I/O Ports, Timers and Interrupts

Jesus Luciano #: 016354921

Rosswell Tiongco #: 016091762

CECS 262 - Intro to Embedded Systems Programming

October 31, 2017

Table of contents

Table of Figures

Introduction

This purpose of this project was to implements the usage of I/O ports, internal hardware timers and external interrupts to create a 4 function, 4 speed output to 8 LEDs. The modes of the project only changed when an external button was pressed.

The 4 modes of this project started with mode 1, a bouncing output where the LED chosen to be on would shift back and forward across an 8 LED output. Mode 2 was a 4 bit counter that would output to the 4 rightmost LEDs. An external switch determined whether the counter would increase or decrease. Mode 3 was a double bouncing mode where the two outmost LEDs would shift inward until they reached each other, then would shift outward until they reach the out most LEDs, then repeat. Mode 4 was a stack mode where the leftmost LED turned on, then the leftmost 2 turned on, then leftmost 3 until all LEDs turned on. Once all LED's were on, it would decrease the number of LEDs on in reverse order and repeat.

There would be 4 speeds that LEDs would turn on. From smallest to greatest, the times were 0.1 seconds, 0.5 seconds, 1 second, 2 seconds. Changes in speed were independent from being set by an external button. The system detected changes in speed immediately.

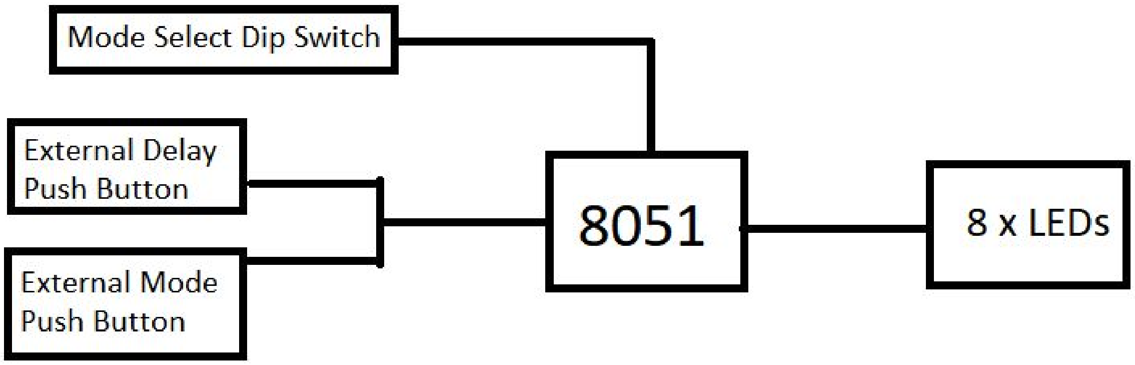
Operation

This project was built upon the use of an 8051 microcontroller connected to a board with 8 dip switches and 8 LEDs. The project used 18 of the 32 I/O ports of the 8051. Once our code was sent to the board, the 8051 would infinitely repeat the functions of the code until power was shut off or the CPU was reset. This was due to the software implementation of a “super-loop” where an infinite loop would be purposefully used execute itself continuously.

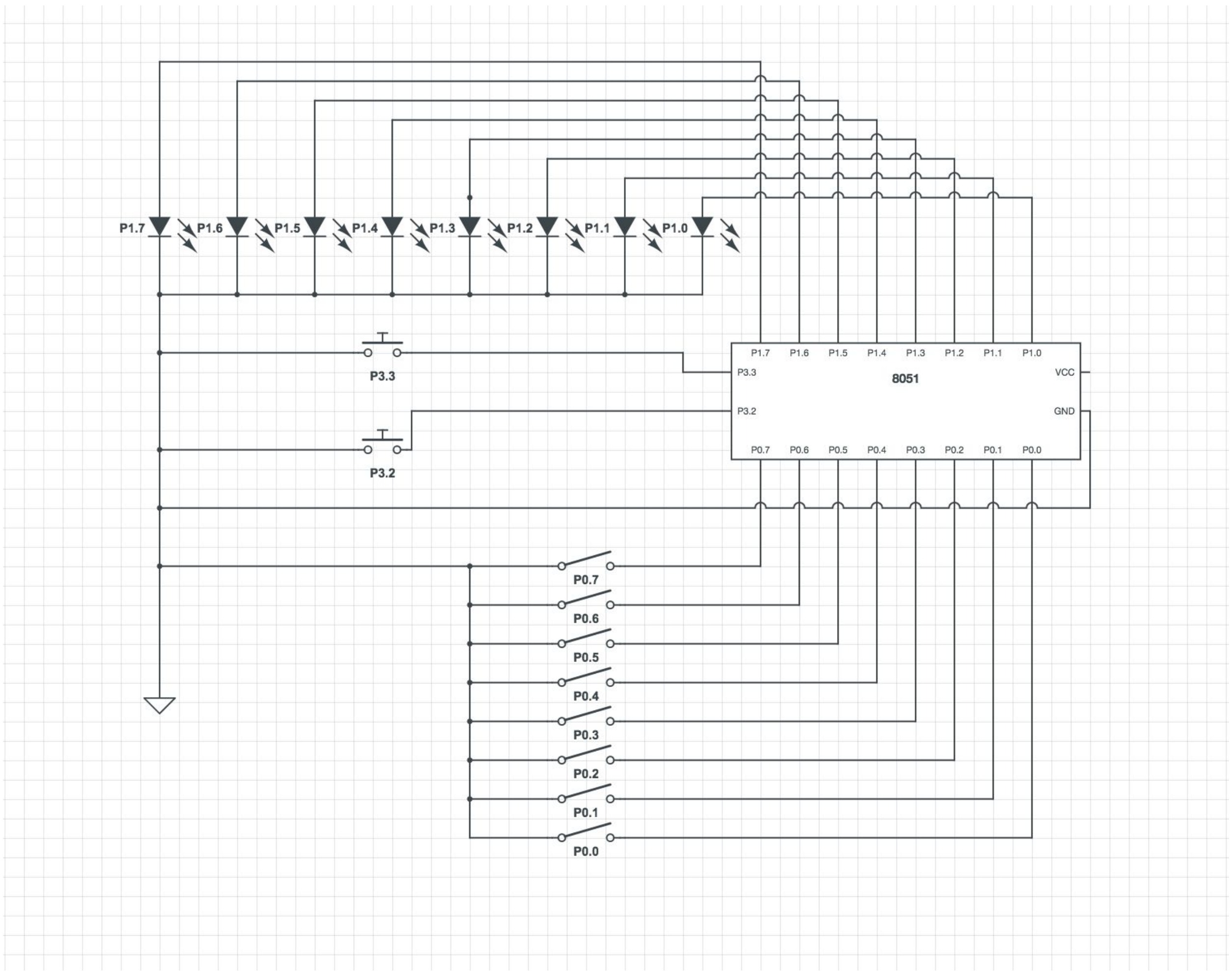
When the 8051 is initially turned on, the chip will default its output mode to mode 0 to the onboard LEDs. The board will read P0.3 and P0.2 to find the amount of time delay between each change in LED output. The board will continuously run in mode 0 until both the mode switches, P0.1 and P0.0, are changed and the external button tied to P3.2 is pressed. Pressing the button will then change the current mode to the mode set by P0.1 and P0.0. If the mode is the same, there current mode will not restart and just continue as if the button were never pressed. If the mode is changed to mode 1, it will read P0.7, P0.6, P0.5 and P0.4 and load their values onto the rightmost LEDs and increase the value by default until it reached its max value and resets to 0. If the button tied to P3.3 is pressed, the value sent to the LEDs will be decreased until it reaches 0 and is reset to 15. Mode 2 and 3 will output to the LEDs until the mode is changed.

Hardware

Hardware block diagram



Schematics


P1.7
P1.6
P1.5
P1.4
P1.3
P1.2
P1.1
P1.0
P3.3
P3.2
8051
VCC
GND
P0.7
P0.6
P0.5
P0.4
P0.3
P0.2
P0.1
P0.0

Hardware

1. 8051 Microcontroller and its trainer board
    a. 8 x LED’s
    b. 1 x 8 dip switch
2. 2 x Push buttons
3. Breadboard

Hardware Justifications

The 8051 microcontroller provides the computational power to repeatedly execute instructions.The microcontroller itself was embedded into the trainer board. Alongside the microcontroller, the trainer board hosts light emitting diodes (LED’s) and a dip switch.

To externally interact with the microcontroller, two push buttons are used as two separate external interrupts. One of the buttons changes the mode, the other button changes whether the counting mode increments or decrements.

Software

Software approach

The project was based on an 8051 microcontroller and utilized internal timers, internal and external interrupts and input and output ports.

This project made use of 3 of the 6 interrupts available on the 8051 chip. The interrupts that were used were external interrupts 0 and 1, and timer 1. The external interrupts were tied to buttons which connected P3.3 and P3.2 to the board's GND. This was done because Port 3 has internal pull up resistors that tie it to VCC by default. This meant that P3.3 and P3.2 had a default value of logical 1. In order to activate the interrupt, the port value had to change to 0, which is what the button would do when it was pressed.

Timer0 was used initially but an issue arose when both external interrupts were called and they would not be executed if the buttons were pressed and released in the middle of a delay. The button would have to be held down longer than the delay, or it would register that it was pressed. To resolve this, timer1 was used because it had less priority than the external interrupts. This would ensure that our external interrupts would be called even in the middle of our delay.

The delay function of the board made use of the internal timer1 interrupt. Timer1 was set to a 16 bit count mode. Since the board's clock ran at 11.0592MHz and it would take 12 clock cycles to increase the timer by 1, each increase in the counter would take a total of 1.0825us. The maximum value of a 16 bit register is 65,536. So the maximum time our counter could count to in one full iteration is 71.1ms. Since the fastest delay our project implemented was 100ms, we decided to round down to a value that would be a multiple of the delay speed we needed. We chose 50ms. In order to delay 50ms, our counter would need to count to 46,189. Converted to hex, this value is 0xB46D. Since we only need to count up to this number, we subtracted it from 0xFFFF to yield 0x4B92, the start value of our timer.

Since the 8051 used 2 8 bit registers to count, the high byte was set with the value 0x4B and the low byte was set with the value 0x92. This would ensure our timer would only count up with the delay time that we needed. Everytime the timer reached its max value, it would overflow and call its interrupt function. We exploited this recurring function call to loop our timer a set number of times until it delayed our desired time. At every change in our output in our functions, our code would call a delay function. Our desired delay, based on P0.3 and P0.2's values shown in **<Fig1>**, would set a variable to the number of times we wanted to repeat a 50ms delay. Once the variable was set, the delay function would enter a while loop that would loop until our variable was equal to zero. Every time the timer interrupt was called, it would subtract 1 from our variable. The net result was looping a 50ms delay multiple times to achieve a desired delay.

P0.3	**P0.2**	**Delay Speed**
0	0	0.1 Seconds
0	1	0.5 Seconds
1	0	1.0 Second
1	1	2.0 Second

Fig1 - Delay Speed Selector

Mode selection on the 8051 is based on inputs from P0.0 and P0.1. The default mode loaded when the board is turned on is mode 0, the bouncing mode. Since this project was interrupt based, our mode only changed when the interrupt function tied to P3.2. In order to accomplish this, we used 2 variables, varM1 and varM0, to represent the mode switches. The default mode value for both of these variables is 0, therefore, the default mode loaded when the board was turned on was mode 0, the bouncing mode. Whenever P3.2 was set to low, the interrupt function would set the values of P0.1 and P0.0 to varM1 and varM0, respectively. This caused the effect of having our mode only change when we pressed a button. The different modes the board had are listed in **<Fig2>** .

P0.1	**P0.0**	**Mode**
0	0	Bouncing Mode
0	1	Counting Mode
1	0	Double Bouncing Mode
1	1	Stack Mode

Fig2 - Mode selector

In mode 1, our output P1 was initially set to 0x80, meaning our leftmost LED was set on. A counter was also initialized to zero. Our code would check the value of our counter and divide the value of P1 by 2, or essentially shift it to the right. It would then increase our counter and then call our delay function. It would check our counter until it was done 7 times. On the 8th iteration, P1 would be multiplied by 2, or essentially shift itself to the right. This would also be done 7 times. Once our counter reached a value of 15, it would be reset to zero and the function repeated until our interrupt changed our mode variables. The net effect of this mode would be that a single LED would shift back and forth on our display of 8 LEDs.

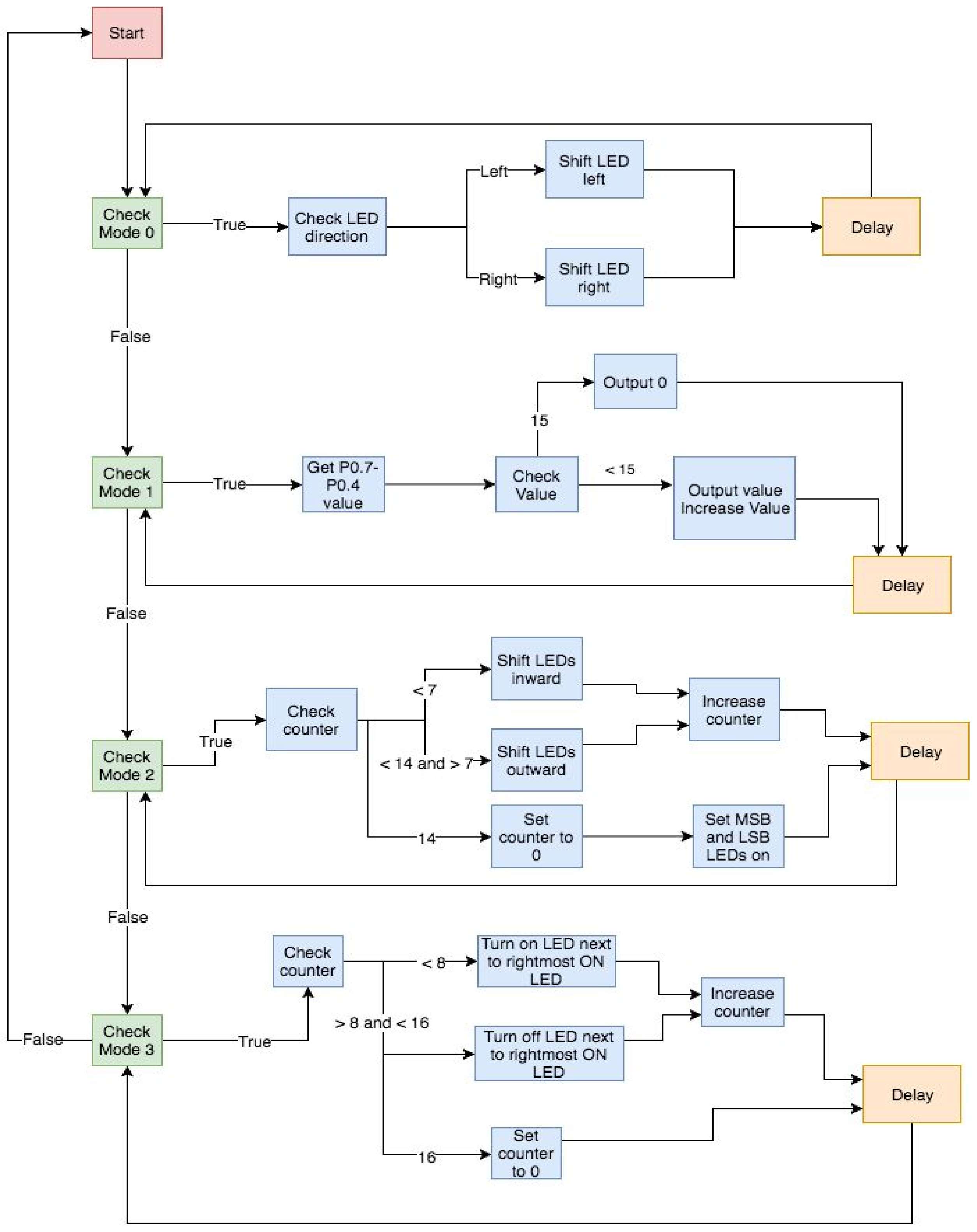
Mode 2's function was to implement a 4 bit counter with its initial value set to P0.7-P0.4. This initial value of P1 was set by dividing P0 by 16, or shifting it to the right 4 bits. This value is then sent to P0. An external variable, direction, was set to a positive 1 in the preprocessor directives. The code then enters a while loop where it adds our direction value until P1 is equal to 16. Once its value reaches 16, it is reset to 0 since we are only counting using 4 bits.

Our counter changes counting direction when P3.3 is set low using our external buttons. In our ISR, the value of our variable, direction, is multiplied by negative 1. This essentially negates the sign. Once our value is set, it is added to the value of P1. This produces the effect of counting down with 4 bits. Whenever the value reaches -1, it is reset to 15, since we cannot represent negative values with our LEDs.

Mode 3 required the LEDs to double bounce. Starting with the outermost LEDS lit up, the LEDs came to light up one at a time while keeping all other LEDs off until the two LEDs met at the center. After meeting, the LEDs bounce in the outer direction. As with all the other modes, they can be achieved by numerous different approaches. Although there is a more simple logical implementation than what was used in our code, we decided to keep it to showcase an additional way of implementing this logic. To start, we initialized P1 to 0x81. In each incrementation, two separate would get shifted to opposite directions. If x was initialized as 0x80, a right shift would be used. Within the same loop, P1 would get var | var.

Mode 4 called for the LEDs to gradually light up each LED while keeping the previous LED lit up. After all the LEDs are lit up, the LEDs turn off one at a time and the process repeats. To implement this logic, a dummy variable was initialized to hold the values of 0x80. The variable would then get shifted to the right and get exclusive OR'd (XOR) with P1. Doing this would gradually flicker all the LEDs. After all the LEDs were lit up, the process would repeat, except the dummy variable would shift to the left. Since XOR is used, the previously lit LED would turn off.

Software Flow Diagram



Conclusion

The project focusing on programming I/O ports, interrupts, and timers was successfully completed. Furthermore, the lab cemented the idea that finishing early is highly beneficial due to reducing anxiety and allowing additional time in case questions regarding the project were formulated before the submission of the project. Within the code, implementing the timer turned out very well as the timer register values we used proved to be closer to 50ms than the values provided by the professor. Finally, as the first partner project, version control was utilized.

The project taught lessons through failure. The main culprits for us initially failing were in variable declaration and scope, as well as timing and interrupts. Since different variables were not created for different loops, debugging the program proved harder than necessary. Furthermore, since Keil can only replicate the target machine so much, we neglected to focus on the physical oscillator's timing. Instead we focused on testing the delay through the simulation. The final issue we had was discerning why our interrupt timer was buggy and uncooperative. This issue was resolved by consulting the interrupt priority register.