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

Assignment **A3: Counter**

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 decomposed and encapsulated in appropriately designed classes. Their data members will then be solely 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. [**Separation of application from drivers**] Implementation of our counter, diodes, and buttons must be consistently separated from each other. Counter, as an analogy to a user application, will, of course, use 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 will not solve any aspect of the counter, they must not even be aware of its mere existence.

304. [**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, …).

305. [**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.

306. [**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`.

307. [**Class for button representation**] Class for a button driver encapsulates all data members and methods we might need for seamless work with the buttons, including necessary internal states or timing. Its implementation must be universal and robust enough, because we cannot preconceive how in particular they will be used by our applications.

308. [**Concealing internal implementation**] Interface of public button methods will be designed in a way to be as simple, intuitive, and elegant for the users as it is possible. In other words, we again want to deliberately hide all the internal implementation or technical details so that we do not need to be aware of them from the outside, let alone understand them or be forced to deal with them in any way.

309. [**Types of button events**] Although we are only interested in the button press event within this task, we will also find useful to detect the release event in the future. When it comes specifically to the press event, it could even be useful to distinguish its specific variant, i.e., whether it is the initial or recurring events when the button is held for a longer time.

310. [**Merging press events**] On the other hand, it would also be useful to have the possibility to query the button press events without distinguishing such variants of the initial or recurring presses, simply because we want to handle them exactly the same way (like us right now).

311. [**Event occurrence detection**] Event detection depends not only on the return value yielded by the `digitalRead` function, but also on the correct work with our internal button state logic. This means that such a detection is not repeatable, though. If we tried it for multiple times in one iteration of the `loop` function, we might get unexpected results. So either we rely (without any guarantee) on the discipline of our users that repeated calls will not occur, or we simply separate event occurrence detection from querying over them completely.

312. [**Repeated press detection**] However we decide, even the press detection using the `digitalRead` function as such cannot be invoked repeatedly. Not only its execution is very slow, but, at the same time, each call could give different values (if the actual state has already changed).

313. [**Public button interface**] In addition to the button initialization method, we will offer an update function (for the actual event detection) and query functions (for individual types of events). We will always call the update function for each button at the beginning of the `loop` function, remember the detected conclusions internally and declare them valid for the entire given iteration of the `loop` function. Thanks to this, it will then be possible to safely carry out queries even repeatedly.

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

315. [**Button debouncing**] As a part of the internal button 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.

316. [**Timer class exploitation**] For the timing control within the buttons, we will, of course, use instances of the timer class we already have from the previous assignment.

317. [**Commenting code**] At least the more complicated parts of the code should be accompanied by sufficiently explanatory comments to make their understanding easier. In particular, we can focus on, for example, solving the state logic of buttons and explaining the actions performed.

318. [**Use of selected buttons**] 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.

319. [**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 will distinguish at least the counter initialization, operations changing its value, handling of events triggered by buttons, or displaying its value on diodes.

320. [**Separation of counter methods**] Specifically, the counter increment and decrement methods will not invoke displaying of its value on the diodes in order to ensure consistent separation of data manipulation from its visualization and so independence of these operations.

321. [**Counter value representation**] 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.

322. [**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.

323. [**Correction of overflowed values**] 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 operation for integer division can return negative numbers, too (this is important for the correct implementation of the decrement action).

324. [**Maximal counter value**] When dealing with overflows, we 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.

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

326. [**Displaying the initial value**] And it is not just about setting this value, we also have to ensure that we will display it on the diodes.

327. [**Order of initialization actions**] Initialization actions performed during the `setup` function should, by nature, be arranged in such a way that we first deal with the hardware aspects and only then with the application aspects.

328. [**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.

329. [**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.

330. [**Unnecessary diode updates**] Since the diode lighting state remains valid until the next change and calling the `digitalWrite` function is very slow, it is appropriate to avoid readjustment of the diodes in every single run of the `loop` function. In other words, we will only perform it when it becomes necessary, i.e., only when the counter value really changed. And even in that case, we will give instructions only to those particular diodes whose state we really need to change.

331. [**Assigning actions 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.

332. [**Misused for loops**] 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 the individual cases using a `switch` or otherwise. Such a loop would then be completely meaningless.

333. [**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.

334. [**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 just take all the intended actions and only wrap them into one auxiliary function that would solely be called here.

335. [**Disallowed system functions**] In addition to the already prohibited system functions, we will also not use functions `bitRead` (because we can easily do without it), and `pow` (is not accurate).