# Introduction

This document describes the current design of the user interface for the ZuluIDE. Figure 1 provides a high level diagram showing the key classes and files used within the solution. The dashed boxes indicate whether the class or file is in the main source tree, the Platform Library (shorted from the full name ZuluIDE\_platform\_RP2040), or the ZuluControl Library. I will use this and other versions of this graphic to explain the behavior through the rest of the document.



Figure 1

## Design Overview

The user interface was designed around the idea of a Model-View-Controller architecture in order to separate user interface concerns from the operation of the device while also facilitating multiple user interfaces to be created with minimal changes to existing components. In the following subsections we look at the model, view, and controller pieces.

### The Model

There are two models: SystemStatus (zuluide::status::SystemStatus) and DisplayState (zuluide::control::DisplayState). Note that the model classes are not shown in Figure 1. The models are used to simply store information and contain no logic to modify system state or to handle interfacing with the user.

SystemStatus is used to store the status of the IDE device. As such, it has properties such as IsPrimary to indicate whether the device is a primary or secondary IDE device. Another example is the IsCardPresent property that indicate whether the SD card is currently inserted.

DisplayState stores information about the state of the user interface as currently expressed in the hardware based UI. This user interface has a status screen showing current information about the device and a menu used for changing settings. DisplayState models the logical state of the user interface, it does not have any implementation that can talk to a display (Such as the SSD1306) or interface the with rotary encoder or user buttons. However, the DisplayState does have a logical flow to the UI. If someone one wanted to create a very different user interface (such as one with only a single screen and everything controlled via buttons, or a CLI-style interface via a serial connection) then a differ model may be necessary for that user interface.

### The View

The purpose of the view is to simply present the current state of the user interface. In our implementation, the class DisplaySSD1306 within the Platform Lib serves as the view. This class is designed around implementation a 128x32 pixel view of the system state using an instance of the SystemStatus and DisplayState classes.

To keep thigs simple, the DisplaySSD1306 class does not focus on being adaptable to other displays, such as one with more pixels. If new display hardware is added, the DisplaySSD1306 could be copied and serve as a starting point.

### The Controller

There are two controllers, one that maintains the current SystemStatus value and one that maintains the current DisplayState.

#### Observable and ObservableSafe

The interfaces Observable and ObservableSafe describe how observers register to receive updates about changes made by controllers. The Observable interface notifies observers using the same core that initiated the update.

ObservableSafe is designed to notify observers using concurrent safe queue. This happens by the implementation of ObservableSafe make a new instance of the value being update and enqueuing the pointer into a queue provided by the observer when registering to receive updates. It is the responsibility of the observer to free the memory used by the copy of the model passed via pointer to the queue.

#### StatusController

StatusController handles updates to the current SystemStatus instance and then notifying observers. The various setters are unremarkable. Of interest are the Begin/EndUpdate() funtions which allow you to have a single notification for several related modifications to the SystemStatus value.

StatusController implements the DeviceControlSafe interface. This interface defines functions that can make modifications to the SystemStatus from a secondary core. This is used by StdDisplayController and its sub-controllers to request changes to the SystemStatus in a concurrent-safe fashion.

#### StdDisplayController

StdDisplayController manages the DisplayState and encodes the logic used to modify the user interface with the status screen and the menu design. Updates for the display occur through the Observable interface because the only changes to the user interface are currently done within the single UI core.

Modifications for each different mode (status, select an image, eject the current image, info, etc) are handled by different controller classes I am calling sub-controllers. There is nothing special about these controllers, the logic is just broken up into different classes for each mode to simplify the house keeping and make the logic easier to follow when in each mode.

# Examples

In this section, we look at how the components interact during a few different operations.

## Setting the Initial Image

When the ZuluIDE is started with an SD card inserted, the first image on the card is selected. Figure 2 provides a visual representation of the ordering of the classes and methods involved in loading the intiail image. During initialization when the the setupStatusController function is called, an observer function (status\_observer) is added to the StatusController. The purpose of this function is to ensure that the device is in the correct state the status controller thinks it should be (with respect to the loaded image.)

At the end of the setupStatusController function the loadFirstImage function takes over. It uses the ImageIterator class to look for valid images. If an image is found, it uses the StatusController to update the SystemState to show that the image is loaded. When this change happens, the StatusController notifies the observer which performs the operations necessary to load the image into the IDE simulation code.



Figure 2

## Removing an SD Card

## Inserting an SD Card

## Pressing the Eject Button

## Selecting an Image

# Adding an I2C Interface