Modeling the Fontys Living Colours assignment in Dezyne and integrating it on the Arduino platform

To get acquainted with modeling in Dezyne and integration of generated code, one of the first assignments of the Fontys ICT & Technology program was recreated in Dezyne.
The purpose of this document is to quickly go over the requirements for the system and to discuss how specific interesting features were implemented in Dezyne.
It is assumed that the reader is familiar with fundamental concepts of the Dezyne modeling language as introduced in the tutorial that can be found at http://www.verum.com/wp-content/uploads/2016/12/Dezyne_Introductory-Tutorial.pdf.

Requirements

Hardware

To use this system, the following pieces of hardware are necessary:

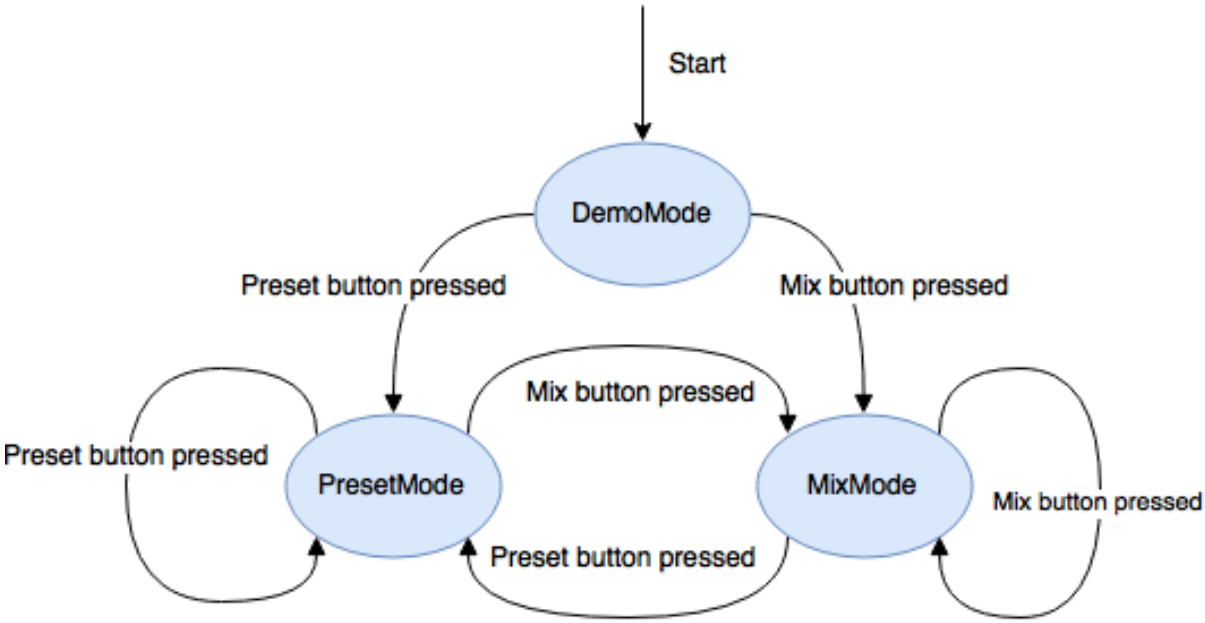
Item	**Quantity**
Arduino Uno	1
RGB LED	1
220 Ohm resistor (for RGB LED)	3
Hardware button	2
Potentiometer	1
Wiring	

Relevant pin definitions for the Arduino connectivity:

Description	**Pin**
Red RGB lead	11
Green RGB lead	6
Blue RGB lead	5
Preset button	12
Mix button	13
Debug pin (optional)	3
Illegal pin (optional)	4
Potmeter	A0

Functionality

The general outline of the program's functionality is illustrated with the following state diagram:



As you can see, the program consists of three modes; it starts in demo-mode, from which it can move into preset-mode or mix-mode at the press of a button. Once it has left the demo-mode, it can no longer return to the demo-mode by design. The program can move freely between preset-mode and mix-mode when the corresponding buttons are pressed. A short overview of the functionality of the modes:

Demo-mode

On startup, the application should be in demo-mode. In demo-mode, the RGB LED will cycle through the colours red, green and blue to demonstrate that those colours in the LED are functional. The interval between changing colours is 1 second.

Mix-mode

When the mix-mode button is pressed, two scenarios may occur:

- [System is not in mix-mode] The system immediately goes into mix-mode
- [System is in mix-mode] The mix-mode selects the next colour to be mixed from the colours {red, green, blue}. When the system is mixing blue and a toggle occurs, the system starts mixing red again.

Mixing a colour is done by turning the potmeter. When toggling to a new colour, the previous colour value(s) should remain active in the LED so the user can see what the result of the mix is.

Preset-mode

When the preset-mode button is pressed, two scenarios may occur:

- [System is not in preset-mode] The system immediately goes into the first preset
- [System is in preset-mode] The system switches to the next preset and the corresponding colour is sent to the RGB LED for output. When the system is toggled on the last preset, it resets to the first preset.

The colours in the preset-mode are {yellow, cyan, magenta}

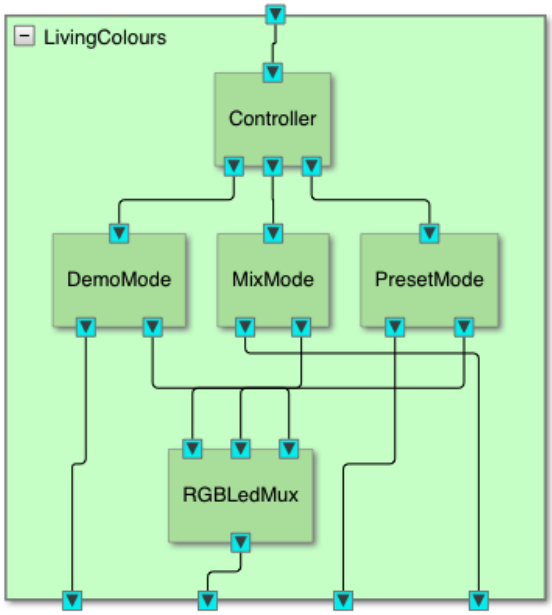
Implementation

The full implementation for this application can be found on https://github.com/Jvaneerd/Verum_Dezyne_Internship/tree/master/Living_Colours_Arduino.

Something important to note is that in this application, there is quite a clear distinction between control logic and data manipulation. The aim of the project was to implement all control logic in Dezyne and integrate the generated code from Dezyne with handwritten code that handles the data manipulation.

Linking your system to handwritten code

Integration between handwritten and generated code is, simply put, done by assigning pointers of handwritten function to their respective bindings in the Dezyne ports from generated code. In order to make this process easier, a single **system** component (LivingColours) was made that contains the DemoMode, PresetMode and MixMode components. This system component then exposes the required ports for these components in an orderly manner. The diagram generated by Dezyne for this system can be seen in the figure to the right.



Some of the ports in the system diagram are already connected to ports on other components, but there are also some ports on the outline of the system. These are ports that have to be bound manually; let's look at an example of this.

By definition, the system is in one of three states- Demo, Mix or Preset. To switch between these states, some user interaction in the form of a button press is required. To handle the switching of states when a button is pressed, the Controller interface (and component- we'll get to that later) was created. It should process three things:

- Process the event that setup was completed and system behavior should start
- Process the event that the mix button was pressed
- Process the event that the preset button was pressed

You may notice a similarity here- the Controller should process events. This is denoted in Dezyne by the '**in**' keyword on its interface. So now, the interface looks like this:

```
interface IController {
        in void begin();
        in void mixButtonPressed();
        in void presetButtonPressed();

        behaviour { … }
}
```

To not distract from the illustrated purpose, the full body of behavior in the interface was left out.
Our LivingColours system is **providing** this port of type IController to the environment that wants to control the Living Colours application- within the scope of this project, this will be the handwritten code.

Interaction between the handwritten code and the generated code from the control logic in Dezyne is done in two directions. **Handwritten → Dezyne** is done by calling a function in handwritten code where the function is implemented in Dezyne generated code. **Dezyne → Handwritten** is done by assigning handwritten functions to events in Dezyne ports.

The interaction for our example interface, IController, would look as follows in handwritten code:

```
// initialization of the LivingColours system
LivingColours lc;
LivingColours_init(&lc, 0);
…
lc.iController->in.begin();
while(forever) {
        …
        // preset button pressed
        lc.iController->in.presetButtonPressed();
        // mix button pressed
        lc.iController->in.mixButtonPressed();
        …
}
```

Depending on the language you are programming in, some syntax may be slightly different, but in general you create an object of your system's type. This object will then contain the ports which you see on the boundaries of the system block. These ports are containers for the **in** and **out** functions. In our example, we only have functions that are going from **Handwritten → Dezyne**, so this translates to function calls of the functions defined in the interface. The implementations for these functions are included in the code generated from the Dezyne models, and as such no further action is needed.

Another scenario is where a Dezyne component wants to communicate with hand-written code. To do this, the component must know what function to call to perform the desired task. For example, the DemoMode needs to be able to start and cancel a timer. This timer is implemented in handwritten code, so in order to let Dezyne take control of this timer, it must be provided with the functions that control the timer.

In this case the LivingColours system **requires** an ITimer interface that looks as follows:

```
interface ITimer {
        in void start();
        in void cancel();

        behaviour { … }
}
```

This time, instead of calling functions in generated code, during initialization you should let the LivingColours system instance know where to find the functions that perform the start and cancel tasks of your timer:

```
lc.iTimer->in.start = tm_StartTimer;
lc.iTimer->in.cancel = tm_ClearTimer;
```

Where tm_StartTimer and tm_ClearTimer are functions that you write yourself that control your (hardware) timer.

The following table might provide some insight into how you integrate handwritten and generated code:

Port type	**Event type**	**User action**
Provides interface (on top of a component)	In	Call Dezyne function from handwritten code
Provides interface	Out	Assign handwritten code to system function
Requires interface (on the bottom of a component)	In	Assign handwritten code to system function
Requires interface	Out	Call Dezyne function from handwritten code

Implementing a timer (working with interrupts)

In addition to being able to start and cancel our timer, we must of course be able to notify our control logic when a timeout occurs. This is a **Handwritten → Dezyne** event, so in the end it will be a function call. To get to the point where the function is called, however, we will need to look a bit deeper into how such a timer could work.

For this application, the decision was made to use a hardware interrupt on the Arduino for our timer. This is handled by an Interrupt Service Routine (ISR), which is a function that is called every time the interrupt flag for our timer is raised. The raising of this flag is done by hardware based on the settings that you use for your timer.

An important feature of an ISR is that time spent within the ISR should be minimized. While inside the ISR, other interrupts of the system are queued. If you spend a lot of time in the ISR, this means that at a certain point the queue might be full and other important interrupts cannot be handled.

How this translates to the targeted behavior in this project is that we cannot simply call the function in our Dezyne system from the ISR. Instead, in the ISR we raise a flag that a timeout has occurred and in the program's main loop we check for this flag. If the flag is raised, we invoke the corresponding function in the Dezyne system and reset the flag. If we were to invoke this function inside the ISR, it is not defined how long it takes for the ISR to end and for normal program behavior to continue.

For some more background on this subject and on ISR etiquette in general, you may check out the following articles:
https://betterembsw.blogspot.com/2013/03/**rules-for-using-interrupts**.html
https://betterembsw.blogspot.com/2014/01/**do-not-re-enable-interrupts**-in-**isr**.html

Another interesting phenomenon with the timer is the possibility of a race condition between two of its states. Say, the timer had been started. After a certain amount of time, it will interrupt and enter our ISR. In the ISR, we raise our flag; so the flag now states that a timeout has occurred.

It takes an arbitrary amount of time for our main loop to reach a state where the timeout flag is handled. If in that amount of time the Dezyne system calls for the timer to be canceled, we find ourselves in a bit of a predicament. The flag states that the timeout event should be fired, but the system expects the timer to be canceled.

This can be solved by erasing the flag contents in hand-written code within the cancel event. This way, when a cancel is called for in between the ISR and flag processing, there will not be a timeout event fired into a Dezyne system that expects the timer to be turned off.

Passing data between control and data domain

As stated before, in this application we can make a rather clear distinction between control logic and data manipulation. However, this does not mean the two are mutually excluded. Data manipulation should not always run, and sometimes control logic requires some data to pass through to other components.

An example of this in the Living Colours project is in the PresetMode component. Switching between different presets is a combination of actions in the control domain and in the data domain.

On the control side of things, the information on what to do when the preset button is pressed is stored. The control logic knows that if the system is in PresetMode it should perform a toggle to the next preset, and what to do otherwise. It does, however, not know what the actual presets are- as of now, that is not relevant to control logic. The data for these presets should also not be stored in the control domain, as data is for the data domain.

When we develop our handwritten components such as writing values to the RGB LED, we try to make sure that components have a single responsibility. So, the LED component is responsible for writing values to the LED and another component is responsible for storing presets. Our control logic is then in charge of passing data from the storage onto the output.

This can be done in Dezyne by declaring a parameter as an **out** parameter:

```
interface IPresetSelector {
	in void toggle();
	in void getPreset(out Colour colour);
}
```

When generated in C, this out parameter translates to a pointer in the function prototype. The control logic initializes an object of type Colour and passes this object to the handwritten function. The handwritten getPreset() function should then set the value of the object this pointer points at.

When this is done, control logic may use this Colour object to pass to another component where it may be processed, for example the LED. In Dezyne, this result is as follows:

```
on iPresetMode.toggle(): {
	[isRunning] {
		iPresetSelector.toggle();
		Colour colour;
		iPresetSelector.getPreset(colour);
		iRGBLed.setColour(colour);
	}
}
```

This will, when triggered to perform a toggle, tell the handwritten PresetSelector to toggle. Then, it initializes a Colour object and passes the object to the handwritten selector where it is filled with values. The filled Colour object is then passed to the RGB LED as input.
In handwritten C, filling the object is as it would be with any other application where you manipulate data based on an output parameter:

```
void ps_GetPreset(IPresetSelector* self, Colour* colour) {
*colour = presets[currentPreset];
}
```

In the above snippet, ‘presets’ is an array of Colour objects where the specific presets are stored. currentPreset is an indicator of what the currently selected preset is, which is changed when the toggle functionality is used.

Defining data types in control domain

In the previous example, we had a look at passing an object of type Colour from one handwritten component to another. This happened through the control domain, where the type of data or its contents does not bear any meaning. All we need to know in the control domain is what kind of interfaces connect the individual components together.

As the interfaces dictate what kind of interaction is possible with the underlying components, we do need to declare some sort of object type with which the functions may be used. In Dezyne, this is done with the **extern** keyword. The Colour data type in our example is defined like this:

```
extern Colour $Colour$;
```

This means that in the external world, an object of the type between $$ exists, and we refer to that type by the alias Colour. Coincidentally, the type and alias are the same here, but it could just as well be 'extern Colour int' to indicate that we use an object with type int but refer to it as Colour.

When generating the code, uses of the Colour object are then replaced with whatever the type we declared it as; in our case, it remains 'Colour'. This does mean that when compiling the generated code the compiler must know the structure of the Colour data type.
As you cannot always manage the order files are compiled in, the following steps were taken to ensure proper compilation:
A file was created called external_types.h which contains all the types we are using in the interfaces for our control model. A compiler flag (--include=external_types.h) was set in the makefile that makes sure that with every source code file, the external_types.h file is included. This way, the compiler always knows what types we are referring to from the generated code without introducing any dependencies to the header file in any Dezyne models.

Multiple outgoing ports to the same device

As you may have noticed in the system view diagram, a component called RGBLedMux was used that is bound to all three modes and has one outgoing port. The reason for this is that we are using 1 handwritten component that interfaces with the LED, but we want to control this same component by all three of our Mode components.
Due to Dezyne's deterministic nature, it is not allowed to bind a port on a system multiple times. To solve this, the Mux component was created. The behavior of this component is to 'collect' function calls from higher-level components and delegate it to the function the higher-level component wants to call. This can then be transmitted over a single outgoing port from the system's point of view. In code, this is how it looks:

```
component RGBLedMux {
        provides IRGBLed iPresetLed;
        provides IRGBLed iMixLed;
        provides IRGBLed iDemoLed;

        requires IRGBLed iPhysicalLed;

        behaviour {
                on iPresetLed.setRed(), iMixLed.setRed(), iDemoLed.setRed(): iPhysicalLed.setRed();
                // rest of the functions in IRGBLed
}
```

So, the Mux component provides an implementation for three interfaces IRGBLed, but all it does is delegate the incoming function calls to a single requires (outgoing) interface. This solution is a pattern to ensure port binding determinism.

Platform challenges

The target for this application was the Arduino platform, which imposes some restrictions:

- Generated code has to be C, as compilers for Arduino do not support C++11 features that are used by Dezyne generated C++ code.
- Compilation of the code must be done with the DZN_TINY compiler flag. This is because with TinyC code generation, certain functions of the dzn runtime that clash with the Arduino platform are omitted.
- Because of the restriction to C code, some functionality from the Arduino library cannot be used. Most notably, this includes the Serial communication class, which makes debugging your program quite a bit more difficult. To make some form of debugging possible, for this project the dzn/runtime.h file was edited to include the Arduino libraries so that some additional LEDs can be used. These LEDs were then used to denote certain breakpoints in generated or handwritten code.