EE247 Programming for Embedded Systems Project #3 Spring 2020

"Interrupts Control Things!" Program
Expected to be working in three weeks (April 20, 2020)

The goal of this project is to do the same kinds of things that we did in project #2, but do them using multitasking and interrupts. We want to include reading the pushbuttons, setting the LED's, running a stepper motor (or overlapped LEDs) at various speeds and either direction, and reading the photosensor and potentiometer. To that we add being able to read the time and set a clock. (Make it an alarm clock and beep a speaker – PWM to an output - when it goes off!)

What is different is that the main program forever loop is to do very little – primarily reset the Watchdog timer, but also send out any needed messages (if using a non-buffered print) to the user. The project is to use interrupts in the following ways:

1. The clock will run off of an interrupt (RTC), which will keep time to at least the nearest second. The user can ask for the time, or set the clock. (This is pretty easy.)

2. The stepper motor will be interrupt driven (or using the TPM channels; I'll allow either). At regular intervals (as generated by a timer) it will step forward (or reverse) at speeds controlled by the user. Preferred: use a TMP timer channels to run the stepper. This requires learning to use some of the more interesting and useful features of the timer. This is the most important "embedded" control issue – you want the motor to operate smoothly in real time.

3. The A/D converter will use an interrupt for sampling the photo sensor and pot. That is, a command merely initiates the conversion. The completion triggers an interrupt that ultimately results in the user receiving a message reporting the value in Volts. (The transmission of the message should be done by the main program.) (What makes this a bit tricky is the creation of three separate tasks: one to initiate conversion, one to pick up the Voltage and initiate sending a message to the user, and the third, the primary business of the main program, to manage messages back to the user.)

4. The terminal reception of characters over the serial port is to be converted to interrupt driven. This requires modification of the "terminal process" function. Instead of being polled (the function terminal_process called from main), it is to be called by an interrupt occurring when the "RDRF" flag is set. (We've done that in class.) The interrupt service routine will either replace terminal_process or call a modified terminal process function, which in turn calls the command functions as now. Command functions should be very fast – no wait loops. They should use the printing facility of the main program to send messages back (or be buffered). If you use a buffered print, it is acceptable to use the original print function for (only) the help command.

What won't be interrupt based is sending messages back to the user (unless you use a buffered print). This will be our "lowest priority" task, operating in (or out of) the main loop. The point here is that we don't want to hold up the stepper or anything else because we are sending a message (at possibly a slow speed). So, we'd like for the actual calls to "print" to be made out of main, not the commands or interrupt service routines which will need to call something like a "requestprint" function that passes a string to be printed later. (We could use prints from those places for debugging.) Figuring out how to do this so that we have more than one pending print request can be tricky. We will need a list of the strings that commands are asking us to print. This is a "stretch" goal – do it after you have the other stuff.