Abstract

This document has the intention of teaching you how to build a layout that will run on the C15 Hardware User-Interface (**HWUI**).

In order to understand how the system can be used we have to learn a little bit about the internal workings of our software (**playground**).

The **playground** keeps track of what user interface is currently being shown/should be shown, by saving three states: **UIFocus** [Sound, Parameters, Presets, Banks, Setup], **UIMode** [Select, Store, Edit, Info] and **UIDetail** [Init, ButtonA, ButtonB, ButtonC, ButtonD]. This pack of states will also be called **UIState** from here on.

These states can be manipulated freely, that means there is no invalid state enforced by the system. This gives the UI/UX designer the freedom to create their own ux-flows and structure the layouts.

If we find that the **UIDetail** is not sufficient we can think about adding more possible UIDetail values, or adding another slot inside the **UIState.**

Dynamic Layouts were introduced as a alternative to writing the UI in concrete C++ classes. The approach of instantiating **Layouts** is almost the same as with our old system: when a change of the **UIState** happens we will search for an applicable available **Layout** and instantiate that. The process of searching for an applicable Layout takes the **UIState** into account as well as checking **Conditions**.

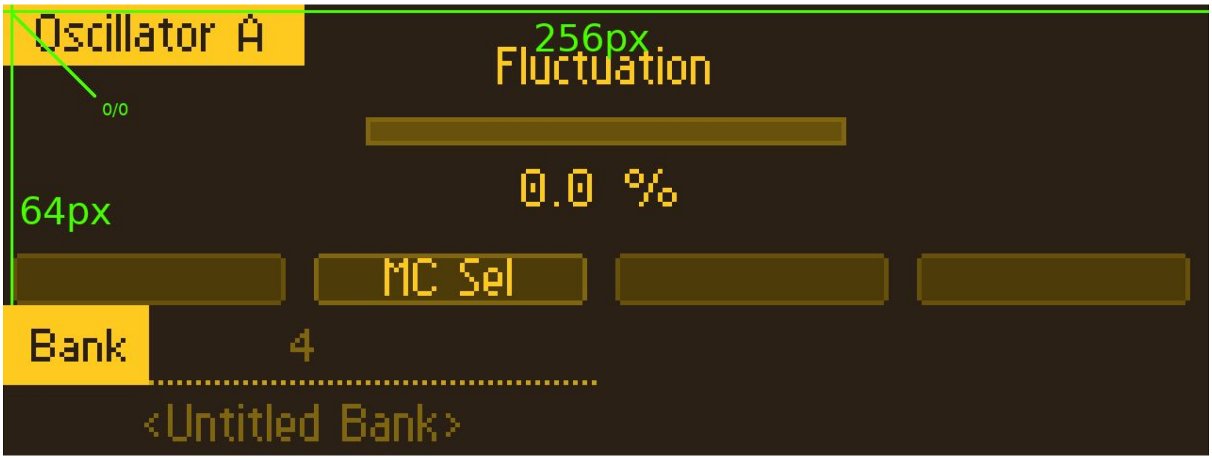
In order to be useful we have to provide (dynamic) data to the UI. Our way of doing that is exposing data via **Event-Sources.** These sources provide as primitive values as possible. For example: “LockedIndication” is either true or false, “ParameterName” provides a string and “ControlPosition” gives a floating-point number.

These Event-Sources can be applied to the building-blocks of UI-Elements, the **Primitives**. **Primitive Classes** are concrete implemented drawables that have to be used to get something to show onscreen. **Primitives** have different properties:
Circle: ControlPosition and **Visibility**
Bar: Range and **Visibility**
Text: Text and **Visibility**

These **primitives** are used to create more complex **controls**, what we call **Control Classes.** They are not only a collection of *named* **Primitives**: type, relative position + size and an optional exposed **Primitive Property** but should also contain a default **Style**.

A *named* **Control Class** will be instantiated inside a Layout. A Layout is in that sense a collection of Control Classes, which in turn consist of Primitives, that might expose properties.

When we instantiate a **Control Class** we hook up **Event-Sources** to **Primitive Properties** of *named* **Primitive Instances** inside that **Control Class** we are instantiating. With that we can display text, numbers, change the visibility of certain elements based on business-logic. When we instantiate a **Control Class** we also *have* to specify a position to instantiate.



screen size and coordinate origin

We are creating UI for the upper part of the screen and have a canvas of 256 pixels in width and 64 pixels in height. The upper left corner is the coordinate 0/0.

But a list of **Control Instances** is not sufficient enough to create a **Layout**. A layout needs criteria that will be used for choosing a layout to instantiate.
The **selector** of a layout specifies the values of **UIState** that have to match in order for the layout to be instantiated. Additionally we introduced **Conditions** that also count towards the **weight** of an layout. The weight is simply the number of specified selectors and conditions. If multiple layouts have all their criteria met the layout with the bigger weight is instantiated.
A **Layout** also contains **Event Sinks:** a mapping of **buttons** to existing **playground-functionality.** This allows the designer to change the **UIState** (using **SwitchToParameterFocus**) or allow interaction with the sound, for example via **IncParam** and **DecParam**.

Using this system of aggregating building-blocks, creation of a Layout is simplified to:
choosing the **Selector** + optional **Conditions**
instantiating **Control Classes** and wiring them up to **Event-Sources**
mapping **Button-Presses** to **Event-Sinks**

Syntax

The system is described in **json** files, these have to be valid and also maintain a specific structure in order for our system to work properly.

Dynamic Layouts Keywords:

```
"layouts": {
     // named json objects of type Layout have to go here
}

"controls": {
     // named json objects of type Control Class have to go here
}

"styles": {
     // named json objects of type Style have to go here
}
```

Inside a named json-Object of type **layout** we expect following structure:

```
  "layout-name": {
   "Selector": {
    ...
   },
   "Conditions": {
    ...
   },
   "Controls": {
    ...
   },
   "EventSinks": {
    ...
   }
  }
```

a **Selector** has the following expected structure

```
"Selector": {
 "UIFocus": "possible value here",
 "UIMode": "possible value here",
 "UIDetail": "possible value here"
}
```

note that not all **UIState** parts have to appear in that list, but at least one has to be present that the layout can be instantiated. Also more **Selectors** mean greater **weight** so keep that in mind when you design the flow and structure of layouts.

Optionally you can (and will have to) add “Conditions” to your layout, simply speaking a list of tests that all have to evaluate to true:

```
"Conditions": [
 "nameOfCondition"
]
```

> The most complete reference of available conditions and their behaviour can be found in the code:
> https://github.com/nonlinear-labs-dev/C15/blob/layouts-release-dual-working-branch/playground/src/proxies/hwui/descriptive-layouts/ConditionRegistry.cpp
>
> or as a path relative to the C15 root:
> playground/src/proxies/hwui/descriptive-layouts/ConditionRegistry.cpp

These **conditions** are optional but will come in handy if the complexity of the system increases. Note: **conditions** count towards the **weight** of the **layout**.

“Controls” contain the instantiation of **Control-Instances** of type **Control-Class**, in order to distinguish different **Instances** we have to name them.

```
"Controls": {
 "NameOfControlInstance": {
  "Class": "ControlClassName",
  "Position": "x/y",
  "Events": "EventSource =>
PrimitiveInstanceName[PrimitiveProperty]"
 }
}
```

analog to the conditions the list of **Event Sources** is continuously growing. And the most complete reference is the source itself: https://github.com/nonlinear-labs-dev/C15/blob/layouts-release-dual-working-branch/playground/src/proxies/hwui/descriptive-layouts/EventSource.cpp

playground/src/proxies/hwui/descriptive-layouts/EventSource.cpp

Events can contain multiple mappings to different **Primitive-Instances** separated by commas. like this:

```
 "Events": "MCModRange => upperBar[Range], MCModRange =>
lowerBar[Range]"
```

Note:
You have to use the correct surrounding tag if you want your json-objects to be read by the system. Your json-objects also have to include some specific tags and have to be named. If you for example put a layout description inside a “controls” tag you will not be able to instantiate that layout.
Placement and consistency is key for the system to work.

Control Instance Example

```
{
"controls": {
 "My-Control": {
  "Left": {
   "Class": "Circle",
   "Tag": "Dark",
   "Rect": "64, -1, 66, 66"
  },
  "Right": {
   "Class": "Circle",
   "Tag": "Light",
   "Rect": "192, -1, 66, 66"
  },
  "Middle": {
   "Class": "Bar",
   "Tag": "Dark",
   "Rect": "64, 0, 128, 64"
  }
 }
},
"styles": {
 "Default-Style-For-My-Control": {
  "selector": {
   "ControlClasses": "My-Control"
  },
  "styles": {
   "Color": "C255"
  }
 }
}
}
```

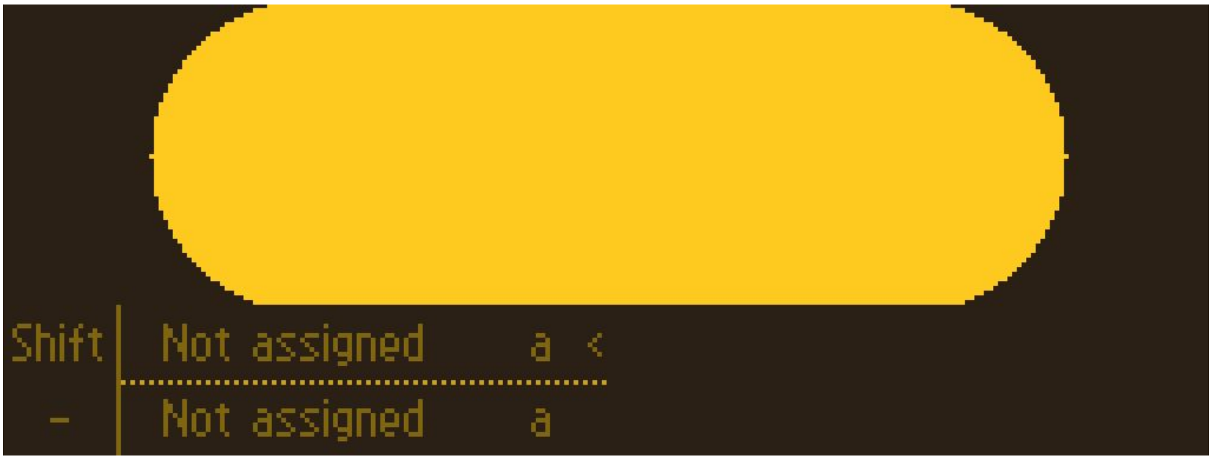
Here we create a **Control-Class** named “My-Control” this control consists of 3 parts: left, middle and right. Two of which are of type *Circle* while the “Middle” is of type *Bar*.
Inside the Primitive-Instance instantiation we can see the **Rect** entity is being used for size and relative position, as well as the optional **Tag** that can be used to apply styling based on *string-wise*-tag-matching.

Below the “control” object we also define the default **style**: paint all **primitives** that are part of the “My-Control”-control using color C255 (see styling below).

We instantiate a **Control Instance** of type “My-Control” inside a **layout** like that:

```
{
 "layouts": {
  "Test-Layout": {
   "Selector": {
    "UIFocus": "Sound"
   },
  "Controls": {
   "FooInstance": {
    "Class": "My-Control",
    "Position": "0,0"
   }
  },
  "EventSinks": {
   "BUTTON_ENTER": "SwitchToParameterFocus"
  }
 }
}
```

The result looks like this:



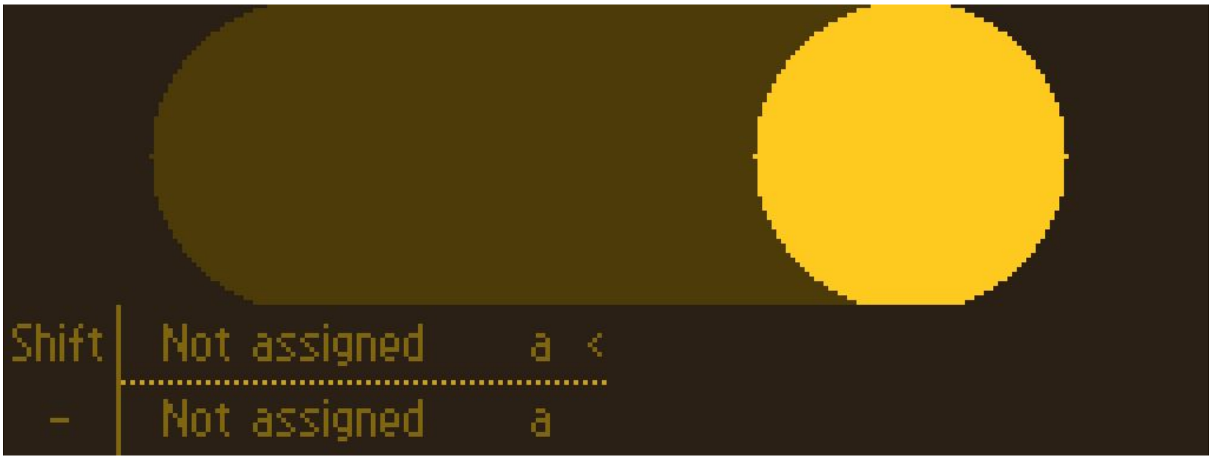
```
Shift | Not assigned     a  <
      ..........................
  -   | Not assigned     a
```

Styling

Because we tagged the **Primitive Instances** we can style the primitives with ease. For that we add more styles to the “My-Control” file:

```
"styles": {
 "LightStyle": {
  "selector": {
   "ControlClasses": "My-Control",
   "Tag": "Light"
  },
  "styles": {
   "Color": "C255"
  }
 },
 "DarkStyle": {
  "selector": {
   "ControlClasses": "My-Control",
   "Tag": "Dark"
  },
  "styles": {
   "Color": "C77"
  }
 }
}
```

Because the weight of the selectors above are greater than the weight of the default style it gets applied instead of the default one.
The styled result looks like this:



Different Primitive-Classes support different style-keys usually the ones that make sense for them. As with all other parts of this system, styling will evolve in the future and a complete reference of styles and according values are usually found in the code itself, but here is a small reference of the currently implemented styles and what primitives support each.

Style-Key	**Possible Values**	supported **Primitive Classes**
Color	C43, C77, C103, C128, C179, C204, C255	Bar, Circle, Border, Text
BorderStyle	Solid, Rounded, None	Border
TextAlignment	Left, Center, Right	Text
BackgroundColor	C43, C77, C103, C128, C179, C204, C255	Text
FontSize	1 - 2,147,483,647	Text

Selectors of **styles** are comparable to **layout-selectors** because you can match styles based on the **UIState** and/or the style specific selectors below.

You can match based on the current instantiated **layout**, where you would have to supply the name of a layout like this: *"LayoutClasses": "YourLayoutNameHere"* If the currently instantiated layout is indeed "YourLayoutNameHere" this part of the selector matches.

Also you have (as described above) the possibility of **tagging primitive-instances** and then you can create styles that match based on the **tag** or tags. *"Tag": "yourTagHere"*

Also you can filter based on the parent (**control class**) of a primitive, that means using the name of an **control class** or **instance**. Use it like "LayoutClasses".
The keywords are those: *"ControlClasses"* for general styling and *"ControlInstances"* for specific instantiated instances.

"PrimitiveClasses" and *"PrimitiveInstances"* match similar to the selector above, but based on **primitives**.

Using these **selectors** in **combination** gives you full control over the style selection process.

Glossary

Name	Description	Notes
Primitive Class	Primitive drawables provided by the developers the implementation of these is not changeable via this interface, these are the only concrete building blocks of this system.	Classes are: Bar, Circle, Border, Text
Primitive Properties	different **Primitive Classes** support different properties that affect e.g. the Color, Visibility, Text or Positioning of **Primitive Classes** when connected with **Event Sources**	extensive list can be found here: https://github.com/nonlinear-labs-dev/C15/wiki/Dynamic-Layouts---Primitives TODO: Explain what effect each property has on the primitive
Primitive Instance	describes a primitive in a **Control Class** by naming the **Primitive Class**, setting a **Rect**, and specifying what **Properties** get exposed to the to be instantiated **Control Instance** **Primitive Instances** are named so that **styling** can be matched based on the name	
Control Class	**Control Classes** are UI-Elements that are build from **Primitive Instances** and corresponding styles. Usually the file containing a **Control Class** also contains a default style for that Control.	
Control Instance	describes a visible **Control** inside a **Layout** by naming the instance, setting a	

	Position and connecting (if applicable) **Event-Sources** to exposed **properties** of that **Control Class**	
Layout	**Layouts** instantiate **Control Classes → Control Instances** contain a **Selector** that matches and instantiates different **Layouts** based on **UIFocus/UIMode/UIDetail** as well as optional **Conditions** also contains optional **Event Sinks** which maps Button-presses to predefined actions (see **Event Sinks)**	
Event Sources	are meant to be consumed by **Primitive Instances** inside a **Layout.** A **Event Sources** are connected to exposed **Properties** of **Primitive Instances** inside the definition of **Control Instances**. The Event Sources are supposed to expose all needed data to populate the User-Interface with all necessary information for the user. Be it *Preset Name*, *Current Bank Number*, *Control-Positions*, *Modulation-States* … etc	
Event Sinks	are used to call **predefined** playground **functionality** on button presses. The connection between Buttons and **Event Sinks** is established inside a **Layout**.	

	The most used **Event Sinks** are the ones that handle **UIFocus/UIMode/UIDetail** changes → enables to switch **Layouts**	
Conditions	are **playground** defined tests that will be executed in the process of selecting a **layout**. If a condition evaluates to *true* the layout can be instantiated (if all other tests also succeed (see **Selectors**))	

Styles → selector
Layouts -> Conditions, Selectors