Version 2 : 2.13.2006

Idan Beck (ib54)
Rohit Gupta (rg242)
Wednesday 4:30

Lab 1: Stop Watch
February 8th, 2006

Introduction

The purpose of this lab was to build a stopwatch, with a start/stop, reset, and split function. The reset button would only work if the stopwatch is stopped, and the split function will copy the current measured interval to the second line of the display without disturbing the current time displayed on the first line. The numeric display is a 2 lined, 16 character LCD. The outcome was a successful working stopwatch.

Homework

2. *Referring to the protoboard design page (and, of course, the Mega32 datasheet), describe the function of each of the components connected to pins 9, 10, 30,31,32 of the MCU.*

   Pin 9: This pin is connected to the RESET push button

   Pin 10: This pin is connected to Vcc which in our case is usually +5V.

   Pin 30: This pin is AVCC or is simply connected to Vcc if the analog digital converter isn't being used. If it is being used pin 11 should be connected to Vcc through a low pass filter.

   Pin 31: This pin is connected to ground.

   Pin 32: This pin is connected to the analog reference voltage used for the analog to digital converter.

3. *Estimate the Thevinin equvalent output resistance of an i/o pin from the Mega32 data sheet. There will be two separate estimates corresponding to whether the output is logic-high or logic-low.*

   From the DC characteristics sheet we found that the DC current per i/o pin is 40 mA. This means for an output high the Thevenin resistance would be about,

assuming a high logic of +5 V (since the high voltage is in-between 4.4V and 5.0V),

$$R = \frac{V}{I}$$

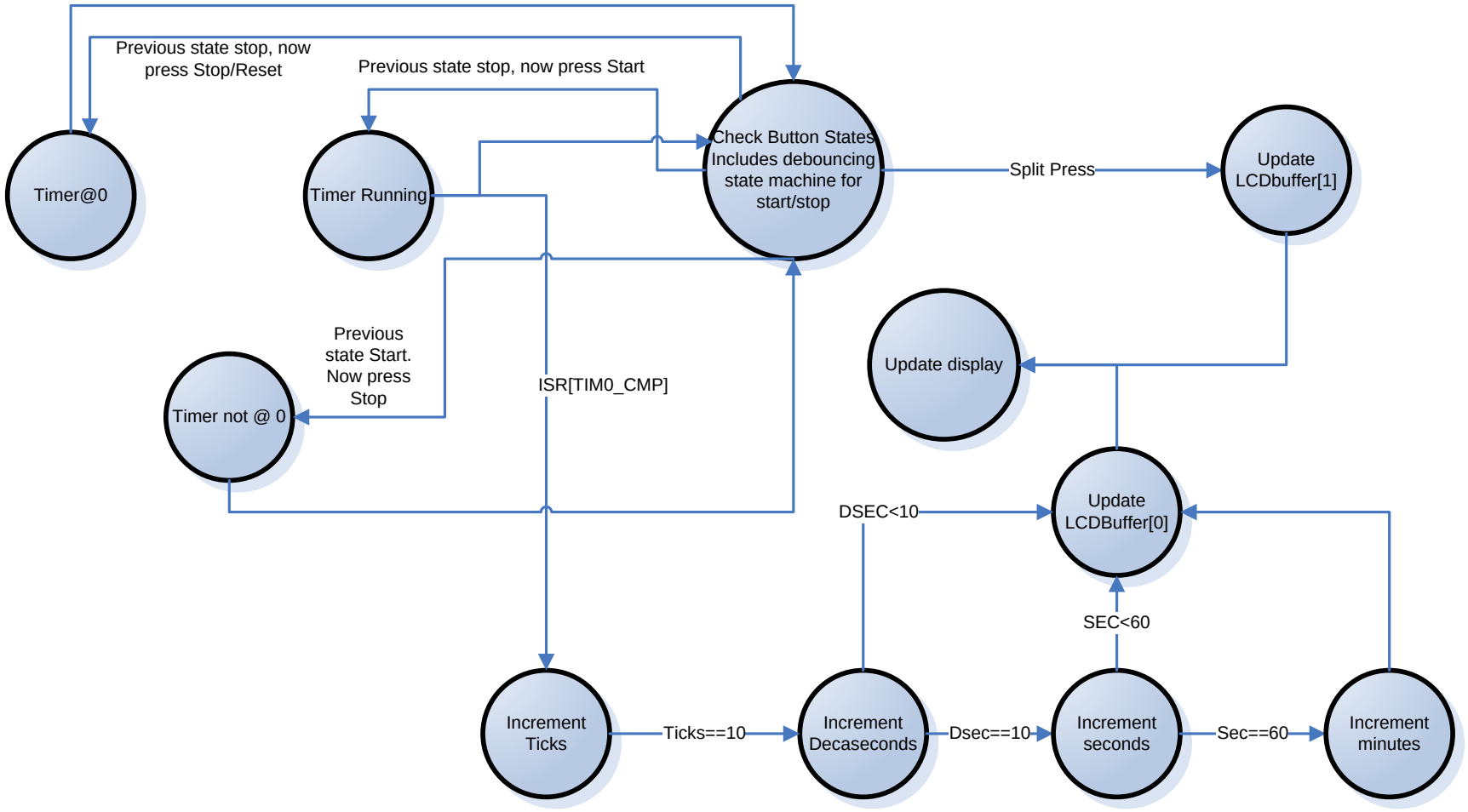
$$R = \frac{5\ V}{40\ mA} = 125\ ohms$$

And for a logic low of about .7 V, we actually have a lower out put resistance of **17.5** Ω . The logic levels will vary and thusly the resistances will as well, but this is marginal and taken into account.

4. *Look up the specifications (foward voltage and current) for a typical LED. Use these specifications to justify the choice of 300 ohms for the resistor on the lower right side of the protoboard design (revision 2).*

   From looking at the datasheet for the Panasonic: LN28RALUS the forward voltage is ~ +4V and the forward current is ~25mA. This is for a wavelength of ~660 nm or approximately red. This justifies using a 300 Ω resistor since this is a resistance that is in series with the LED. Since Vcc is tied to +9-12 V is too much voltage and would blow out the LED. The 300 ohm resistor produces a voltage divider which dissipates 65% of the voltage giving us a voltage range of 3.2V – 4.2V. This allows the LED to operate without burning out.

5.


Previous state stop, now press Stop/Reset
Previous state stop, now press Start
Timer@0
Timer Running
Check Button States Includes debouncing state machine for start/stop
Split Press
Update LCDbuffer[1]
Previous state Start. Now press Stop
Timer not @ 0
ISR[TIM0_CMP]
Update display
DSEC<10
Update LCDBuffer[0]
SEC<60
Increment Ticks
Ticks==10
Increment Decaseconds
Dsec==10
Increment seconds
Sec==60
Increment minutes

Design and Test

Initial Setup:

Hardware

This lab had very simple hardware interface which consisted of an LCD display that was wired up to port C which we interfaced with using the LCD library functions. The buttons were wired up to port D which passed logic lows to the CPU when they were pushed. Also later we implemented a tone generator for every button pressed, using the 8 bit timer2 on the CPU, using a piezo driven speaker that was hooked up to port B.0 which was toggled every time the timer2 compare interrupt was thrown. This produced a 1.4 V peak to peak signal in the speaker that produced a tone.

Software

We wrote most of the code before we got to lab. The code was written to comply with all the specifications stated on the assignment sheet. After getting all the basics working we also implemented a tone generator that produced a different tone for each button pressed. This was implemented using the timer2 TIM2_CMP interrupt service routine. The frequency of the tone was altered for each button press by changing the OCR2 register so that port B.0 was toggled more frequently or less frequently.

Design

Design of the Hardware

No design of the hardware or hardware interface was needed here except for the actual construction of the protoboard on which the LCD display was mounted.

Design of the Software

The software was designed in very close compliance with the state machine. Essentially when the CPU is powered on and boots the program initializes all the proper variables and registers to the proper values. Timer0 and Timer2 are turned on and their TCCR registers are set.

Timer0 was used for the actual timing algorithms in the stop watch and so it was set to a divide by 64 prescalar. This meant that if the OCR0 register was set to the value of 250 the timer0 compare interrupt service routine was called a thousand times a second. We used this to set up a variable tick which was initially set to 0. This variable was incremented by one every time the ISR was called (only if the timer was “on” which is explained more later). The value of ticks controlled all the timing calculations such that after 100 ticks the tick variable was reset to 0 and the decaSeconds variable was incremented by one. Similar calculations were performed for seconds and minutes (e.g. 10 decaSeconds → +1 second ; 60 seconds → +1 minute). Also every time a decaSecond was incremented the LCD string buffer for the first line was updated and the LCD display was updated as well. This allowed us to only update the

LCD display and buffer when we knew something had changed and saved precious CPU cycles which could be crucial in a precision timing program.

Timer2 was used for the purpose of generating tones every time the buttons were pressed. This timer's TCCR was set to a prescalar where we were happy with the frequency that it output. The tone signal was generated by setting the OCR2 to a value where each time the compare ISR was called port B.0 was toggled between 0 and 1. This output a .7V peak to peak square wave. Since no real frequency specifications were given we played around with the prescalar until the base frequency (when OCR2 = 200) sounded pleasing to the ear. To alter the tone for the different button presses we used different OCR2 values and only toggled when a button press was flagged (for button2 this was not the same as finding the logic level on that bit to be low since this button required a debouncer).

The rest of the initializer simply sets up the LCD display and all the variables such that the timer is initialized at time 0. The last thing the program does before entering the main program loop is initialize both the LCD buffer strings to "0:00.0" and update the LCD display to display the LCD buffer since we want to know that when we turn on the stop watch that it is actually on. Not only is this a good debug function but also it is a general specification expected in a user interface.

At this point our program was pretty much ready to operate other than the button checking algorithm. This was a simply algorithm. We set up a buttonFlag array of three unsigned chars globally so that at any point the program could simply index this array and find out what that specific button's state was. For the first two buttons this was simply done by checking the logic level of their respective bits. For the third button (button2) we had to implement a debouncer to eliminate the start/stop toggle variable to toggle an arbitrary number of times and unreliable timing results. All of this was done in task3() while the update() function was used to update the state of the program variables (and indirectly the state of the machine) every time the appropriate button was pressed. For sake of simplicity the start/stop functionality was hard coded into the debouncer and utilizing a two generation memory could guarantee that the start/stop button would not double bounce and maintain its value even if the user held the button down.

The start/stop button controlled a variable named started which if it's value was 0 the timer stopped and the ISR would not increment the tick variable, if it was 1 then the timer would be running and the ISR would increment the tick variable. The reset button would set the started variable to 0 and reset all the timing variables but would only be functional if the started was at 0 meaning the timer was stopped. The split time would simply update the LCD buffer for the second line and the LCD display was updated whenever the ticks were incremented anyways so there was no need to call this function twice. The way we implemented the split time button was that the elapsed time was recorded when the button was released and while the button was pressed the elapsed time display ran along side with the timer.

Testing, Hurdles and Solutions

(note, the software explained above was the final software)

When we first tested our raw code we found that the timer was working correctly except it was running at a very fast rate. This was rectified by changing the value of a decaSecond from 10 ticks to 100 ticks (a simple define at the top of the code done to anticipate any such timing calculation mistakes). The next problem we noticed was that the start/stop button was skipping and not debouncing properly. We utilized a two generation history variable to record the previous state to when the button was pushed such that we could tell when the button changed state and used this change to toggle the button flag rather than use only the state machine. This allowed the user to hold down the button and the timer would not perform any erratic functions. We also had to alter the task3() buttonFlag routine timing to make sure that the program was checking the button states at a proper interval since even using this two generation variable it was possible to get defunct results with intervals that were too large or too small.

After we had all of these things working we went on to implement the signal generator for the speaker that was a different frequency for each button press. This was all done using the timer2 compare ISR.

Questions

1) To measure the time-base accuracy we could toggle a port bit on the CPU and plug it into an oscilloscope and see how accurately the ISR is being called to the value we assume it to be (a thousandth of a second). The same could be done for the decaSecond, second, or minute variables where every time they are incremented a port bit is toggled, the resulting waveform them put into an oscilloscope and the error found.

2) Wait-loops are software based and are not very accurate since the other code in the loop will alter the refresh rate of the wait-loop. Also an interrupt will halt the CPU and run the ISR code right away while a software based wait-loop will wait until the CPU is done doing its current function before it can run it's code.

3) As described above we used a polling loop at a certain interval to detect the pushbutton states for all the of the push buttons.

4/5) design aspects are listed above, code is attached in the appendix.

Conclusions and observations:

This lab was successful in that our code worked mostly when we got to lab except for that the start/stop button seemed to be delayed (initially it seemed it was a debouncing error). After looking over our code again, we noticed that we were not saving the previous state of the stop watch properly, and a quick fix made it work. We feel that we could have created a more pleasing sound to each button rather than have a single frequency being produced for the extra credit. We also think this lab could be

expanded by having a LCD "power on" button (possibly the first time the start button is pushed), and an LCD automatic turn-off if the stopwatch is stopped and has not been used for over a minute.