NSWI170 – **Computer Systems** – 2023/24 Summer – Martin Svoboda

Assignment **A4: Counter**

Binding Instructions | Well-Meant Advice | Ideas for Thought | Common Mistakes

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. [**Display functionality**] Display will offer at least a function to initialize it and a low-level function to display a required glyph at a selected position, in both cases based to the provided masks. It will also offer a function to display a specific digit at a given position, but this time at the logical level, i.e., not using masks. 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.

404. [**Adherence to the terminology**] To avoid potential mutual misunderstandings, it is necessary to strictly 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`.

405. [**Logical numbering of positions**] 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 therefore has number `0`, on the contrary, position on the left has `3`. With the exception of the low-level code over masks, we will work with these logical positions wherever possible.

406. [**Calculation of constant values**] Again, we do not want to hard-wire the specific number of available display positions anywhere in the code, even though it is clear that we do not have much freedom in this sense anyway. I.e., we use data type `byte` for position masks, which simply cannot offer more than 8 positions. Nevertheless, we will again calculate all the constants that can be derived from the others.

407. [**Array for digit glyph masks**] We will define glyph masks for digits within an array, which we will then also use for their translation. 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**] We can easily imagine that a user may call our display functions with invalid parameter values, e.g., a non-existent digit to be displayed. We will intentionally not detect or treat such situations, though. In the case of low-level programming, it is a common practice for reasons of efficiency, especially in the case of the C++ language. 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, because this fact, analogously to diodes, is not guaranteed.

410. [**Inaccuracy of the built-in pow function**] 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 are the numbers themselves representable exactly.

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, we should certainly avoid a recursive solution. Depending on the 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 value of the exponent.

413. [**Calculating powers of two**] Of course, it is more efficient to calculate the powers of `2` using a bit 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 of 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. We simply start using another new variable in that case. This will have no impact on the speed of the code, compiler optimizer can handle such situations easily.

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 the function body. In addition, we also declare them only inside the most specific nested block where they are needed, so as not to expand their context and lifetime unnecessarily.

417. [**Counter class**] As expected, we will encapsulate all the logic and data items related to the counter functionality in a separate class. Of course, 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. [**Currently 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. [**Display and counter initialization**] Although we have to show the initial value of the counter on the display during the counter initialization, again for the same reason, this fact cannot have any effect on the necessity of performing a complete and unchanged initialization of the display itself, i.e., including the already mentioned turning off all its positions.

420. [**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.

421. [**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.

422. [**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.

423. [**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.

424. [**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 about 30× slower than calling the `digitalWrite` function we used when working with the diodes.

425. [**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 to find and eliminate the errors.

426. [**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.

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