7 | Exercise 2: Less Simple Output

The original lab manual used an LCD display example as a less simple peripheral exercise in order to display text. We left the original exercise description in this manual (starting at Section 7.1) but **we will not implement the LCD output**. However, you should read through this LCD exercise and try to understand how that would have been implemented. If you have questions, consult your lab demonstrator

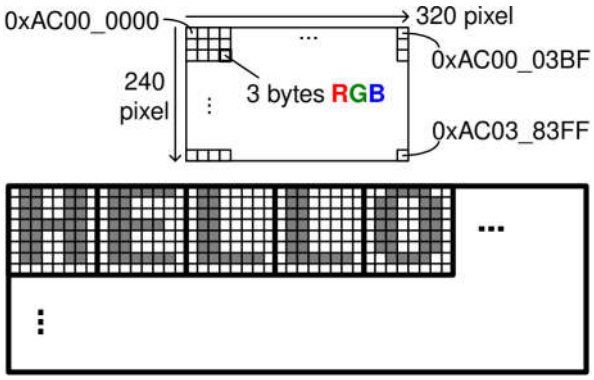

Figure 7.1: Screen buffer memory organisation.

For our virtual lab, we are building a terminal window in our frame buffer instead (displayed in the Virtual Screen window that you start with Komodo). You can see this as a very basic graphics card that uses shared RAM as a frame buffer[1]. Here, the frame buffer is a bitmap that stores 3 bytes for each pixel representing the RGB (red, green, blue) brightness values, as illustrated in Figure 7.1. The screen buffer has the following specification:

- Width: 320 pixel
- Height: 240 pixel
- Start address: `0xAC00_0000`
  pixel data is stored row-by-row starting from the top-left pixel
- Colour: 3 byte RGB per pixel (960 bytes per row)

[1]This is a concept that was commonly deployed in early home computers. That went that far that for example in the Commodore 64 (of which about 15 million units were sold), the CPU clock was actually defined by the required video output speed. There had been two versions of the Commodore 64: one for the US market supporting the NTSC video standard at a CPU clock of 0.985 MHz and a European version supporting the PAL video standard running at 1.023 MHz (see `https://en.wikipedia.org/wiki/Commodore_64`). Ironically, at that time, Europe had the better television standard (PAL) because television started later in Europe which allowed for implementing a more advanced standard. To give you an appreciation for these early home computers, instructions took about 3 CPU cycles on a Commodore 64 (`https://sta.c64.org/cbm64mctime.html`) and the CPU had only a very limited set of 8-bit instructions and no multiplier. Nevertheless, those home computers had been powerful enough to run arcade games and simple office applications. Hence, you can get an idea what you could get out of a modern microcontroller.

The goal of this lab is to implement a terminal window that is displayed on the video output. For this, we are using am 8×8 pixel bitmap font. You can use the font provided in Chapter 28 on page 95 (a file is provided on the materials website). With 320×240 pixel, we are implementing a terminal window of 40×30 characters. The terminal window should be able to display text messages and to set/move a cursor. You should implement this exercise in a bottom-up manner:

1. Implement a function `printc` that prints one (ASCII) character into our terminal window
2. Think about how you have to handle a cursor? It is not important in this exercise to show the cursor position, but your terminal window should move the cursor position in a sensible way.
3. Think about methods to pass parameters to functions and return results from functions (you may also have a look into Section 18.2).
4. Write a function `printstr` that prints a `Hello World!` message on the screen that is using your function `printc`. You should call your `printstr` function by passing a pointer as a reference to the beginning of the string. You should think about how strings are terminated or how the length of a string is encoded? There are multiple answers to this answer[2], and in this lab, we use zero terminated strings (i.e. the end of a string is indicated by the "magic" symbol `0x00`). This is also known as `NUL`-terminated strings. Think about the pros and cons of zero-terminated strings (including security issues)!
   For calling functions inside functions, you have to think of memorising your function return address. An elegant way to do this is using stacks, as introduced in Section 9.

7.0.1 Useful extensions

- The ASCII standard does not only define the encoding of characters, it also includes some control sequences[3]. The following table list some relevant control sequences for this lab that you could support:

code	name	function
08	Backspace (BS)	Move the cursor left one place
09	Horizontal Tabulate (HT)	Move the cursor right one place
0A	Line Feed (LF)	Move the cursor down one place
0B	Vertical Tabulate (VT)	Move the cursor up one place
0C	Form Feed (FF)	Clear Screen
0D	Carriage Return (CR)	Move the cursor to the start of the line

- Use only one line of the window, but display a message that gets scrolled through in the case it doesn't fit this line (like in a news ticker).
- Display a cursor.
- We will add a keyboard later, but you can already think of a terminal window that can buffer more lines that you currently see. This means, you should be able to scroll back (and forth) some lines in your terminal window.
- You could add some support for test colour printing.

[2] You may like the following background discussion "The Most Expensive One-byte Mistake – Did Ken, Dennis, and Brian choose wrong with NUL-terminated text strings?" by Poul-Henning Kamp (online: `https://queue.acm.org/detail.cfm?id=2010365`).

[3] An issue with these control sequences is that the exact behaviour is not strictly specified (or obeyed), which means that different terminal windows may display the same ASCII encoded message differently. A famous example is the different encoding of line feeds of text files in Windows and Linux systems. Standardisation is often a political rather than an engineering exercise (in particular if big industry players are involved). Nevertheless specifications should be free from misunderstandings (just in the case you will ever be involved in such things).