|  |  |
| --- | --- |
| **Mark** | **A** |

|  |  |  |  |
| --- | --- | --- | --- |
| Team name: | *B5* | | |
| Homework number: | *08* | | |
| Due date: | Tuesday, 28th November, 08:30 | | |
|  |  |  |  |
| Contribution | NO | Partial | Full |
| Ghidini Alessandro |  |  | *x* |
| Latino Francesco |  |  | *x* |
| Luppi Eleonora |  |  | *x* |
| Bravin Riccardo |  |  | *x* |
| Feltrin Elia |  |  | *x* |
| Notes: | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Project name | *Encoder readout* | | |
| Not done | Partially done  (major problems) | Partially done  (minor problems) | Completed |
|  |  |  | *x* |
| Explanation:  The goal of this project is to perform the readout of the encoder and print the rotation speed in rpms on the terminal through the UART in DMA mode.  The homework also requires understanding how the encoder works and what is the meaning of the parameters which can be selected and modified during the setup.  **Part 1:**  To read the value of the encoder every second, we have set up Timer 1 with the internal clock source, a prescaler of 8400-1, and a counter of 10000-1, in addition to activating the necessary interrupts (update interrupt).    To manage the encoder reading, Timer 3 has been set up with a combined channel in encoder mode, internal clock division equal to 4, channel 1 rising edge, and channel 2 falling edge. On both channels, the digital filter is set to 15.    For communication via UART, we have enabled USART2 with the respective interrupts and set up DMA as the transfer mode with default parameters.    After generating the code, just before the *while(1)* loop we started the two timers with *HAL\_TIM\_Encoder\_Start\_IT(&htim3, TIM\_CHANNEL\_ALL)* and *HAL\_TIM\_Base\_Start\_IT(&htim1)*.  We then overrode the *HAL\_TIM\_PeriodElapsedCallback()* function, where we in order:   * save the counter inside an *int16\_t* variable using *\_\_HAL\_TIM\_GET\_COUNTER(&htim3)* * reset the counter with *\_\_HAL\_TIM\_SET\_COUNTER(&htim3, 0)* after each reading to avoid calculating the difference each call and simultaneously to avoid the over/underflow problem.   Note that by operating in this manner or calculating differences as explained in point 2, there remains a limit given by the number of maximum rotations per reading. In other words, there is no way to know the direction of movement if the encoder exceeds 32767 counts.  Finally, we composed the output string, calculating RPM as counter/24\*60, and transmitted it via UART using DMA.  **Part 2:**  The use of *int16\_t* variables instead of *uint16\_t* ones is enough to overcome the issue of wrong rpms computation due to over- and underflows, because the MCU recognizes that the stored value must be considered a 2’s complement number. This happens for values which are larger or equal to 32768, as we can see:  32767 = 0b 0111 1111 1111 1111  0b 0111 1111 1111 1111 + 1 = 0b 1000 0000 0000 0000   * -32768 in 2’s complement   Considered as 0b 0111 1111 1111 1111 + 1 = 0b 1000 0000 0000 0000 with negative sign  This means that values from 32768 to 65535 are considered negative because the MSB (sign bit) is set to 1, so the MCU recognizes the 2’s complement value, indicating the interval from -32768 to -1.  We can consider two examples to better explain the solution:   * For the **overflow**, we look at two sequential iterations of the code and see how the datum is considered when stored in the variables.   int16\_t val1; // 65535 = 0b 1111 1111 1111 1111  int16\_t val2; // 0 = 0b 0000 0000 0000 0000  // considered as -1 in 2’s complement  int16\_t delta = val2 – val1  The MCU computes the difference between the two values as the sum between the second value and the 2’s complement of the first value. Since we are using an *int16\_t* variable, the latter is considered as -1, so all the bits are inverted and we add 1 (as seen before). Finally, the MCU can perform the difference as a sum between the two values:  0b 0000 0000 0000 0000 +  0b 0000 0000 0000 0001 =  -----------------------------------  0b 0000 0000 0000 0001  è +1  The *int16\_t* variables correctly handle the overflow problem.   * At the **underflow**, we have the opposite situation. As before, we can consider two sequential iterations of the code:   int16\_t val1; // 3 = 0b 0000 0000 0000 0011  int16\_t val2; // 65535 = 0b 1111 1111 1111 1111  // considered as -1 in 2’s complement  int16\_t delta = val2 – val1;  In this case the MCU computes the difference between the two values as the sum between the second value and the 2’s complement of the first value and then considers the result’s 2’s complement value for the correct computation:  0b 1111 1111 1111 1111 +  0b 0000 0000 0000 0000 =  -----------------------------------  0b 1111 1111 1111 1111  è -1 (0b 0000 0000 0000 0000 + 1 with negative sign)  In this way *int16\_t* variables also handle the underflow problem.  **Part 3:**    To avoid hardware debouncing we use a digital filter: by choosing the internal clock division CKD we define a sampling frequency to read the output of the encoder.  We choose a slow frequency since we are going to rotate the encoder by hand, so we choose CKD division by 4 and input filter as 15, so the reading is valid after 8 equal samples at a frequency of .  In the internal configuration of the encoder there are two switches that are maintained open by pull-up resistors. The two switches can be closed by rotating the encoder in one direction or another. When one switch is closed, we can see a LED is turned on. For example, if we rotate clockwise the green (connected to the line of channel A) is turned on before the red one (connected to cannel B); otherwise, if we rotate counterclockwise the red is the first one to be turned one. This means that the signals produced in channel A and channel B are phase-shifted by 90°.  The quadrature of phase is used to compute the rotation direction based on which signal first arises or falls, depending on the selected polarity.  The encoder mode tab is used to select the resolution we want which is equal to how many events to read at each rotation: counting only on T1, T2 or both. Then we can define the polarity of each channel to have an increment or decrement when rotating clockwise. If we select one channel with rising edge and the other with falling edge, the counter will increment when rotating clockwise. If we select both channels’ polarity as rising edge or falling edge instead, the counter will decrement when rotating clockwise.  Therefore, the polarity selection is used to indicate which rotation direction has to be considered as positive depending on the sequence of edges generated when rotating the encoder:   * If we select rising or falling edge for both the channels the timer considers positive counts for the direction of rotation which generates first a rising (falling) edge on TI1 and then on TI2, so the counterclockwise direction, while the other one generates negative counts. * If we select falling edge for channel 1 and rising edge for channel 2, this means that the MCU considers the direction of rotation which generates first a falling edge on TI1 and then a rising edge on TI2, so the clockwise direction in our case. This also explains why setting rising edge for TI1 and falling edge for TI2 does not change the counting sign for clockwise rotations. | | | |
| Professor comments:  OK, very good! | | | |