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

Assignment **A6: Messages**

601. [**Basic display extension**] We will preserve the existing functions of the basic display and add new ones according to our needs. It will at least be a function allowing to display an arbitrary letter of the English alphabet and subsequently also a function for displaying arbitrary symbol.

602. [**Domain of supported symbols**] This second wrapping function will accept a single arbitrary character in the form of the `char` data type, and determine a correct glyph to be displayed for all of the following situations via branching: the already mentioned letters of the English alphabet, decimal digits, an empty glyph for any white character (space, tab, etc.) and likewise an empty glyph for any other, say, unsupported character.

603. [**Suppressing the size of letters**] Specifically as for the letters of the English alphabet, we will suppress their size. This means that we will support both lowercase and uppercase letters in the input, but we will only have one single glyph for each corresponding pair.

604. [**Passing symbols by value**] Both the mentioned functions will expect a parameter of type `char` and not a pointer to such a character. This will lead to faster and more universal implementation since we will be able to call them over separate characters as well as individual characters belonging to strings or arrays.

605. [**Temporary debugging glyphs**] For debugging purposes, it will be worthwhile to temporarily use some other, non-empty, and in particular different special glyphs instead of the expected empty glyphs for spaces and unsupported characters. Thanks to that, we can distinguish these situations (including the explicitly requested empty glyphs) from each other visually, and thus increase the chance of getting a correctly working code. Only before the assignment submission do we adjust both the constants accordingly.

606. [**Preserving position numbering**] Although it would make sense to align letters on our display to the left as opposed to digits, we nevertheless preserve the convention of numbering display positions from `0` to `3` from the right to the left.

607. [**Text display class**] Similarly to the numerical display from the previous assignment, this time too we will program our text display using the inheritance by deriving it from the basic display. It will then be able to display a string of the requested characters using the time multiplex mechanism.

608. [**Driver and application separation**] The goal of this task is to implement displaying of the running messages. This does not mean that such a functionality should be implemented in full directly by the text display as such, though. Let us not forget that we understand it as a hardware device driver. Its interface and capabilities may thus be non-trivial, but it must be universally applicable and limited to reasonably expected general functionality. On the contrary, it cannot preconceive and solve specifics of applications that might want to use it in the future and which, above all, do not even exist yet.

609. [**Text display interface**] The text display driver will therefore remember and be able to display only and only four particular symbols. That is, an array of characters with the length exactly corresponding to the number of positions we have on the display.

610. [**Alternative string representation**] Instead of an array of characters, we could alternatively or even additionally store the corresponding array of the already translated glyphs. On the one hand, this will make it possible to display arbitrary special glyphs, but it will suppress the logical nature of the entire text display on the other.

611. [**Setting the requested string**] Inside the function for setting a new string, we need to copy all the characters into our internal data member. In other words, it would not be enough to just remember a pointer to this string since we do not want to force the caller to guarantee us the immutability and even the very existence of such a string for the entire time during which we will use it.

612. [**String constancy flag**] In terms of the interface, we will expect a pointer to a constant string, i.e., `const char*`. By using the constancy flag we are saying that we are not interested in changing the content of this string, which the compiler will then help to enforce. More importantly, however, we would not be able to even call this function with constant strings otherwise. And that includes anonymous strings written as literals, too.

613. [**Input string length**] Even though we primarily expect that the length of the provided string will correspond exactly to the size of our display, we will nevertheless implement the entire function in a way that we will correctly handle even shorter or longer strings, respectively. In the former case, we append them with spaces from the right, in the latter case, we cut them off and simply ignore the rest.

614. [**Treatment of low-level errors**] If we did not want to detect the mentioned situations, it would certainly be in accordance with the convention we already explained in the previous task regarding the treatment of similar low-level errors. However, in order to learn how to work with strings better and to understand the consequences of such errors, we will treat them carefully this time.

615. [**Array bounds checking**] We specifically need to prevent potential reading or even writing beyond the array end. In the best case, these errors will lead to an immediate crash of the program when accessing unallocated memory, in the worst case, we will overwrite some of our other data. The problem can then manifest itself in strange behavior of the program at any time later and it will be very difficult to debug it.

616. [**Strings and arrays of characters**] A traditional string in C is modeled as an ordinary array of characters, which, however, contains one special termination character `'\0'` at its end. Without it, we would not be able to recognize this end, since strings themselves are not aware of their length.

617. [**Use of the pointer arithmetic**] If we want to sequentially process a whole string or at least some larger continuous part of it, it is faster to manually use pointers instead of the square bracket operator. So, instead of a loop over the position numbers and accessing the elements using the `string[i]` construct, we will use a gradually incremented pointer and the `*` operator for dereferencing.

618. [**Limiting the number of passes**] We will process the input string using only one pass, no more are needed. We also do not need to calculate the length of this string.

619. [**Filling the internal array from outside**] Finally, let us add that the data member of the internal array for the current characters is owned by the text display, so we need to fill the content of this array from inside using the discussed setting function. In other words, it is not possible to provide a pointer to this internal array to someone else to fill it from the outside, thus losing the control over it.

620. [**Extended control functions**] As we have already stated, the text display cannot solve the actual logic of the running messages. On the other hand, this does not mean that, in addition to the basic function for setting the entire string, we cannot offer some others, for example, allowing to set characters on particular positions, or for basic logical modification of the current string, let us say in the sense of its shifting or appending.

621. [**Display turning off**] As with the numeric display, we will also implement the text display in a way that we have the option to explicitly turn it off.

622. [**Class for running messages**] We will encapsulate the entire functionality of running messages in a separate class. It will, of course, use and control the text display to achieve the necessary behavior.

623. [**Further display class extensions**] The messages class must be completely isolated from our text display, as well as we cannot implement it as its extension using the inheritance. Simply because it is not a driver, but a specific application.

624. [**Volatile data members**] Each class should only contain data members that have a long-term nature. We should therefore not use them to store purely temporary or auxiliary values for which ordinary local variables inside functions would suffice.

625. [**Concealing internal methods**] Similarly as we normally use private data members, we should also mark as private methods that are of a purely internal nature and are not intended for direct use by the users, especially if their thoughtless execution could cause inconsistencies or other complications.

626. [**Retrieving messages to scroll**] In order to obtain messages intended for displaying, we create an instance of a pre-implemented class `SerialInputHandler` available via the attached `input.h` header file. It allows to receive messages sent from a computer to the Arduino via the serial line.

627. [**Alternative sources of messages**] Although we will solely use the described message retrieval mechanism within this task, we want to support alternative approaches as well. This means that our running message class cannot hard-wire our default source in any way, and thus it needs to allow the entire process to be controlled from the outside through appropriately designed methods.

628. [**Messages class public interface**] This involves at least a method for setting a new message to be displayed, but it could also be useful to enable detection of a successfully completed message run, or provide even other methods depending on the chosen implementation.

629. [**Controlling the entire process**] Having displayed the entire current message, perhaps the most reasonable response is to stop the internal message scrolling mechanism, do nothing else, and simply wait for another instruction for starting a new message, if any. This will elegantly allow for one-time use, repeated use of the same message, or even subsequent alternation of different messages.

630. [**Setting up a new message**] Although it would be preferable to make our own copy of the provided message string, we will just store a pointer to this string. In other words, this time we will rely on the fact that this pointer will be valid and the original string available all the time.

631. [**Dynamic memory allocation**] The reason is that our message can have an arbitrary length unknown in advance. If we really wanted to make its copy, we would have to use the so-called dynamic memory allocation. We would like to avoid this mechanism in this course, however, because its use requires a certain level of circumspection and discipline. Otherwise, we could uncontrollably and irreversibly lose available memory in a running program.

632. [**Arbitrary message length**] Length of a message requested to be displayed can really be arbitrary, including zero. However, we will never get an invalid pointer.

633. [**Redundant display changes**] It goes without saying that we give instructions to the text display only when a change occurs, i.e., at the moment the current message is scrolled to the next position.

634. [**Disallowed system functions**] We avoid dynamic allocation, it is not necessary, and we did not learn it anyway. Likewise, we will not use any resources offered by the `cstring` library.