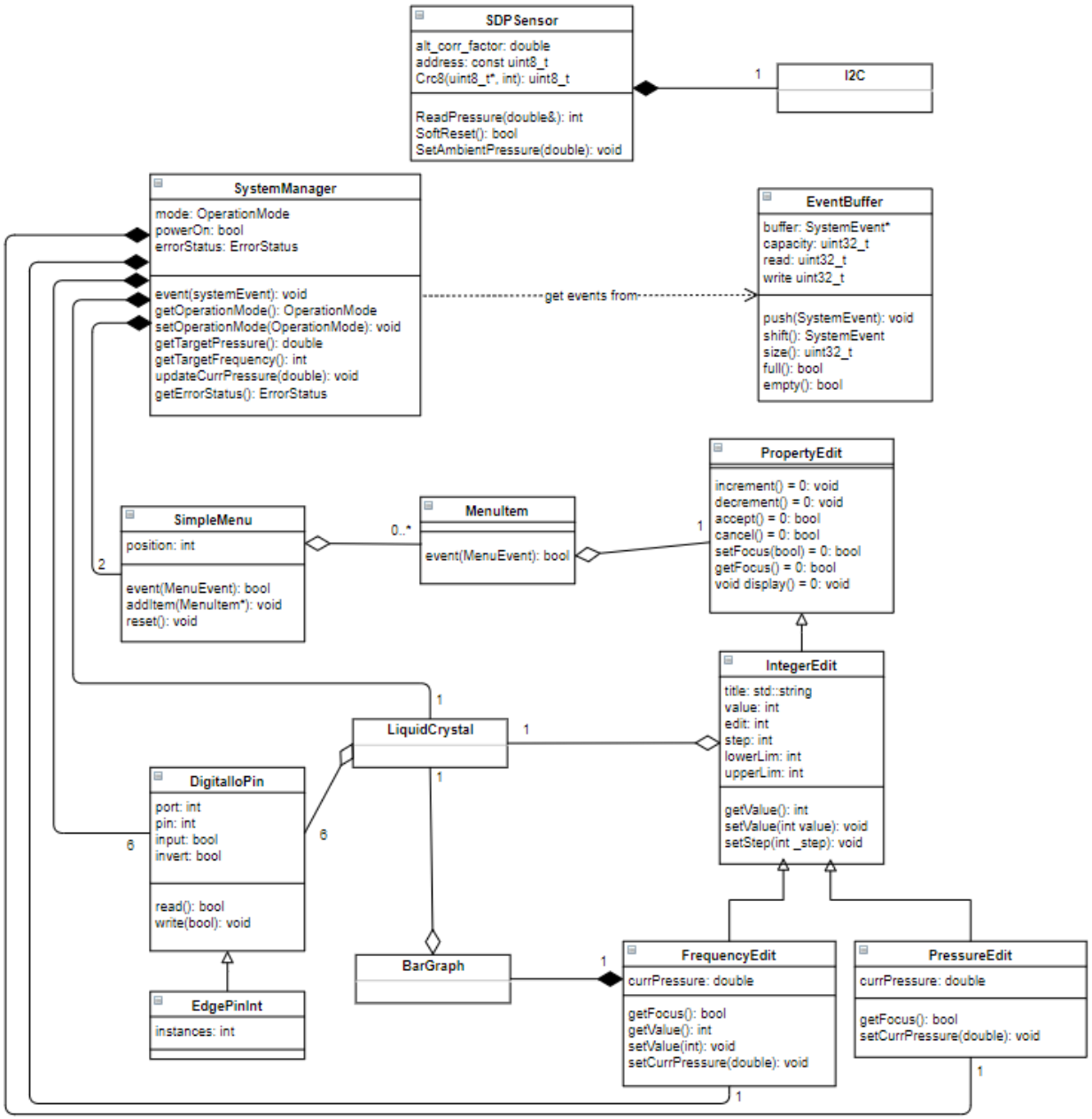
SOFTWARE IMPLEMENTATION OF THE SYSTEM

Introduction

The software implementation itself was done through a ”trial and error” experimantation in many parts, since many of the problem domain components were quite new to the team. In those areas where it was possible (mainly in the design of UI and the button/interrupt/system dispatch part) a more traditional OOP-based program design model was used. In other areas, like the modbus communication, the design followed a a more ”trial and error” approach.

THE UML DIAGRAM OF THE SOFTWARE

Figure X_R below shows the UML diagram of the program design.



FigureX_R. UML-diagram of the classes relevant to the project

The UML diagram only shows the classes which were relevant to the project. A few additional classes – mainly those involved in the modbus and UART communication and created by external sources – were excluded from the diagram.

The SDPSensor class was completely our own design, so it was included here. And since the SDPSensor class contains a pointer to the I2C class, the I2C class was included for reference also. The same applies for the BarGraph, LiquidCrystal, PropertyEdit classes: while they are external classes completely created by others, they are essential to the system UI and the SystemManager class as a whole. Therefore they have also been included here for reference and clarity.

The diagram itself shows a highly hierarchical, inheritance/composition based build where the UI consists of two different menus: one for the manual mode and one for the automatic mode. The Menus for each mode currently consist of only one MenuItem (an MenuItem associated with PressureEdit in automatic mode and a MenuItem associated with FrequencyEdit in manual mode), but it would be easy to augment the system to have more customized menu items for each mode. For example, each mode could have its own ”show status”-MenuItem, and the manual mode could have a ”timed frequency boost” MenuItem which could be used to boost the frequency of the fan for a specified time interval before stepping back to its normal frequency.

The SystemManager takes care of the entire UI. It also keeps track of the latest measured pressure. The main program sends it SystemEvents, which are mainly created by pressing different buttons. Pressing a button causes a pin interrupt, which puts an associated SystemEvent into the EventBuffer. Then, whenever there is an unhandled event in the EventBuffer, the main program sends it to the SystemManager.

For the buttons with pin interrupts, a special EdgePinInt class is used which inherits from DigitalIoPin.

OTHER NOTABLE SOFTWARE IMPLEMENTATION DETAILS

While the system specs themselves were pretty simple software-wise, a decision was made to try to develop the software in a way that would allow for better extendability later on. This resulted in a few notable design choices that would make it much easier to add new features in the future:

1. The system’s error status was defined to be of type *enum Class ErrorStatus*, so that new possible errors could be implemented later with as little effor as possible.

2. A support in the form of *SystemEvent::SELECT_SW_PRESSED* -event was added for making it easier to extend and control each mode’s menus/UI. In addition, new SystemEvents could easily be implemented into the code as a matter of fact,(the SystemManager’s event handling together with the EventBuffer functionality, works as a more sophisticated base for a state machine to handle any kind of event extensions.

3. The simple menu class was augmented with a reset-method. Even though it is not currently in use, it could be used to reset any datathat needs to be reset in a particular mode's UI.

4. The current EdgeIntPin class with pin interrupts and the EventBuffer can easily be augmented to both include more buttons in the system and to introduce new SystemEvents to be processed without having to make any real modifications to the main program.