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.

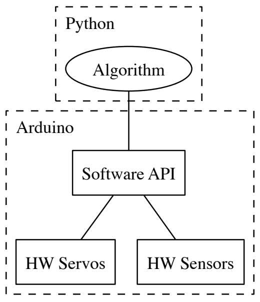

Figure 1: Rough overview of software location

1.1 Classes in Arduino

`http://arduino.cc/en/Hacking/LibraryTutorial`

The underlying software Arduino is built on attempts to implement the C/C++ language and is able to understand almost all of it. There are a few things that are not, such as the `new` and `delete`, but there is a work around for this. The one thing it will not support, is the use of template functions. Template functions allow you to make functions that can operate on arbitrary data types, and wouldn't really be useful for our purposes anyways.

When you write a class on the Arduino, you write it in a library. Where a library is just another set of source files which are included in the build. The Arduino IDE is very picky about where you put the library files, but the eclipse IDE just requires you to have a valid include statement. Check out the link above for more info.

1.2 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.

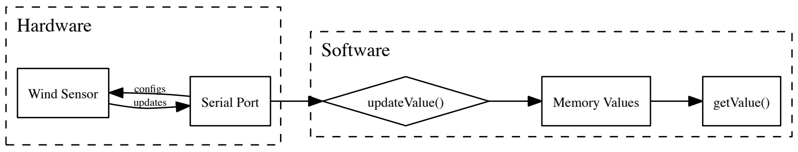

Figure 2: Wind Sensor to Memory path

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 isolated values. If there are two comma's, then a null value will be stored to 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.

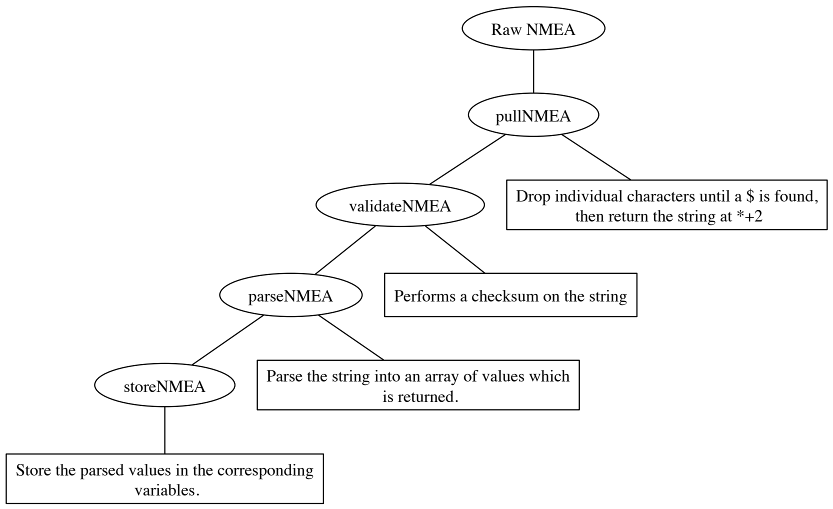
Raw NMEA
pullNMEA
validateNMEA
Drop individual characters until a $ is found, then return the string at *+2
parseNMEA
Performs a checksum on the string
storeNMEA
Parse the string into an array of values which is returned.
Store the parsed values in the corresponding variables.

Figure 3: Wind sensor algorithm

1.3 NMEA Parsing

Rather than using a built-in function, the sentences will be parsed one character at a time. The built-in strtok function has the advantage in that it is so flexible, whereas this function will only be able to parse NMEA sentences. At the same time, it should be able to parse any NMEA style sentence, since all it is going to do is break the sentence into an array of character-array strings, which will be converted into the correct values in the next function, storeNMEA.

The function starts with an array of arbitrary size, which should be greater than the maximum number of possible values included in an NMEA sentence. There are three strings involved and three index variables to keep track of their progress. The first is the index for the input NMEA sentence, which be incremented with every iteration until an asterisk character is reached and the function will stop. The second string is for building a new value and also has an index to keep track of how many digits have been added to it. The third string is actually an array of strings. This is what the function will return once it has finished. The array also has an index, but this is to keep track of the number of strings which has been added to it.

The parseNMEA function is meant to return an array of strings. However, I have opted to use strings in their most basic form, as an array of characters. Initially, the idea was to use an array of char array pointers, which have malloc assign them them memory based off of the length of the value. Given the complication of assigning memory though, we will instead set the size of the array in a static way. Simply using a 2-dimensional array of type char.

This function won't actually return an array anymore, instead it will modify a two dimensional char array which is handed to it as an argument. The declaration of the array is done in the WindSense object header file as a private member. The actual parsing of the strings isn't that tricky. The algorithm just needs to follow the decisions given in Figure 4.

Once an array of strings is created, they should contain both the identification tags for the NMEA string and the values themselves. Next the storeNMEA function will be able to take these parsed values and turn them into integer, double and char values depending on what it is.

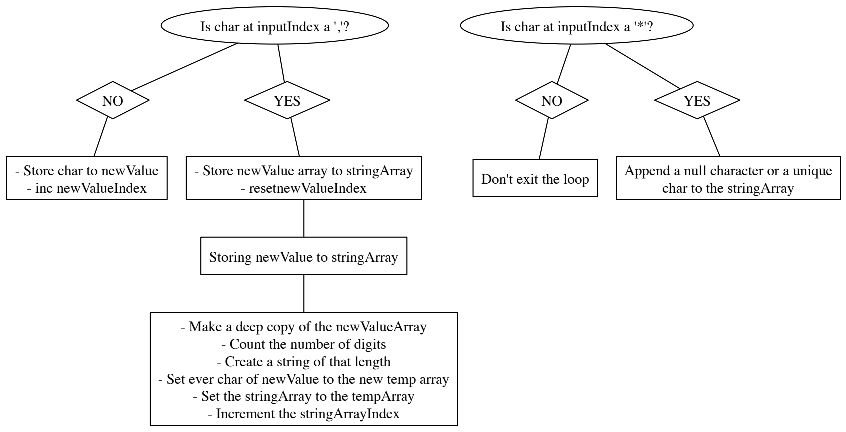
Is char at inputIndex a ','?
NO
YES
- Store char to newValue
- inc newValueIndex
- Store newValue array to stringArray
- resetnewValueIndex
Storing newValue to stringArray
- Make a deep copy of the newValueArray
- Count the number of digits
- Create a string of that length
- Set ever char of newValue to the new temp array
- Set the stringArray to the tempArray
- Increment the stringArrayIndex
Is char at inputIndex a '*'?
NO
YES
Don't exit the loop
Append a null character or a unique char to the stringArray

Figure 4: Parsing decision tree

It would definitely be less memory intensive to isolate each of the string values, then update the corresponding variable in the object, but this method lends itself to being more modular. If memory does become an issue, we can switch to the other method.

The while loop which makes up the core of this function has a weird condition. It loops as long as the input index is less than or equal to 1 or if the character just before the index does not equal the '*' character. This was done to allow the loop to execute once even after the asterisk was encountered. The first condition was included to prevent the second condition from accessing a negative index value from the input string on its first iteration. This is nice because if the first condition is true, then the compiler doesn't check the second condition.

1.4 Interfacing the Hardware

The wind sensor is connected to the Arduino through Serial port 3, which runs through the power board.

Data Rate	4800 BPS
Commmand	$PAMTC,EN,RMC,0,10

Table 1: Useful Specs

- The wind direction relative to the front of the boat
- The current GPS Location of the boat

2 Sailing Navigation

2.1 Navigation with GPS Data

The boat is going to be given certain points that it needs to reach. These will be given to it through a data file containing the GPS locations. Given that the boat cannot sail a straight line to the objective, it is going to need to deal with changing wind conditions and any other obstacles.

From this point on, the locations which the robot must reach will be called hard coded locations. In order to reach these locations, the robot will generate a series of in between points to take care of the challenges of sailing to the hard coded points.

```c
#include <stdio.h>

int main(int argc, const char *argv[])
{
        printf("This shit is bananas\n");
        return 0;
}
```