ASSESSED INDIVIDUAL WORK

COMP1006
LARGE COURSEWORK 01

Steven R. Bagley

Submission Deadline:
02/12/2022 15:00

Version: 1.01

Throughout COMP1006, we've looked at how we can write programs using ARM assembler. In this coursework, we'll create a program that can read in text typed into the keyboard and print it out to the screen formatted to fit a specific column width (e.g. if the column width was specified as 15 then each line of text printed should, in general, have no more than 15 characters of text printed on it). The coursework is split into four parts with each part getting you to build a more refined and more advanced version of the program. You will probably want to use your current solution to the as a starting point for the next iteration. This document outlines how the coursework is assessed, and descriptions of the various versions, along with some tips on how to implement them.

This coursework is worth 20% of your final COMP1006 mark, and will be given a mark out of 40. As ever, a mark of 40% (i.e. a mark greater than 16) or above is a pass mark for this coursework. The coursework has been designed such that completing the two basic versions of the program will give a pass mark. A standard git marking pipeline will be available for this coursework.

As ever, you might want to start by thinking about how to implement these routines in C, or a similar language before jumping straight into the assembler versions.

The skeleton .s files can be found in the following git repository:

`https://projects.cs.nott.ac.uk/2022-COMP1006/2022-COMP1006-LargeCoursework01`

Good luck…

01 Text Formatter — first steps

In the first iteration of the program, which must be implemented in the supplied `01-format.s`, you should implement a program that continuously reads in characters and prints them to the screen ensuring that the length of each line your program prints is below a specified limit. The line length is specified in the variable labelled `width`.

You should write a program that repeatedly reads a character (from the keyboard) and then prints it out (to the features window) until the hash character ('`#`') is typed, at which point your program show quit.

You should ensure that your program enforces the specified line length defined in the supplied variable, `width`. In other words, if `width` is 12, then after 12 characters are printed your program should print a new line character (ASCII code 10) *before* the thirteenth character is printed to force the text onto a new line. The program should then continue outputting characters on that line until 12 characters have been output and so on… Appendix A contains sample input and output for this exercise.

Some hints for completing this exercise:

- Your program should work for any positive integer value of `width` greater than zero.
- You will need to keep count of the number of characters that have been typed, and *if* `width` characters have already been printed then you will need to print a newline character to move to the next line.
- As shown in the example, spaces count as characters.
- The user may press the RETURN/ENTER key when typing, this should still cause the text to start again *from the beginning* of a new line — remember you can have up to `width` characters on a line. If the RETURN key is pressed, then `SWI 1` will return 10 in `R0`.
- You might find it helpful to print out a 'ruler' at the start of the program to help you check that the text is printed to the correct length — define a string containing width characters (e.g. for a `width` of 12, define a string such as '`123456789012`') and print it out at the start of your program. This will easily help you see if there are move characters printed per line (since the line will be longer than the ruler :)). Remember, to remove this ruler before submitting to the pipeline.

Assessment Criteria

The gitlab pipeline will mark your program and give you feedback on how well it works. Specifically, 10 marks are awarded for this iteration as follows:

- 2 marks are available for the program outputting the original text input
- 2 marks are available for the program breaking the text into multiple lines with no extraneous output
- 4 marks are available for the program correctly breaking the text into multiple lines
- 2 marks are available for the program correctly resetting when the return key is pressed.

02 Text Formatter — fixing the broken words

As it stands, the first iteration of the program will split words in two when it reaches `width` characters on each line. With this iteration, you are to modify the behaviour of your program so that your program waits until the end of the current word before breaking the current line and outputting text on the next line. We will define a word as a sequence of characters ended by either a space characters (ASCII code 32), the newline character (ASCII code 10) or the end of the input.

Therefore, your program should waits until the next space character (ASCII code 32) is typed after `width` characters have printed before moving on to the next line, unless the last character is a space in which case it should print a newline immediately. The extra space character after a word should not be printed (since the line break also signifies the end of a word). This will inevitably lead to some lines that are longer than `width` characters but will give printed output that is easier to read. Consult Appendix B for example output from this advanced variant.

Some hints for completing this exercise:

- As before, you should stop processing input when a '`#`' character is typed, and the RETURN key should be handled as in the previous iteration, and your program should work for any positive integer value of `width`.
- At the start of each line, your program will work identical as the in your first iteration above, it is only after width characters have been printed on a line that your program needs to do something different.
- Spaces count as printed characters — except when they don't… (see next bullet point)
- Think about the different states the output can be in at the end of a line: line ends with a word finishing as the last character on the line, line ends with a word that goes past the end of the line, line ends with a space as the last character on a line. Only in the final case should the space typed be printed.
- It is probably worth using your solution from the first iteration as a starting point for this iteration, so you'll want to copy your code from `01-format.s` to the supplied skeleton, `02-broken.s`

Your solution must be implemented in the file `02-broken.s`.

Assessment Criteria

The gitlab pipeline will mark your program and give you feedback on how well it works. Specifically, 10 marks are awarded for this iteration as follows:

- 2 marks are available for the program breaking the text into multiple lines with no extraneous output
- 4 marks are available for the program correctly outputting the text with the words printed on the expected lines
- 2 marks are available for programs handling the case of a space being the last character of a line.
- 2 marks are available for the program correctly resetting when the return key is pressed.

Successfully completing parts one and two will give you a passing grade for this exercise…

03 Text Formatter — keeping the line length below width

In the two previous iterations, you implemented a simple text formatting program that would attempt to keep the lines of text below a certain number of characters wide. However, with the first iteration your program would break words in half when it reached the specified line length, and in the second iteration your program printed some lines slightly longer than specified line length since it waited until the next space before breaking the line.

In this iteration of the program, you are to implement a version that will always keep the lines below the specified length, unless there is a word longer than the specified length (in which case, we have no option but to print the word in its entirety). To do this, rather than immediately printing the characters out as they are read, we will store the characters into memory first and then when we encounter a space character decide whether to print it on the current line, or to print it on a new line based on how many characters are in the current word.

Your solution for this program should be in the file `03-breaker.s`, and you may want to use your previous iterations as a starting point. As ever, the rules specified for previous iterations still apply to this iteration of the program, unless the new rules override them.

Example output for this version can be found in Appendix C.

Some hints for completing this iteration:

- As before, you should stop processing input when a '`#`' character is typed, and the RETURN key should be handled as in the previous iteration, and your program should work for any positive integer value of `width`.
- Your program will need to read characters and store them in memory until the end of the word is detected. You will need some memory allocated to store them, in the supplied skeleton you will see some memory labelled `buffer` for this. The characters are all a single byte long.
  Do not worry about the numbers already in the buffer, just overwrite them with your characters (defining a string like this is a quick and easy way to reserve sufficient space, the numbers making it easy to count how much space is available). You can assume that no word typed will be larger than the memory allocated.
- Once a complete word has been entered (the end of a word is signified by a space, newline, or the end of the input), then you need to decided whether to print it out on the current line or on the next line. The stored word should only be printed out if the number of characters already printed on the line (including spaces) plus the length of the stored word and the length of the space (1) before the word is less than or equal to `width`. Otherwise, the stored word should be printed on a newline.
- You will need to keep count of the number of characters that have already been printed on the current line (including spaces), and the length of the stored word.
- `SWI 3` can print out strings by giving it the address of the start of the string in `R0`, but the string must be terminated by a null character at the end (i.e. 0, not the digit '`0`').

- On input, words are separated spaces and/or RETURN characters. On output, the words should be separated by either a single space, or a newline character. If multiple spaces are entered between words, only a single space (or a newline character if it happens at the end of a line) should separate the words.
- The user may press the RETURN key when typing, this should still cause the text to start again *from the beginning* of a new line — remember you can have up to `width` characters on a line.
- When the end of file character is reached, any partial word currently stored should be printed.
- Think and test what happens in the corner cases…

Assessment Criteria

The gitlab pipeline will mark your program and give you feedback on how well it works. Specifically, 10 marks are awarded for this iteration as follows:

- 4 marks are available for the use of memory by your program
- 2 marks are available for the output being less than the specified line width
- 2 marks are available for correctly outputting the words on the expected lines
- 2 marks are available for removing multiple spaces from the input

04 Text Formatter — final version

The previous iteration of the program produced a version of the program that correctly breaks the text up into lines that are less than width characters (wherever possible). However, typing text into is not a nice experience since you cannot see the word you are typing since the word is not printed until you have finished typing it.

In this iteration of the program, you are to correct this by making your program display the input as it is typed, and then if the word makes the line too long you should move the word to the next line. To do this, you will need to implement a routine that can delete or rub out text that has already been printed. A single character can be deleted from the Features window output by printing the *backspace* character (it's equivalent to pressing the backspace key, and has ASCII code `8`), and so a whole word can be delete by printing sufficient backspace characters.

Your solution should be implemented in the supplied skeleton file `04-final.s`.

Some hints for completing this iteration:

- As before, you should stop processing input when a '`#`' character is typed, and the RETURN key should be handled as in the previous iteration, and your program should work for any positive integer value of `width`.
- You will need to implement a function/sub-routine called `rubOut`. This should print out a number of backspace characters to *rub out* something that has already been printed — the number of characters to rub out must be passed in `R0`.
- This version should work in a similar fashion to the third iteration although it will now print characters as they are typed.
- You will need to think about the various states that the program can be in when a word ends: the word could fit on the current line with space to spare, the word could fit on the current line exactly, the word does not fit on the current line and needs to move to the next line.
- Think about when spaces need to be printed…
- What happens when a word is too long to fit on a line, where does the next word start?

Assessment Criteria

The gitlab pipeline will mark your program and give you feedback on how well it works. Specifically, 10 marks are awarded for this iteration as follows:

- 3 marks are available for correctly implementing the `rubOut` function subroutine.
- 3 marks are available for correctly breaking lines according to the rules.
- 2 marks are available for ensuring that no extraneous spaces are printed.
- 2 marks are available for ensuring that characters aren't reprinted unnecessarily.

APPENDIX A

Example output for the first iteration of the Text printer program, given the typed input:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean ultricies porttitor tellus congue semper.

The program should print (note the space at the end of the first line) for a width of 12:

```
Lorem ipsum
dolor sit am
et, consecte
tur adipisci
ng elit. Aen
ean ultricie
s porttitor
tellus congu
e semper.
```

APPENDIX B

Example output for the second iteration of the Text printer program, given the typed input:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean ultricies porttitor tellus congue semper.

The program should print (note the space at the end of the first line) for a width of 12:

```
Lorem ipsum
dolor sit amet,
consectetur
adipiscing elit.
Aenean ultricies
porttitor tellus
congue semper.
```

APPENDIX C

Example output for the third iteration of the Text printer program, given the typed input:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean ultricies porttitor tellus congue semper.

The program should print for a width of 12:

```
Lorem ipsum
dolor sit
amet,
consectetur
adipiscing
elit. Aenean
ultricies
porttitor
tellus
congue
semper.
```