|  |  |
| --- | --- |
| **Mark** | **/11** |

|  |  |  |  |
| --- | --- | --- | --- |
| Team name: | *A5* | | |
| Homework number: | *HOMEWORK 10* | | |
| Due date: | 1/12/2024 | | |
|  |  |  |  |
| Contribution | NO | Partial | Full |
| Alessio Spineto |  |  | *x* |
| Riccardo Lamarca | *x* |  |  |
| Sofia Cecchetto |  |  | *x* |
| Annamaria De Togni |  |  | *x* |
| Emma Crespi |  |  | *x* |
| Notes: none | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Project name | Keyboard and encoder | | |
| Not done | Partially done  (major problems) | Partially done  (minor problems) | Completed |
|  |  |  |  |
| **Part 1a: Scan each column and read the keyboard using a timer interrupt**  As suggested by the project hints, we started by setting as GPIO the PINs connected to the keyboard. The ones set as *output* are the column drivers, the ones set as *inputs* are used for the row readout.  We set TIM2 in interrupt mode (and enabled its global interrupt) to trigger the readout at a rate equal to 100 Hz.  For the USART communication we enabled the global interrupt and configured the DMA in the “memory to peripheral” direction.    We first define a few useful variables:   * *SAMPLE\_RATE* = number of times we are going to sample a single column; * *matrix[4][4]* = matrix that stores the values of each button; * *p\_matrix[4][4]* = matrix that stores the status of each button   + 1: button pressed;   + 0: button not pressed;   + -1: button status already sent via UART; * *Keyboard[4][4]* = matrix that stores how the buttons are mapped as characters.   Every time we enter in the callback of the timer we consider the column j of the matrix (which corresponds to the column j of the keyboard).  In the timer callback, after checking which handler called the function, we read all the rows of column j (initialized as 0).  To tackle the debounce problem, for all columns, we sum and store the values of the buttons of each row in the corresponding *matrix* positions. Since we sample each column *SAMPLE\_RATE* times, when we reach the last sample of the last column (j = 3 and k = SAMPLE\_RATE\*4 - 1) we check all *matrix* values: if the value of the matrix is equal to *SAMPLE\_RATE* and *p\_matrix[i,j]* is 0 it means that the button of the keyboard has been under constant pressure. In this case the corresponding value on *p\_matrix* is set to 1 meaning the corresponding key must be sent via UART. If instead *p\_matrix[i,j]* is -1, meaning that the button position has already been managed, and *matrix[i,j]* is 0, meaning that data has already been initialized, we set also *p\_matrix* to 0. In this way, *p\_matrix[i,j]* is ready to handle new incoming data.  In all the other cases the value is discarded by setting *matrix[i,j]* to 0.    Then the column j is disabled by setting its pin to GPIO\_PIN\_RESET, the j counter is then incremented, and the next column is enabled. Doing so at the end of the callback and not at its start prevents that two columns are enabled at the same time, due to the timer’s period being too low. Finally, the variable k used to count the number of samples, is incremented.  In the while loop, the status of the buttons is checked at every iteration. If the status is 0 or -1 nothing happens, otherwise the button status is set to -1 and the corresponding key is sent via UART. | | | |
| **Part 1b:** Read the encoder position and send to the PC the rotation speed in rpm  The encoder is connected to pins PC6 and PC7 of the STM32, controlled by TIM3.    We set up TIM3 in encoder mode, with a counter period (ARR) of 65535 and Internal Clock Division (CKD) set to division by 4.      We set TIM3 to encoder mode TI1 and TI2, where every mechanical step results in a ±4 increase of the timer counter. We defined the parameters for the two channels to obtain a positive increment when the encoder is turned clockwise, and negative when turned counterclockwise.  We set the digital input filter to 15, to use N = 8 consecutive events and a sampling frequency  where to validate a transition on the output.  We start TIM3 in encoder mode and TIM2 in interrupt mode. TIM2 is set to send an interrupt every second.    We also set USART in DMA transmit mode and enabled the global interrupts for TIM2 and USART2.  In the callback of the timer, we keep track of two consecutive values of the position of the encoder in the variables *old* and *new*, by reading the value of the counter using *\_\_HAL\_TIM\_GET\_COUNTER()*.  We use *\_\_HAL\_TIM\_IS\_TIM\_COUNTING\_DOWN()* to detect if the counter is decreasing or not, to understand in which direction the encoder is being rotated.  We compute the difference between the *old* and *new* variables and store it in *variation*.  If the sign of the variation and the direction of rotation are coherent (e.g. counter increases when the encoder is turned clockwise and decreases when turned counterclockwise) or there is no variation, we compute the speed of the rotation.  Otherwise, we handle the overflow and underflow of the counter.  We detected two scenarios in which this could happen:   * If the variation is negative and the rotation is clockwise, it could be due to overflow of the counter. To obtain the correct variation, we compute the value corresponding to the actual mechanical steps that occurred. To do this, we sum the total number of values of the counter (65535 + 1) to the (negative) variation. * If the variation is positive and the rotation is counterclockwise, it could be due to underflow of the counter. To obtain the correct variation, we subtract the total number of values of the counter (65535 + 1) to the (now positive) variation.   We compute the rotation per minute in the variable *rotation.* We multiply *variation* by a factor obtained by dividing by 48 (a full rotation of the encoder) and multiplying by 60 (seconds in a minute) to convert the measurement acquired in rotations per minute (rpm). We send the computed speed to the remote terminal, along with the *old* and *new* values of the counter to check the result.  Example of output without overflow/underflow management    Example of output with overflow/underflow management    **Note for the professor:** We noticed that very quick and repeated changes in the rotation direction of the encoder can result in incorrect speed estimation, probably due to rotation direction and position being read at two different times, thus triggering the over/underflow management.    We weren’t able to mitigate this issue by reading two consecutive values of the variable *expectNegative*. A possible solution could be an additional check of the range of values considered, to only do over/underflow evaluation if the variation suggests this issue.  This kind of evaluation would be based on the fact that the encoder is being rotated by a human hand, and for this reason very fast changes are not likely. | | | |
| Professor comments: | | | |