Prologue

In the labs, you will work on guided exercises that will lead to a lift emulator at the end. In the first two labs, you will be constructing basic data structures and the algorithms for the lift controller. In the last two labs, you will implement the lift emulator on the AVR development board provided later. The outline of the lift emulator is as follows. Further information will be given in the next labs.

- The lift you will be emulating has floor select, emergency, open and close buttons inside the lift. On each floor, there will be a single button to call the lift.
- Your lift will be able to travel up and down amongst 10 floors numbered 1-10.
- Multiple requests may be given and should be served like in a real-life lift. Examples:
    - somebody may push the button on floor 8 while the lift is moving up to floor 7
    - When the lift is passing the 5th floor upwards to the 8th floor, the request for floor 3 and 4 will be serviced after servicing floor 8.

Lab 1 (Week 2 and 3)

Instructions

Complete each task and demonstrate the working program to your tutor. Tasks should be demonstrated using AVR Studio's simulator. **Part A of this lab must be marked by the end of the lab session in week 2. The rest of this lab must be marked by the end of the lab session in week 3.**

Part A – 16-bit addition (1 mark)

Load the 16-bit numbers 40 960 and 2 730 into register pairs r17:r16 and r19:r18. Add them together and store the result in register pair r21:r20. You can directly load the numbers into registers as constants.

When that's working try adding the numbers 640 and 511. Does your program deal with overflow correctly?

Part B –division (2 marks)

Write a program that divides a 16-bit unsigned integer by an 8-bit unsigned integer. Assume that the dividend and the divisor reside in registers of your choice. Your program should output the 16-bit quotient and the 16-bit remainder in registers.

Test corner cases such as : maximum / minimum, divisor > dividend.

Part C - binary to ASCII Conversion (3 marks)

Write a program that loads a 2-byte unsigned integer from the program memory into registers and converts it to ASCII representation. Store ASCII results in data memory.

Data memory should be allocated using the assembler directives instead of hard-coding the address.

This conversion will be later required to print the current floor number of the lift on an LCD display.

Part D – In-place array sorting (4 marks)

Write a program that:

a. first stores a sorted single byte integer array (1, 2, 5, 7, 8, 12, 20) in program memory using *.cseg* and *.db* directives;
b. copies the array from program memory into data memory using a loop (use the assembler directives *.dseg* and *.byte* to reserve space in data memory. You can assume that the maximum array size is 256 bytes);
c. and then inserts a single-byte value (from a register) into the array (residing in the data memory) while retaining the sorted order. If the value is already present in the array, do not insert it again.

Then extend your program as follows:

a. first store a sorted single byte integer array (1, 2, 5, 7, 8, 12, 20) in program memory;
b. copy the array from program memory into data memory using a loop (let's call this array in data memory A);
c. store a queue of single-byte numbers (0, 10, 1, 25, 6) separately in the program memory using assembler directives *.cseg* and *.db* (let's call this array B);
d. then iteratively (one-by-one) insert each number from array B to the array A (in the data memory) while retaining the sorted order. If the value is already present in the array, do not insert it again.

Caution: It is important that the elements in array B should be inserted into A at the same location of memory, instead of creating another copy of A in the data memory.

Note that this program will be later extended to build the data structure to decide the service requests for the lift. So do not give up.