NSWI170 – **Computer Systems** – 2024/25 Summer – Martin Svoboda

Assignment **A4: Counter**

401. [**Decomposition into classes**] As usual, we organize all the necessary code into appropriately designed classes or universally applicable global functions.

402. [**Display class**] Class for our segment display will represent the display as a whole. Decomposing it into separate positions would not make sense, because these positions do not form separate functional units. Moreover, it would not even bring us any other benefits.

403. [**Adherence to terminology**] To avoid potential mutual misunderstandings, it is necessary to abide by the established terminology when naming. In other words, we have 1 *display*, it consists of 4 *positions*, each of which contains 8 individual *segments*. Since we consider only the decimal system, we have *digits* `0` to `9`.

404. [**Logical position numbers**] In order to simplify the user interface and possibly also the code portability, we will assume a logical numbering of display positions in the direction from right to left. I.e., position on the right has number `0`, position on the left `3`. With the exception of the low-level code over masks, we will work with these logical positions wherever possible.

405. [**Display functionality**] In addition to the initialization function, the display will offer at least a low-level function to display an arbitrary glyph, and then another function to display a particular digit. While we describe the intended position / positions using a mask in the first case, logical position number will be used in the latter case. We will gradually add more such functions at a higher level of abstraction during the following assignments in order to simplify the use of our display by its users as much as possible.

406. [**Calculation of constant values**] Although our display has 4 positions in particular and thanks to the `byte` data type used for the position masks we even cannot have much more of them, we again do not want to hard-wire such a fact in our code. In other words, we should not forget to define all relevant constants, systematically name them, and also calculate their values from the other constants.

407. [**Array for digit glyph masks**] We will define glyph masks for digits within a translation array. It is expected we provide these masks in a binary form, e.g., `0b11000000` for digit `0`, because it is the values of individual bits that correspond to the logical problem we are solving here. At the same time, let us add that we cannot emplace masks of any other glyphs than digits within this array, that would breach its semantic meaning.

408. [**Low-level error handling**] It is obvious that a user may call our display functions with invalid parameter values, e.g., an invalid digit to be displayed. We will intentionally not detect or treat such situations, though. Within low-level programming and especially in the case of the C++ language, it is a common practice for reasons of efficiency. We simply carefully describe the conditions of use of our functions and it is up to the users to respect them. If they do not, it is considered their mistake, the program behavior becomes undefined, and it may even end up crashing.

409. [**Display initialization**] In the `setup` function, as usual, we first need to initialize our display. In addition to setting the required pin modes, this also means turning off all segments on all positions. Analogously to the diodes, this fact is again not guaranteed.

410. [**Built-in pow function inaccuracy**] We will probably need to calculate various powers within our code. However, it is not possible to use the built-in function `pow` for this purpose, since it does not perform accurate calculations. Simply because it works with data type `float`, i.e., with floating point numbers. Therefore, we do not have exact arithmetic above them, nor have such numbers themselves exact representation.

411. [**Custom function for powers**] In other words, we will need to introduce our own function to calculate powers. Let us note that it should be universal, and so allow to work with any bases. On the other hand, only non-negative exponents will suffice.

412. [**Efficient calculation of powers**] We should also focus on finding an implementation of this function that would be reasonably efficient. Specifically, we should avoid a recursive solution. Depending on a chosen approach, we may also try to do without `if` conditions if possible. Although not necessary, we can even try to propose a solution with better time complexity than linear with respect to the exponent value.

413. [**Calculating powers of two**] However, it still applies that it is more efficient to calculate the powers of `2` using the bitwise shift operation, not the general function we introduced.

414. [**Changing input parameter values**] It is generally not a good practice to change values of input parameters of our functions, as this could give a false impression that they are output parameters. All the more so because the language really offers us such a mechanism. We just did not get acquainted with it on purpose, it was not necessary.

415. [**Recycling local variables**] If we start using a local variable inside a function for some purpose, we should not suddenly start using it for a completely different purpose later just because we do not actually need the original variable anymore and we are also satisfied with the matching data types. Instead, we should start using another new variable in that case.

416. [**Context of variables validity**] We will always declare local variables only at the point where we really need to start working with them, not automatically all at the beginning of a given function body. In addition, we also declare them only inside the most specific nested block where they are needed, so as not to extent their context and lifetime unnecessarily.

417. [**Counter class**] As expected, we will encapsulate all the logic and data members related to our counter application in a separate class. It will use the services of buttons and display as needed, but they themselves must not solve any aspect of the counter, or even be aware of its existence.

418. [**Current selected position**] This also means, for example, that the display as such must not be responsible for remembering the currently selected position, since that undoubtedly belongs to the logic of our application.

419. [**Maximal counter value**] Due to the method of deriving the maximal allowed value of the counter and the version of the language used, we will no longer be able to use a constant expression, but we can still use at least a constant variable.

420. [**Extracting selected digits**] Undoubtedly, there are various ways how to obtain a digit at a given order of the current value of the counter, but no matter how we do it, we should again avoid an inefficient solution, i.e., we can call our function for calculating powers at most once.

421. [**Array of precalculated orders**] Precalculating all relevant powers and storing them in memory will not be a suitable solution either. And not only because such an approach would not be realistically scalable to larger displays.

422. [**Extraction of digits by display**] In relation to this extraction, let us note that it should definitely not be handled by the display class as such, because we understand it as a driver of an output display device, which is not even supposed to understand such things.

423. [**Unnecessary display updates**] It goes without saying that we will change the displayed glyphs on the display only if it is genuinely necessary, i.e., if there really have been any changes in the counter value or the selected position. Just for reference, one such change is even about $30\times$ slower than calling the already slow `digitalWrite` function we used for work with the diodes.

424. [**Using debugging dumps**] Let us not forget that when we run into problems while implementing our assignment, we can use our own debugging dumps sent from the Arduino to the computer via the serial line in order to find and eliminate the errors.

425. [**Final code cleanup**] Let us add, however, that such fragments should be commented out or otherwise disabled in the final code. Specifically, it is not possible to leave any parts in the code that are not relevant, completed or correct, not even commented.

426. [**Disallowed system functions**] In addition to the already prohibited system functions, we will not use `bitRead` or `pow` functions, too.