Exercise 1 Documentation

## Summary of Task:

Exercise 1 should take a pre-defined string from the program and load the memory address that references the start of the string into an index register. Once this has been completed 4 tasks involving bit manipulation must be performed on the string with the resulting modified string to be printed out on the terminal once the function has completed. The tasks are as follows:

1 – Make all of the letters lower case

2 – Make all of the letters upper case

3 – Capitalise only the first letter of each word

4 – Capitalise the first letter of the string, and the first letter after a full-stop

## Program Summary:

* The program defines required string and other variables
* Loads the starting address of input string into x
* Using subroutine “processstring” & “keepProcessing”
  + Accumulator B is loaded with byte 0+x to allow iteration
  + B is pushed onto the stack
  + Char is checked to be valid once transferred to accumulator A, and branched to “validchar” if valid & otherwise is skipped and the next byte is transferred in
* Using a subroutine, the current char is compared to a terminating character and will terminate if comparison returns true
* Another subroutine is used to check if the current char is less than decimal value 65, or greater than 122, as these the range of values for valid input characters
* Fullspace & capspace are used to check if a fullstop has just been encountered, and if it has then the next char is to be made uppercase, else ensure the char is lower case.
* Required function is then applied as decided in “validchar”
  + allToUpper
    - checks if the char present is already capitalised (less than decimal 90)
    - If it is greater than 90, 32 is subtracted from the value to get its lower case equivalent
  + allToLower
    - Checks if the char present is already lower case (greater than decimal 90)
    - If it is less than 90, 32 is added to attain the lower case decimal value
  + Capspace & fullspace
    - Checks for fullstop in current address, and space character in the following
    - If both return true, then char will be pulled from the stack and put into “allToUpper” and the following characters all run through “allToLower

## Discussion Questions:

* What happens if the end of the string is not detected?
  + If the end of the string has not been recognised by the definition given by the label “terminating\_character” when used in the function “processstring” with a CMPB operation, then the program will continue functioning. As the input string variable (“inpstr”) has been declared with a “fcc” command, and despite the fact that this operation doesn’t concatenate a terminating character, by default the string is delimited with the quotation marks and so the address of the final character of the input string is known. However if the end of the string is still not identified, by comparing the decimal value of the next char to be analysed with the range of values of valid characters it is ensured that once the end of the string is reached that the next function will begin, and if all functions have been performed then an “rts” operation is read.
* What else can go wrong?
  + Time taken for relatively long strings will start to increase noticeably due to the requirement of using branches instead of jump operations. However as the program is relatively small, and conditional jumps are required, branching operations are the most suitable operation to be used in this program.
  + There is the possibility that the string entered exceeds the available memory to store its contents, in which case it would be truncated. Although this is not a serious concern as the program would become unfeasibly slow, it would result in the memory being overwritten by the character manipulations which would then result in the following tasks receiving a different string to the original.
  + Non-English characters entered that may translate to valid decimal values
* How do you select which function you want to perform?
  + The selection is made in the sub-routine “validchar” by commenting out the functions that are not wished to be tested at the time. An iterative way was previously looked at where the task numbers (shown at the top) where used to set their precedence and could all be run one after the other, however this did not allow for full control and concise version testing. As all functions remain defined and it is just the branching that is commented out, it does allow the use of them in other functions which becomes particularly useful in tasks 3 & 4

## Testing:

* Inputting a string from the terminal
  + Check that the sub-routine is being called and run by setting a breakpoint in the RE subroutine.
  + Check that register X is loading the start of the memory’s location
* Is the string being loaded char by char
  + Use pull byte from stack to ensure
  + Go through debug memory address values
  + Single step to see if registers A & B change values at appropriate steps
  + Follow up to ensure valid chars
* Is the current char valid?
  + Is the current char upper case
    - Dec value 65🡪90
  + Is the current char lower case
    - Dec value 97🡪122
  + Is the current char a full stop
    - Dec value 46
  + Is the current char a space
    - Dec value 32

## Diagram:

