFPS GmbH

Button

```
import { Button, VerticalBox } from "std-widgets.slint";
export component Example inherits Window {
    width: 200px;
    height: 100px;
    VerticalBox {
        label := Text {
            text: "Button not clicked";
        }
        Button {
            text: "Click Me";
            clicked => {
                label.text = " Button clicked";
            }
        }
    }
}
```

slint

A simple button. Common types of buttons can also be created with [StandardButton](#).

Properties

checkable

`bool` default: false

Shows whether the button can be checked or not. This enables the `checked` property to possibly become true.

```
Button {  
    text: "Checkable Button";  
    checkable: true;  
}
```

slint

checked

`bool` (in-out) default: false

Shows whether the button is checked or not. Needs `checkable` to be true to work.

enabled

`bool` default: true

Defaults to true. When false, the button cannot be pressed.

has-focus

`bool` (out) default: false

Set to true when the button has keyboard focus

icon

`image` default: the empty image

The image to show in the button. Note that not all styles support drawing icons.

pressed

`bool` (out) default: false

Set to true when the button is pressed.

text

`string` default: ""

The text written in the button.

```
Button {  
    text: "Button with text";  
}
```

slint

primary

`bool` default: false

If set to true the button is displayed with the primary accent color.

colorize-icon

`bool` default: false

If set to true, the icon will be colorized to the same color as the Button's text color.

Callbacks

clicked()

Invoked when clicked: A finger or the left mouse button is pressed, then released on this element.

```
Button {  
    text: "Click me";  
    clicked() => {  
        debug("Button clicked");  
    }  
}
```

slint

Previous

Widget Styles

Next

CheckBox

© 2025 SixtyFPS GmbH

CheckBox

```
import { CheckBox } from "std-widgets.slint";
export component Example inherits Window {
    width: 200px;
    height: 25px;
    background: transparent;
    CheckBox {
        x: 5px;
        width: parent.width;
        height: parent.height;
        text: "Hello World";
    }
}
```

slint

Use a `CheckBox` to let the user select or deselect values, for example in a list with multiple options. Consider using a `Switch` element instead if the action resembles more something that's turned on or off.

Properties

checked

`bool` (in-out) default: false

Whether the checkbox is checked or not.

```
CheckBox {
    text: self.checked ? "Checked" : "Not checked";
```

slint

```
    checked: true;  
}
```

enabled

`bool` default: true

Defaults to true. When false, the checkbox can't be pressed.

has-focus

`bool` (out) default: false

Set to true when the checkbox has keyboard focus.

text

`string` default: ""

The text written next to the checkbox.

```
CheckBox {  
    text: "CheckBox with text";  
}
```

slint

Callbacks

toggled()

The checkbox value changed

```
CheckBox {  
    text: "CheckBox";  
    toggled() => {  
        debug("CheckBox checked: ", self.checked);  
    }  
}
```

slint

Previous

Button

Next

ComboBox

© 2025 SixtyFPS GmbH

ComboBox

```
import { ComboBox } from "std-widgets.slint";
export component Example inherits Window {
    width: 100px;
    height: 100px;
    background: transparent;
    ComboBox {
        x: 5px; y: 5px;
        width: 100px;
        model: ["first", "second", "third"];
        current-value: "first";
    }
}
```

slint

A button that, when clicked, opens a popup to select a value.

Properties

current-index

`int` (in-out) default: -1

The index of the selected value (-1 if no value is selected)

```
ComboBox {
    model: ["first", "second", "third"];
    current-index: 1;
}
```

slint

current-value

`string` (in-out) default: ""

The currently selected text

enabled

`bool` default: true

Defaults to true. When false, the combobox can't be interacted with

has-focus

`bool` (out) default: false

Set to true when the combobox has keyboard focus.

model

`[string]` default: []

The list of possible values

```
ComboBox {  
  model: ["first", "second", "third"];  
}
```

slint

Callbacks

selected(string)

A value was selected from the combo box by the user. The argument is the currently selected value.

```
ComboBox {  
  model: ["first", "second", "third"];  
  selected(value) => {  
    debug("Selected value: ", value);  
  }  
}
```

slint

```
}
```

Previous

CheckBox

Next

ProgressIndicator

© 2025 SixtyFPS GmbH

ProgressIndicator

```
import { ProgressIndicator } from "std-widgets.slint";  
export component Example inherits Window {  
    width: 200px;  
    height: 25px;  
    background: transparent;  
    ProgressIndicator {  
        width: 90%;  
        height: parent.height;  
        progress: 80%;  
    }  
}
```

The `ProgressIndicator` informs the user about the status of an on-going operation, such as loading data from the network.

Properties

indeterminate

`bool` default: false

Set to true if the progress of the operation cannot be determined by value.

progress

`float` default: 0

Percentage of completion, as value between 0 and 1. Values less than 0 or greater than 1 are capped.

```
ProgressIndicator {  
    progress: 0.5;  
}
```

slint

Previous
ComboBox

Next
Slider

© 2025 SixtyFPS GmbH

Slider

```
import { Slider, VerticalBox } from "std-widgets.slint";
export component Example inherits Window {
    width: 200px;
    height: 40px;

    VerticalBox {
        alignment: center;

        Slider {
            value: 42;
        }
    }
}
```

Properties

enabled

`bool` default: true

You can't interact with the slider if enabled is false.

has-focus

`bool` (out) default: false

Set to true when the slider currently has the focus

value

`float` (in-out) default: 0

The value. Defaults to the minimum.

```
Slider {  
    value: 50;  
}
```

slint

step

`float` default: 1

The change step when pressing arrow key.

```
Slider {  
    step: 1;  
}
```

slint

minimum

`float` default: 0

The minimum value.

```
Slider {  
    minimum: 10;  
    value: 11;  
}
```

slint

maximum

`float` default: 100

The maximum value.

```
Slider {
```

slint

```
    maximum: 10;  
    value: 9;  
}
```

orientation

```
enum Orientation default: horizontal
```

If set to true the Slider is displayed vertical.

Callbacks

changed(float)

The value was changed

```
Slider {  
  changed(value) => {  
    debug("New value: ", value);  
  }  
}
```

slint

released(float)

Invoked when the user completed changing the slider's value, i.e. when the press on the knob was released or the arrow keys lifted.

```
Slider {  
  released(position) => {  
    debug("Released at position: ", position);  
  }  
}
```

slint

Previous

Next

ProgressIndicator

SpinBox

© 2025 SixtyFPS GmbH

SpinBox

```
import { SpinBox, VerticalBox } from "std-widgets.slint";
export component Example inherits Window {
    width: 200px;
    height: 50px;

    VerticalBox {
        alignment: center;

        SpinBox {
            value: 42;
        }
    }
}
```

Properties

enabled

`bool` default: true

You can't interact with the spinbox if enabled is false.

has-focus

`bool` (out) default: false

Set to true when the spinbox currently has the focus.

value

`int` (in-out) default: 0

The value. Defaults to the minimum.

```
SpinBox {  
    value: 50;  
}
```

slint

minimum

`int` default: 0

The minimum value.

```
SpinBox {  
    minimum: 10;  
    value: 11;  
}
```

slint

maximum

`int` default: 100

The maximum value.

```
SpinBox {  
    maximum: 10;  
    value: 9;  
}
```

slint

step-size

`int` default: 1

The size that is used on increment or decrement of `value`.

horizontal-alignment

```
enum TextHorizontalAlignment default: left
```

The horizontal alignment of the text.

Callbacks

edited(int)

Emitted when the value has changed because the user modified it

```
SpinBox {  
    edited(value) => {  
        debug("New value: ", value);  
    }  
}
```

slint

Previous

Slider

Next

Spinner

Spinner

```
import { Spinner } from "std-widgets.slint";  
export component Example inherits Window {  
    width: 80px;  
    height: 80px;  
    Spinner {  
        progress: 50%;  
    }  
}
```

slint

The `Spinner` informs the user about the status of an on-going operation, such as loading data from the network. It provides the same properties as [ProgressIndicator](#) but differs in shape.

Properties

indeterminate

`bool` default: `false`

Set to true if the progress of the operation cannot be determined by value.

progress

`float` default: `0.0`

Percentage of completion, as value between 0 and 1. Values less than 0 or greater than 1 are capped.

```
Spinner {  
    progress: 0.5;  
}
```

slint

Previous

SpinBox

Next

StandardButton

© 2025 SixtyFPS GmbH

StandardButton

```
import { StandardButton, VerticalBox } from "std-widgets.sli"
export component Example inherits Window {
    VerticalBox {
        StandardButton { kind: ok; }
        StandardButton { kind: apply; }
        StandardButton { kind: cancel; }
    }
}
```

The StandardButton looks like a button, but instead of customizing with `text` and `icon`, it can used one of the pre-defined `kind` and the text and icon will depend on the style.

Properties

enabled

`bool` default: false

Defaults to true. When false, the button can't be pressed

has-focus

`bool` (out) default: false

Set to true when the button currently has the focus

kind

```
enum StandardButtonKind default: the first enum value
```

The kind of button, one of `ok` `cancel`, `apply`, `close`, `reset`, `help`, `yes`, `no`, `abort`, `retry` or `ignore`

```
StandardButton {  
    kind: ok;  
}
```

slint

primary

```
bool default: false
```

If set to true the button is displayed with the primary accent color.

pressed

```
bool (out) default: false
```

Set to true when the button is pressed.

Callbacks

clicked()

Invoked when clicked: A finger or the left mouse button is pressed, then released on this element.

```
StandardButton {  
    clicked() => {  
        debug("Button clicked");  
    }  
}
```

slint

Previous

Spinner

Next

Switch

© 2025 SixtyFPS GmbH

Switch

```
import { Switch } from "std-widgets.slint";
export component Example inherits Window {
    width: 200px;
    height: 40px;

    Switch {
        text: "Hello World";
    }
}
```

slint

A `Switch` is a representation of a physical switch that allows users to turn things on or off. Consider using a `CheckBox` instead if you want the user to select or deselect values, for example in a list with multiple options.

Properties

checked

`bool` (in-out) default: false

Whether the switch is checked or not.

```
Switch {
    text: self.checked ? "Checked" : "Not checked";
    checked: true;
}
```

slint

enabled

`bool` default: false

When false, the switch can't be pressed

has-focus

`bool` (out) default: false

Set to true when the switch has keyboard focus

text

`string` default: ""

The text written next to the switch.

```
Switch {  
  text: "Switch with text";  
}
```

lint

Callbacks

toggled()

The switch value changed

```
Switch {  
  text: "Switch";  
  toggled() => {  
    debug("CheckBox checked: ", self.checked);  
  }  
}
```

lint

Previous

StandardButton

Next

LineEdit

© 2025 SixtyFPS GmbH

LineEdit

```
import { LineEdit, VerticalBox } from "std-widgets.slint";
export component Example inherits Window {
    width: 200px;
    height: 60px;

    VerticalBox {
        LineEdit {
            placeholder-text: "Enter text here";
        }
    }
}
```

A widget used to enter a single line of text. See [TextEdit](#) for a widget able to handle several lines of text.

Properties

enabled

`bool` default: true

When false, nothing can be entered.

font-size

`length` default: 0px

The size of the font of the input text

has-focus

`bool` (out) default: false

Set to true when the line edit currently has the focus

horizontal-alignment

`enum TextHorizontalAlignment` default: left

The horizontal alignment of the text.

input-type

`enum InputType` default: text

The way to allow special input viewing properties such as password fields.

```
LineEdit {  
    input-type: password;  
}
```

slint

placeholder-text

`string` default: ""

A placeholder text being shown when there is no text in the edit field

read-only

`bool` default: false

When set to true, text editing via keyboard and mouse is disabled but selecting text is still enabled as well as editing text programmatically.

text

`string` (in-out) default: ""

The text being edited

```
LineEdit {  
    text: "Initial text";  
}
```

slint

Functions

focus()

Call this function to focus the `LineEdit` and make it receive future keyboard events.

clear-focus()

Call this function to remove keyboard focus from this `LineEdit` if it currently has the focus.

See also [FocusHandling](#).

set-selection-offsets(int, int)

Selects the text between two UTF-8 offsets.

select-all()

Selects all text.

clear-selection()

Clears the selection.

copy()

Copies the selected text to the clipboard.

cut()

Copies the selected text to the clipboard and removes it from the editable area.

paste()

Pastes the text content of the clipboard at the cursor position.

Callbacks

accepted(string)

Invoked when the enter key is pressed.

```
LineEdit {  
    accepted(text) => {  
        debug("Accepted: ", text);  
    }  
}
```

lint

edited(string)

Emitted when the text has changed because the user modified it

```
LineEdit {  
    edited(text) => {  
        debug("Text edited: ", text);  
    }  
}
```

lint

key-pressed(KeyEvent) -> EventResult

Invoked when a key is pressed, the argument is a [KeyEvent](#) struct. Use this callback to handle keys before `LineEdit` does. Return `accept` to indicate that you've handled the event, or return `reject` to let `LineEdit` handle it.

key-released(KeyEvent) -> EventResult

Invoked when a key is released, the argument is a [KeyEvent](#) struct. Use this callback to handle keys before `LineEdit` does. Return `accept` to indicate that you've handled the event, or return `reject` to let `LineEdit` handle it.

Previous

Switch

Next

ListView

© 2025 SixtyFPS GmbH

ListView

A ListView is like a Scrollview but it should have a `for` element, and the content are automatically laid out in a list. Elements are only instantiated if they are visible

```
import { ListView, VerticalBox } from "std-widgets.slint";
export component Example inherits Window {
    width: 150px;
    height: 150px;

    VerticalBox {
        ListView {
            for data in [
                { text: "Blue", color: "#0000ff", bg: "#eeeeee" },
                { text: "Red", color: "#ff0000", bg: "#eeeeee" },
                { text: "Green", color: "#00ff00", bg: "#eeeeee" },
                { text: "Yellow", color: "#ffff00", bg: "#222222" },
                { text: "Black", color: "#000000", bg: "#eeeeee" },
                { text: "White", color: "#ffffff", bg: "#222222" },
                { text: "Magenta", color: "#ff00ff", bg: "#eeee" },
                { text: "Cyan", color: "#00ffff", bg: "#222222" }
            ] : Rectangle {
                height: 30px;
                background: data.bg;
                width: parent.width;
                Text {
                    x: 0;
                    text: data.text;
                    color: data.color;
                }
            }
        }
    }
}
```

```
        }
    }
}
}
```

Properties

Same as [ScrollView](#).

Callbacks

Same as [ScrollView](#).

Previous

[LineEdit](#)

Next

[ScrollView](#)

© 2025 SixtyFPS GmbH

ScrollView

```
import { ScrollView } from "std-widgets.slint";
export component Example inherits Window {
    width: 200px;
    height: 200px;
    ScrollView {
        width: 200px;
        height: 200px;
        viewport-width: 300px;
        viewport-height: 300px;
        Rectangle { width: 30px; height: 30px; x: 275px; y: 5 };
        Rectangle { width: 30px; height: 30px; x: 175px; y: 25 };
        Rectangle { width: 30px; height: 30px; x: 25px; y: 275 };
        Rectangle { width: 30px; height: 30px; x: 98px; y: 5 };
    }
}
```

slint

A Scrollview contains a viewport that is bigger than the view and can be scrolled. It has scrollbar to interact with. The viewport-width and viewport-height are calculated automatically to create a scrollable view except for when using a for loop to populate the elements. In that case the viewport-width and viewport-height aren't calculated automatically and must be set manually for scrolling to work. The ability to automatically calculate the viewport-width and viewport-height when using for loops may be added in the future and is tracked in issue #407.

Properties

enabled

`bool` default: true

Used to render the frame as disabled or enabled, but doesn't change behavior of the widget.

mouse-drag-pan-enabled

`bool` (in) default: true for Material style, false for all others

When true, the view can be scrolled by dragging with the mouse.

has-focus

`bool` (in-out) default: false

Used to render the frame as focused or unfocused, but doesn't change the behavior of the widget.

viewport-width

`length` (in-out) default: 0px

The width of the viewport of the scrollview.

viewport-height

`length` (in-out) default: 0px

The height of the viewport of the scrollview.

viewport-x

`length` (in-out) default: 0px

The x position of the scrollview relative to the viewport. This is usually a negative value.

viewport-y

`length` (in-out) default: 0px

The y position of the scrollview relative to the viewport. This is usually a negative value.

visible-width

`length` (out) default: 0px

The width of the visible area of the ScrollView (not including the scrollbar)

visible-height

`length` (out) default: 0px

The height of the visible area of the ScrollView (not including the scrollbar)

vertical-scrollbar-policy

`enum ScrollBarPolicy` default: the first enum value

The vertical scroll bar visibility policy. The default value is `ScrollBarPolicy.as-needed`.

horizontal-scrollbar-policy

`enum ScrollBarPolicy` default: the first enum value

The horizontal scroll bar visibility policy. The default value is `ScrollBarPolicy.as-needed`.

Callbacks

scrolled()

Invoked when `viewport-x` or `viewport-y` is changed by a user action (dragging, scrolling).

```
ScrollView {  
    width: 200px;  
    height: 200px;  
    viewport-width: 300px;  
    viewport-height: 300px;  
    Rectangle { width: 30px; height: 30px; x: 275px; y: 50px; background: blue }  
    Rectangle { width: 30px; height: 30px; x: 175px; y: 130px; background: red }  
    Rectangle { width: 30px; height: 30px; x: 25px; y: 210px; background: yellow }  
    Rectangle { width: 30px; height: 30px; x: 98px; y: 55px; background: orange }
```

slint

```
scrolled() => {
    debug("viewport-x: ", self.viewport-x);
    debug("viewport-y: ", self.viewport-y);
}
}
```

Previous

[ListView](#)

Next

[StandardListView](#)

© 2025 SixtyFPS GmbH

StandardListView

```
import { StandardListView, VerticalBox } from "std-widgets.s
export component Example inherits Window {
    width: 200px;
    height: 200px;

    VerticalBox {
        StandardListView {
            model: [ { text: "Blue" }, { text: "Red" }, { tex
                { text: "Yellow" }, { text: "Black" }, { text
                { text: "Magenta" }, { text: "Cyan" },
            ];
        }
    }
}
```

Like ListView, but with a default delegate, and a `model` property.

Properties

Same as [ListView](#), and in addition:

current-item

`int` (in-out) default: 0

The index of the currently active item. -1 mean none is selected, which is the default

model

```
struct StandardListViewItem default: a struct with all default values
```

The model.

```
StandardListView {  
  model: [{ text: "Blue" }, { text: "Red" }, { text: "Green" }];  
}
```

slint

Functions

set-current-item(int)

Sets the current item by the specified index and brings it into view.

Callbacks

current-item-changed(int)

Emitted when the current item has changed because the user modified it

```
StandardListView {  
  model: [{ text: "Blue" }, { text: "Red" }, { text: "Green" }];  
  current-item-changed(index) => {  
    debug("Current item: ", index);  
  }  
}
```

slint

item-pointer-event(int, PointerEvent, Point)

Emitted on any mouse pointer event similar to `TouchArea`. Arguments are item index associated with the event, the `PointerEvent` itself and the mouse position within the listview.

Previous

ScrollView

Next

StandardTableView

© 2025 SixtyFPS GmbH

StandardTableView

The `StandardTableView` represents a table of data with columns and rows. Cells are organized in a model where each row is a model of

```
struct StandardListViewItem default: a struct with all default values
```

```
import { StandardTableView } from "std-widgets.slint"; slint
export component Example inherits Window {
    width: 230px;
    height: 200px;

    StandardTableView {
        width: 230px;
        height: 200px;
        columns: [
            { title: "Header 1" },
            { title: "Header 2" },
        ];
        rows: [
            [
                { text: "Item 1" }, { text: "Item 2" },
            ],
            [
                { text: "Item 1" }, { text: "Item 2" },
            ],
            [
                { text: "Item 1" }, { text: "Item 2" },
            ],
        ]
    }
}
```

```
    ];
}
}
```

Properties

Same as [ListView](#), and in addition:

current-sort-column

`[int] (out) default: 0`

Indicates the sorted column. -1 mean no column is sorted.

columns

`[[struct]] (in-out) default: a struct with all default values`

Defines the model of the table columns.

```
StandardTableView {
  columns: [{ title: "Header 1" }, { title: "Header 2" }];
  rows: [[{ text: "Item 1" }, { text: "Item 2" }]];
}
```

slint

rows

`[[struct]] (in-out) default: a struct with all default values`

Defines the model of table rows.

```
StandardTableView {
  columns: [{ title: "Header 1" }, { title: "Header 2" }];
  rows: [[{ text: "Item 1" }, { text: "Item 2" }]];
}
```

slint

current-row

`int` (in-out) default: 0

The index of the currently active row. -1 mean none is selected, which is the default.

Callbacks

sort-ascending(int)

Emitted if the model should be sorted by the given column in ascending order.

sort-descending(int)

Emitted if the model should be sorted by the given column in descending order.

row-pointer-event(int, PointerEvent, Point)

Emitted on any mouse pointer event similar to `TouchArea`. Arguments are row index associated with the event, the `PointerEvent` itself and the mouse position within the tableview.

current-row-changed(int)

Emitted when the current row has changed because the user modified it

```
StandardTableView {  
  columns: [{ title: "Header 1" }, { title: "Header 2" }];  
  rows: [[{ text: "Item 1" }, { text: "Item 2" }]];  
  
  current-row-changed(index) => {  
    debug("Current row: ", index);  
  }  
}
```

slint

Functions

set-current-row(int)

Sets the current row by index and brings it into view.

Previous

[StandardListView](#)

Next

[TabWidget](#)

© 2025 SixtyFPS GmbH

TabWidget

```
import { TabWidget } from "std-widgets.slint";
export component Example inherits Window {
    width: 200px;
    height: 200px;
    TabWidget {
        Tab {
            title: "First";
            Rectangle { background: orange; }
        }
        Tab {
            title: "Second";
            Rectangle { background: pink; }
        }
    }
}
```

slint

`TabIndex` is a container for a set of tabs. It can only have `Tab` elements as children and only one tab will be visible at a time.

Properties

current-index

`int` default: 0

The index of the currently visible tab.

```
TabWidget {  
    current-index: 1;  
  
    Tab {  
        title: "First";  
    }  
    Tab {  
        title: "Second";  
    }  
}
```

slint

Properties of the Tab element

title

`string` default: ""

The text written on the tab.

```
TabWidget {  
    Tab {  
        title: "First";  
    }  
}
```

slint

Previous

[StandardTableView](#)

Next

[TextEdit](#)

TextEdit

```
import { TextEdit, VerticalBox } from "std-widgets.slint";
export component Example inherits Window {
    width: 200px;
    height: 200px;

    VerticalBox {
        TextEdit {
            font-size: 14px;
            text: "Lorem ipsum dolor sit amet,\nconsectetur
        }
    }
}
```

Similar to [LineEdit](#), but can be used to enter several lines of text

Properties

font-size

`length` default: 0px

The size of the font of the input text.

text

`string` (in-out) default: ""

The text being edited

```
TextEdit {  
    text: "Initial text";  
}
```

slint

has-focus

`bool` (in-out) default: false

Set to true when the widget currently has the focus.

enabled

`bool` default: true

When false, nothing can be entered.

read-only

`bool` default: false

When set to true, text editing via keyboard and mouse is disabled but selecting text is still enabled as well as editing text programmatically.

wrap

`enum TextWrap` default: the first enum value

The way the text wraps (default: word-wrap).

horizontal-alignment

`enum TextHorizontalAlignment` default: the first enum value

The horizontal alignment of the text.

- **placeholder-text** : *(in string)*: A placeholder text being shown when there is no text in the edit field.

Functions

- **focus()** Call this function to focus the `TextEdit` and make it receive future keyboard events.
- **clear-focus()** Call this function to remove keyboard focus from this `TextEdit` if it currently has the focus. See also [focus handling](#).
- **set-selection-offsets(int, int)** Selects the text between two UTF-8 offsets.
- **select-all()** Selects all text.
- **clear-selection()** Clears the selection.
- **copy()** Copies the selected text to the clipboard.
- **cut()** Copies the selected text to the clipboard and removes it from the editable area.
- **paste()** Pastes the text content of the clipboard at the cursor position.

Callbacks

edited(string)

Emitted when the text has changed because the user modified it

```
TextEdit {
    edited(text) => {
        debug("Edited: ", text);
    }
}
```

slint

key-pressed(KeyEvent) -> EventResult

Invoked when a key is pressed, the argument is a `KeyEvent` struct. Use this callback to handle keys before `TextEdit` does. Return `accept` to indicate that you've handled the event, or return `reject` to let `TextEdit` handle it.

key-released(KeyEvent) -> EventResult

Invoked when a key is released, the argument is a [KeyEvent](#) struct. Use this callback to handle keys before `TextEdit` does. Return `accept` to indicate that you've handled the event, or return `reject` to let `TextEdit` handle it.

Previous

TabWidget

Next

GridBox

© 2025 SixtyFPS GmbH

GridBox

A GridBox

Previous

TextEdit

Next

GroupBox

© 2025 SixtyFPS GmbH

GroupBox

```
import { GroupBox , VerticalBox, CheckBox } from "std-widget"  
export component Example inherits Window {  
    width: 200px;  
    height: 100px;  
    GroupBox {  
        title: "Groceries";  
        VerticalLayout {  
            CheckBox { text: "Bread"; checked: true ;}  
            CheckBox { text: "Fruits"; }  
        }  
    }  
}
```

A `GroupBox` is a container that groups its children together under a common title.

Properties

content-padding

`length` default: Depends on the style

The padding within the layout of the content as a whole. This single value is applied to all sides.

enabled

`bool` default: true

When false, the groupbox can't be interacted with

title

`string` default: ""

A text written as the title of the group box.

Previous

GridBox

Next

HorizontalBox

© 2025 SixtyFPS GmbH

HorizontalBox

A `HorizontalBox`

Previous

GroupBox

Next

VerticalBox

© 2025 SixtyFPS GmbH

VerticalBox

A `VerticalBox`

Previous

[HorizontalBox](#)

Next

[AboutSlint](#)

© 2025 SixtyFPS GmbH

AboutSlint

```
import { AboutSlint } from "std-widgets.slint";
export component Example inherits Window {
    height: 175px;
    AboutSlint {}
}
```

slint

This element displays a “Made with Slint” badge.

Previous

[VerticalBox](#)

Next

[DatePickerPopup](#)

© 2025 SixtyFPS GmbH

DatePickerPopup

Use a date picker to let the user select a date.

```
import { DatePickerPopup, Button } from "std-widgets.slint";
export component Example inherits Window {
    width: 400px;
    height: 600px;

    date-picker-button := Button {
        text: @tr("Open Date Picker");

        clicked => {
            date-picker.show();
        }
    }

    date-picker := DatePickerPopup {
        x: (root.width - self.width) / 2;
        y: (root.height - self.height) / 2;
        close-policy: PopupClosePolicy.no-auto-close;

        accepted(date) => {
            date-picker.close();
        }
        canceled => {
            date-picker.close();
        }
    }
}
```

```
}
```

Properties

title

```
string default: ""
```

The text that is displayed at the top of the picker.

date

```
struct Date default: a struct with all default values
```

Set the initial displayed date.

```
DatePickerPopup {  
    date: { year: 2024, month: 11 };  
}
```

slint

Callbacks

canceled()

Invoked when the cancel button is clicked.

```
date-picker := DatePickerPopup {  
    canceled() => {  
        date-picker.close();  
    }  
}
```

slint

accepted(Date)

Invoked when the ok button is clicked.

```
date-picker := DatePickerPopup {  
    accepted(date) => {  
        debug("Selected date: ", date);  
        date-picker.close();  
    }  
}
```

slint

Previous

AboutSlint

Next

TimePickerPopup

© 2025 SixtyFPS GmbH

TimePickerPopup

Use the timer picker to select the time, in either 24-hour or 12-hour mode (AM/PM).

```
import { TimePickerPopup, Button } from "std-widgets.slint";
export component Example inherits Window {
    width: 400px;
    height: 600px;

    time-picker-button := Button {
        text: @tr("Open TimePicker");

        clicked => {
            time-picker.show();
        }
    }

    time-picker := TimePickerPopup {
        x: (root.width - self.width) / 2;
        y: (root.height - self.height) / 2;
        width: 360px;
        height: 524px;
        canceled => {
            time-picker.close();
        }

        accepted(time) => {
            debug(time);
            time-picker.close();
        }
    }
}
```

```
        }
    }
}
```

Properties

use-24-hour-format

`bool` default: system default

If set to `true` 24 hours are displayed otherwise it is displayed in AM/PM mode. (default: system default, if cannot be determined then `true`)

title

`string` default: ""

The text that is displayed at the top of the picker.

time

`struct Time` default: a struct with all default values

Set the initial displayed time.

```
TimePickerPopup {
    time: { hour: 12, minute: 24 };
}
```

slint

Callbacks

canceled()

The cancel button was clicked.

```
time-picker := TimePickerPopup {
    canceled() => {
```

slint

```
    time-picker.close();
}
}
```

accepted(Time)

The ok button was clicked.

```
time-picker := TimePickerPopup {
    accepted(time) => {
        debug("Selected time: ", time);
        time-picker.close();
    }
}
```

lint

Previous

[DatePickerPopup](#)

Next

[Introduction](#)

© 2025 SixtyFPS GmbH

404

Page not found. Check the URL or try using the search bar.

© 2025 SixtyFPS GmbH