EET363 Introduction to Microcontrollers
OIT Portland West, Fall 2010

Lab Assignment #4 — Using Interrupts
Due November 25 (Thanksgiving Day)

Objective: The student will use an interrupt service routine to implement a clock function and polling of the user interface.

Project Description: Even when the Star Trek was being filmed 40 years ago, processors had interrupts. So let's make use of an interrupt service routine to improve the Captain Pike's signaling device of Lab Assignment 3. In that assignment, coming up with the half second delay was basically a trial and error process. In this assignment, the delay will be as exact as possible, and will be much easier to calculate.

Equipment and Software needed:

- Everything you used in Lab Assignment 3
- While you can develop your program using the simulator, use the Dragon12-Plus for final testing.
- You may use the C compiler, if you wish (and know C).

Part 1:
Add an interrupt service routine to your program to handle the RTI interrupt and configure the RTI interrupt to occur as close as possible to a 1 millisecond interval. The interrupt service routine should increment a word variable named *count*. If you are using C, be sure to declare the global variable *count* as *volatile*.

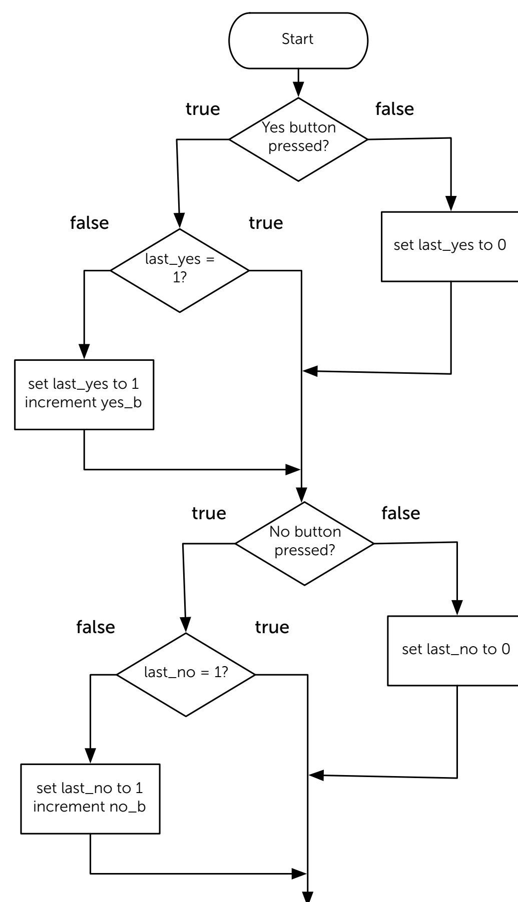
Replace the subroutine that used timed loops to generate the 500 millisecond delay. The subroutine will need to fetch the value in count and add to it the number of RTI interrupts that occur in 500 milliseconds to get the time in the future when 500 milliseconds will have passed. Call this value *futureTime.* Note that the value to add will not necessarily be 500 because the interrupt rate isn't exactly 1 millisecond.

Then the subroutine just loops until the time has passed, namely futureTime-count < 0. The measurement is far more accurate than just using the software loop, since the software loop time will be lengthened if other interrupts occur during its execution.

Check your results by either accurately measuring the time the LED is on, or using the simulator to log changes to the port (which will also tell how long the LED is on). If you use the simulator you will need to use minidbug12 mode but without the Dragon-12 I/O simulation in order to log the changes to the parallel ports.

Part 2:

Normally we want input devices (the push buttons in our case) to be handled in interrupt service routines so that polling them unnecessarily doesn't slow down the system, yet we can service the device as soon as possible. We could use the key interrupt feature to cause an interrupt when the button is pressed, however the mechanical switch contacts bounce and each press/release cycle will appear as many. Switch "de-bouncing" is accomplished by using a low pass filter. This could be an RC network, but that isn't needed if our low pass filter is simply to check the switch at a slow rate.



In Lab Assignment 3, we effectively de-bounced the push buttons by not checking the button during the time the light is flashing. But in this assignment we will check the buttons at a reduced rate of about 100 Hz in an interrupt service routine. This is slow enough that we won't be affected by contact bounce but fast enough that we can detect multiple presses.

Modify your RTI interrupt service routine so that once every ten times it is entered the buttons are checked. Add four variables to your program named *yes_b, last_yes, no_b* and *last_no*. These variables should be initialized to zero. Yes_b and no_b have a count of the number of yes and no button presses while last_yes and last_no have the value of the button during the last button test. Implement the algorithm on the right to detect and count the number of button presses.

Then in the main part of your program (called the "idle loop" since it only executes if there is no interrupt routine running), change the code to check the yes_b and no_b variables rather the the buttons. If yes_b is non-zero, decrement yes_b and flash the light twice. If no_b is non-zero decrement no_b and flash the light once. Add an

additional second of “off time” to the end of each flashing sequence so that pressing “yes” twice is distinguishable from pressing “no” once.