NSWI170 – **Computer Systems** – 2022/23 Summer – Martin Svoboda

Assignment **A3: Counter**

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

301. [**Decomposition into classes**] All code related to the operation of diodes, buttons and timers, as well as code for the counter itself, will be encapsulated in appropriately designed classes. Their data members will then exclusively be private.

302. [**Universal functions**] An exception to the previous requirement could be standalone global functions if they are usable universally even outside of the context of our particular problem.

303. [**Systematic names for constants**] Given that the number of various constants in our program is increasing, it is reasonable to start naming them in a systematic way, i.e., so that it is clear just from their names at first glance what they relate to (diodes, buttons, …).

304. [**Analogy of buttons to diodes**] In general, we have basically the same or similar requirements for working with buttons as we had with diodes before. Therefore we represent them using an appropriate class, maintain a global array of their instances, etc.

305. [**Logical button numbers**] Specifically, this also means that we will again use logical numbers `0` to `2` when working with buttons `B1` to `B3` instead of low-level pins corresponding to constants `button1_pin` to `button3_pin`.

306. [**Class for button representation**] Class for buttons must encapsulate all the data members and methods we need to be able to work with a given button, including the necessary internal states or timing. Specifically as for the timing, we will of course use the timer class we already have available.

307. [**Concealing internal implementation**] Interface of public methods of buttons must be designed in a way to be as simple, intuitive, and elegant for the users as it is possible. In other words, we want to deliberately hide all the internal implementation or technical details so that we do not have to understand them from the outside, let alone solve or control them somehow.

308. [**Public methods of buttons**] Each button will offer at least a method for its initialization and methods through which we will be able to query the occurrence of button press and release events. When it comes specifically to pressing, we will even allow queries on specific variants, i.e., the initial or recurring long-hold events, as well as a general press event covering both the variants without distinction.

309. [**Possibility of repeated queries**] Query functions must be implemented in a way that they can be correctly called repeatedly within one run of the `loop` function. We will thus have to isolate the internal logic of detecting a change in the button state and completely separate it from the querying, and hence solve it within a special update function, which we will always call exactly once at the beginning of the `loop` function.

310. [**Activation of recurring presses**] If we did not want to handle recurring button press events, it would be sufficient to ignore them, of course. However, it would be better if we could explicitly enable or disable such a functionality during the button initialization.

311. [**Button debouncing**] As a part of the button internal logic, we should also be capable to filter out short state fluctuations caused by mechanical aspects and button imperfections. Specifically, we will work with the idea that a state change (both press and release) must last continuously for at least, let us say, 10 ms in order to register it. If the commenced intention is violated during this time, no change will occur at all. On the contrary, until a successful transition really takes place, we will continue to function without any change, e.g., we will not stop triggering any recurring press events.

312. [**Commenting button logic**] In particular, more complicated aspects related to the state logic of buttons deserve to be carefully commented directly in the code and meaning of the performed operations explained.

313. [**Use of selected buttons only**] The way in which we will work with instances of individual buttons in our code cannot be influenced by the fact whether we really want to work with all of them or just with some selected ones. Therefore we need to have them all available, only the application itself (i.e., our counter) will determine which ones it really wants to work with and which ones it does not.

314. [**Assigning functions to buttons**] Assignment of user functions handling the respective events of our individual buttons must not be hard-wired in the code, so we must be able to change it easily. In this sense, it is fully sufficient to use suitable named constants.

315. [**Class for counter representation**] Entire logic of our counter will again be solved by a suitably designed class. As for the interface of its public methods, we must distinguish at least counter initialization, operations changing its value, handling of events triggered by buttons, or displaying its value on diodes.

316. [**Separation of application from drivers**] At the same time, it is necessary to add that the implementation of our counter, diodes and buttons must be consistently separated from each other. Counter, as an analogy to the user application, will of course use the services provided by diodes and buttons, but from the opposite point of view, it is necessary to strictly ensure that our drivers for diodes and buttons do not solve any aspect of the counter, they must not even be aware of its mere existence.

317. [**Representation of counter value**] Even though we will have to obtain binary decomposition of the counter value in order to display it on diodes, the counter as such should remember information about this value at the logical level, i.e., as an ordinary integer number. It is certainly not appropriate to use an array of binary digits, let alone an array of diode logical states.

318. [**Counter overflow checking**] Value of the counter as such must always fall within the allowed interval, therefore we must check for a possible overflow during each increment or decrement operation. In addition, this must be handled without any undue delay in order to ensure consistency at the expected level of atomicity.

319. [**Correction of overflowed value**] If an overflow occurs, we would certainly be able to adjust the new value using conditional branching. However, it is also possible to do it via a simple calculation without branching, which is certainly the preferred option. Just be aware that the modulo operator for integer division can return negative numbers, too.

320. [**Maximal value of the counter**] When dealing with overflows, we certainly cannot do it without determining and then using the maximal allowed value of our counter. We will of course introduce this value using a named constant. However, be careful that it can be derived from other already known information, and therefore it is necessary to calculate it and not define it as a fixed literal.

321. [**Initial counter value**] Part of the counter initialization at the end of the `setup` function should be setting and displaying its current, i.e., default value. Even though it is specifically equal to `0` in our context, it might not be the case in general. I.e., we could legitimately want to start with some other initial value.

322. [**Distinction between initialization and reset**] Although in general we should try not to repeat similar or even the same code fragments, it is also necessary to prefer preserving the expected logical meaning of the code over its technical form. Specifically, the content of our functions for initializing and resetting the counter can be analogous, but they are semantically different operations, and therefore we cannot combine them into just one function.

323. [**Calculation of binary decomposition**] When calculating the binary decomposition of a counter value, we should focus on its efficiency. In particular, we should avoid functions that calculate general powers. I.e., specifically for powers of two, we can easily do without them. We just need to elegantly use selected bitwise operations and masks.

324. [**Inappropriate optimization of diodes**] On the other hand, let us note that some seemingly straightforward optimizations, on the contrary, can be inappropriate. E.g., setting a new diode state regardless of the current state will actually be faster than retrieving the current state and causing it to change only when necessary.

325. [**Conversion of boolean values**] If a boolean value is expected somewhere in our code, we should not rely on automatic language-driven conversions to determine it (`0` means `false` and anything else `true`). For example, the result of bit operations is a number, so we first need to convert it to the required logical value explicitly, for example, using comparison.

326. [**Simple loop content**] Like the `main` function in a normal program, the `loop` function should not contain any complex or low-level code. On the other hand, it also does not make sense to take all the intended actions and just wrap them into one auxiliary function that would just be called here.

327. [**Pseudo-unified for loop**] Loops are generally suitable in situations where we expect each of their iterations to look, let us say, similar. Therefore, if we have buttons and each of them is supposed to invoke a different action, it does not make sense to handle the events triggered by them using a `for` loop, so that we would in turn need branching for individual situations using a `switch` or otherwise. Such a loop would then be completely meaningless.

328. [**Independence of individual buttons**] Individual buttons are independent on each other, therefore we must be able to handle even situations when an event is triggered by several of them at once within just one run of the `loop` function.

329. [**Unnecessary diode updates**] Since the lighting state of diodes remains valid until the next change, it is certainly suitable to avoid setting of the diodes according to the counter value in every single run of the `loop` function. In other words, we only do it when it really becomes necessary.

330. [**Disallowed system functions**] In addition to the already prohibited system functions, we will also not use function `bitRead`, because we can easily do without it.