In my binary two bit representation the Morse string “-… .#-. .. -.-. .” is 10010101000111100100010100100110010001

dot = 01

dash = 10

space = 00

pound = #

1. The MEMsg class is entirely static, which means you do not create an instance/object in order to use it. Simply call the methods of the class with the scope operator (for example MEMsg::isEnglish(userInput);). The first method of MEMsg class that you should be aware of is the isEnglish() method function. Use this method to determine whether a string is in the form of English or Morse. The parameter of this method function—userInput—is the string that it will analyze. Return for isEnglish() is a bool value that indicates whether userInput is English (true) or Morse (false). Use this return value to determine whether to use the toEnglish() method or toMorse() method to translate the string. Before you do so however, you must call the initStaticVariables() method which will initialize the translation units that toEnglish and toMorse use for translating. Once you have called initStaticVariables (it has no parameters or return value), you can use the toEnglish or toMorse methods to translate the string. Both of these methods have parameter of type string the holds the string to be translated, and return type string that holds the translated string. Finally, the MEMsg class has the ability to write a Morse string to a file in binary (not string, so you can’t read or write to the file like a text file), and read that same file to retrieve and convert the binary to Morse. To see what these functions will write/read, set the verbose parameter to true. This will set the functions to output what they have written/read from/to the file.
2. I used two-bit binary encoding for send and receive.
3. I chose this approach because I believe it is the most strait forward method and one that doesn’t cause too many issues in terms of how to interpret it. You don’t have extra bits in between the actual message information as in 3-bit meta prefix, and you don’t have a varying amount of bits to read such as in signal simulation (which could cause problems interpreting between a pound 000 and a dot 0). The only disadvantage that this method has is that I couldn’t find a way to represent error codes. For example if someone enters … -- h # -- the only thing I can do is treat the h as a space. However considering we are assuming correct input, I don’t see this disadvantage as too big of an issue.
4. If there is an error code …….. I treat it as eight dots in binary (in my case eight 01’s). I don’t have any special binary encoding for this.
5. My send method needs an extra parameter of type string because I don’t store the input of the user in my class.
6. I solved the issue of what hex numbers to use for masking my bytes in the correct format/order. I also solved a particular issue of reading and masking the bytes in my receive functions when they would think a pound is a dot (I masked with & operator that would think a 1100 is 1000 or 0100).
7. A particular modification that I made to make this approach work better is by adding one extra byte to the beginning of my bytes that indicated the amount of bits that follow. This way I don’t have trailing 00’s.
8. The only thing I had to change in my pre-existing code was the driver so it would call send() and receive()
9. I chose unsigned char because it is clear and easy to use. I don’t feel it is necessary to use something from stl when primitives are easier and perhaps just as efficient to use (maybe you don’t get the exact amount of bits as in bitset, but still close enough). You also don’t have to worry about first converting to 0’s and 1’s or doing some other odd manipulation with the weird syntax of bitset. All you need is |= and & operators.