1 The Sailcode Application Interface

The goal of this software rewrite is to make the code more modular. Modular code can be changed without worrying about breaking other parts of the code. Whenever a change is made to a module, we will run a *unit test* to check to make sure that it performs the same operation by handling the same input and output. This way if something has been changed in such a way that it would break other parts of the code, we will know.

The new API is going to contain as much code which has already been written in an effort to save time, since a lot of work went into it which did produce some good results.

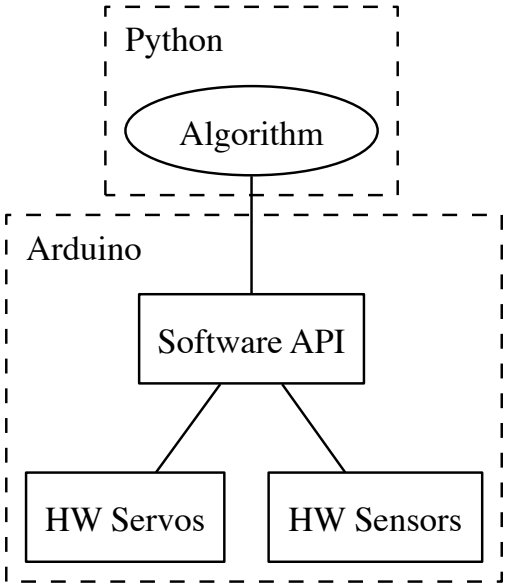

Figure 1: Rough overview of software location

2 Classes in Arduino

Although Arduino is written in C, it is possible to create classes with some limitations. You can create a Library in a seperate file/folder, which is compiled separately. Then in the Arduino source code, you can just include it and create an instance.

You cannot use the `new` keyword in C, but if you really need to declare an object off the heap, you can use the relloc and malloc functions which can do the same thing. Though these would be pretty dangerous if there ever to be a memory leak.

3 The Wind Sensor

Will be contained in its own class and carry all of its own supporting variables.

The sensor will just keep sending serial data at intervals rather than waiting for a command, so we have to have a way to keep the most recent value stored. This could mean having it update every-single time it gets a new string, then have the values ready for when a function is called, or it could ignore all messages until a function is called which requests an update of the variables.

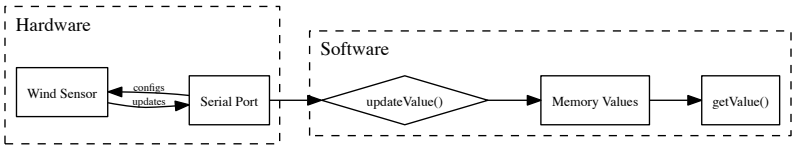

Figure 2: Wind Sensor to Memory path

When the wind sensor is asked to update its values from the sensor, it will call the series of functions in Figure 3 in order to get the values. It just breaks up everything required to get the data from the serial buffer, isolate an NMEA sentence, parse it into values and update the local object variables.

The wind sense object is going to have a variable which will store a partially completed sentence, which will continue to be completed by the pullNMEA function. Once the partial sentence is complete, it will raise a flag, and allow the validateNMEA function to operate on it. If it is valid the sentence will be passed on to the parse function, which will return an array of strings containing the isloated values. If there are two comma's, then a null value will be stored to

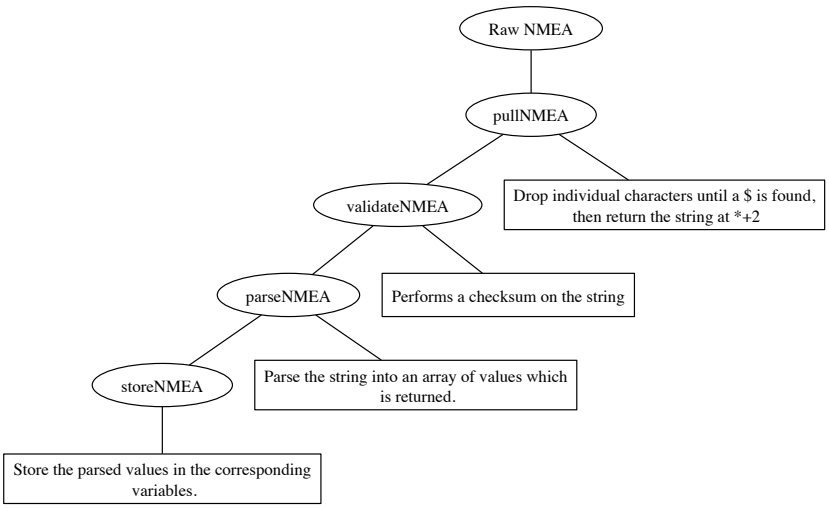

Figure 3: Wind sensor algorithm

that location. That way when the array is given to the storage function, it will be able to figure out why there are null values, and put something appropriate in its place.

Pulling the individual characters is done by calling the `grabNMEAChar` function, which accepts a single character and stores it in the `partSentence` variable, but only if it starts with a $ or if it has already started adding characters to a string. If it encounters anything else, it simply doesn't store the character. If it were reading from the serial buffer, this would also remove it from the serial buffer.

Once the function reaches an asterisk, it stops after two more characters so that it can include the given checksum. In this manner, we have isolated an NMEA sentence which can be sent through to the next function.