Tufts University
Electrical and Computer Engineering Department

Senior Design Project: NerdGirls
Instrumentation Panel Group

ADDENDUM

Gladys Magh and Megan Schwartz

May 12, 2003

ADDENDUM: Interrupt Code Results

An investigation into the functionality of the interrupts was performed late in project development, and while the results are not as desired, they will assist in the development of future code for the instrumentation panel group. On the suggestion of Matthew Heller, the primary consultant, separate code was written to do nothing other than run a PORTB-triggered interrupt. While this was not successful, the following code displays a timer-triggered interrupt that is the same in concept and different only in the prompt.

The three concepts that have been proven work are: an interrupt can be triggered, an incrementing counter can be the trigger, and the results of the ISR can be displayed through the incremented rev_count value. The only part of this that is not functional is the fact that it was not proven that PORTB (the four external interrupt pins) can prompt the ISR instead of the timer.

The code, named mattcode_timer_int.asm, uses the same configuration bit settings and variable names as previously defined in the original code for the program. The vector declarations differ only in that the interrupt routine vector is stored in addresss 0x0004, instead of 0x0008, as before. The interrupt save context section clears the TMR0 overflow interrupt flag in the INTCON register, as well as re-enables the Timer0 interrupt again. If these two lines are not included, then the interrupt will run once, but then never again. This section also saves the current values of the WREG and the STATUS register, to be restored after the ISR is completed.

The interrupt case statement is there to stop interrupts other than the timer interrupt from running the ISR. If the timer did indeed trigger the interrupt, the ISR_TACHOMETER will run. This code reads PORTB to clear the interrupt and clears the zeroth bit of the INTCON variable. These two lines are from the previous code, but do not interfere and were thus left in the code. The rev_count (number of revolutions occurring by the wheel) is incremented here as well. The ISR finally restores the STATUS and WREG values as it stored them before, and returns to the main program.

The program consists of the same data table, and LCDInit call as before. The UART Initialization was left in the code for future work, as well as the PORTB interrupt initialization (albeit commented out). It is important to keep the commented sections there so that future groups may see what was previously attempted by our group to run the PORTB interrupt. The main part of the code that is important is the initialization of the Timer0 interrupt bits. The TMR0H and TMR0L registers, which store the value of Timer0, were set to 0xFFF0 in order to count down time before the interrupt gets triggered. The INTCON register is set to enable both high and low interrupts, as well as the TMR0 overflow interrupt. Inside the Timer0 Control Register, T0CON, the TMR0 interrupt was enabled also. Finally, rev_count was initialized to zero.

The main code begins with a bit set test on the zeroth bit of the rev_count. If the bit is zero, then the even-interrupt message is displayed. Else, the odd-interrupt message is displayed. This means that MESSAGE_EVEN and MESSAGE_ODD code is the same for displaying data on the LCD, just different output to the screen. A three second delay is allowed after either message, and then the program loops back to the Main.

This successful code will run the TMR0H:TMR0L registers each up to 0xFF (or 0xF0), then triggers the interrupt once the FF rolls over to 0x00. There is an estimated 2-second count that occurs between the interrupts from running. As displayed in the watch window, the rev_count value is incremented each time the TMR0 interrupt runs. This shows our group that an

interrupt can be run successfully when prompted by the timer, and code inside the ISR runs successfully as well.

Thank you to Matthew Heller for his assistance with this code.

References

Predko, Myke. *Programming and Customizing PICmicro Microcontrollers.* McGraw-Hill, 2002. Page 849.