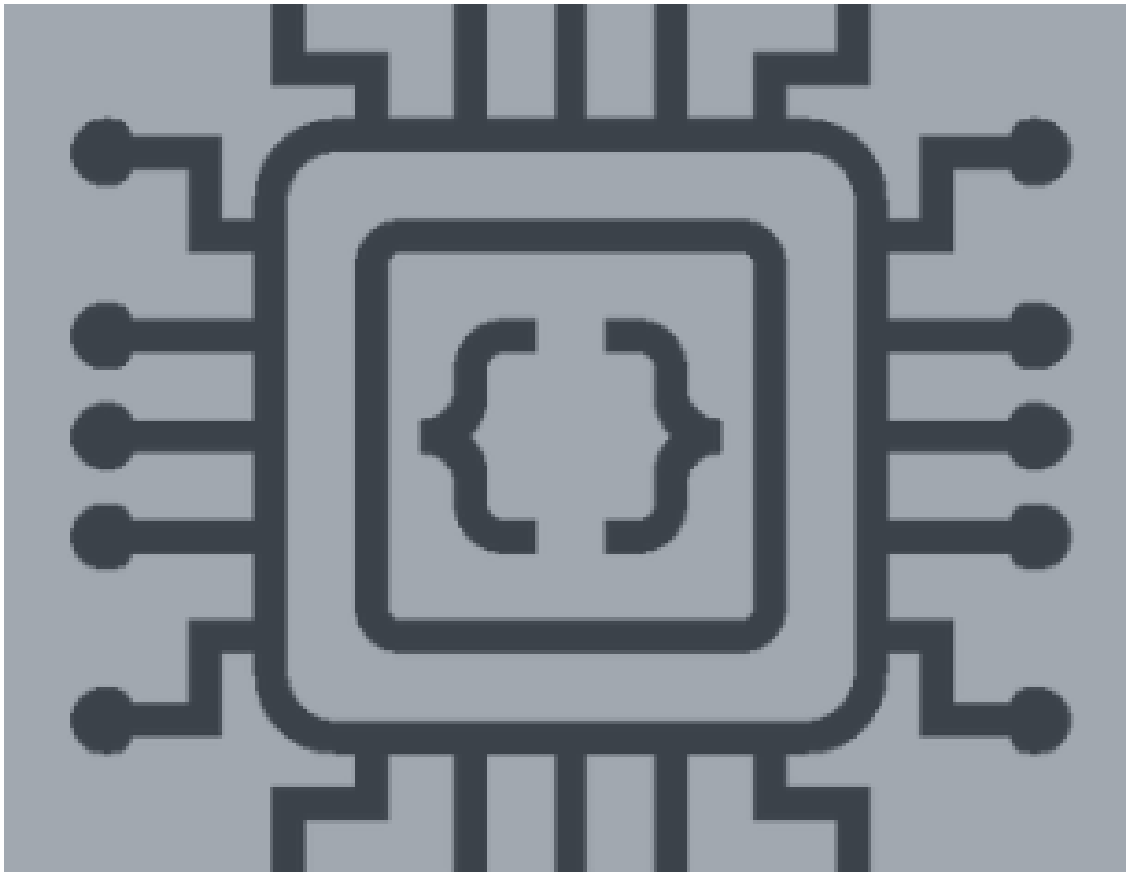
Embedded Systems



Assignment: 5
Date: 29/04/2023
Group: Johnson Domacasse, Iskren Zhechev
Version: V1

Contents

Abstract: 3
Introduction: 3
Procedures: 3
Hardware setup: 3
Software setup: 4
Assignment A: 5
Plan for code: 5
Design: 6
Code: 6
Assignment B: 8
Plan for code: 8
Design: 8
Code: 8
Assignment C: 10
Plan for code: 10
Design: 11
Code: 11
Notable anomaly: 12
Conclusion: 13
References: 13

Abstract:

Interrupts and timers are powerful mechanics within the Arduino. They are both used for real-time I/O programming. The reason for the usage of interrupts is so that the processor can respond quickly to important events. The usage of timer is so that we can execute code without having to disturb the rest of the code. The Millis function is an example of a function that utilizes timers in the Arduino. In conclusion they make the program quicker, more efficient and more reactive.

Introduction:

The essence of this practicum was to teach us about two important mechanisms within a processor. These mechanisms are used for real-time I/O handling. Namely interrupts and timers. This practicum is about low-level programming of interrupts and timers on the Arduino using hardware registers. Our first two experiments, Part A and B, will be about working with the interrupts. Part C will be about working with timers. This practicum enabled us to create real-time software.

The conclusion that we have pulled from these experiments is that interrupts and timers are indeed powerful mechanisms within the Arduino if utilized properly. At our current level, I would say that we don't have knowledge to make the full usability of the timers and interrupts. Interrupts can be used so that important events will not be missed by the processor and the timers are used as a sort of what we know as the Millis function.

Procedures:

Hardware setup:

The following hardware was used in order to pull off these experiments:

- A laptop + VS Code + Platform IO
- 1 x Arduino + USB cable
- 1 x breadboard
- 2 x buttons
- 2 x LEDs
- 2 x 10k Ohm resistors
- 2 x 220-470 Ohm resistors
- Jumper wires.

Software setup:

There were a few things we have to consider in our implementation. Along with these, some additional tips.

- We need to get most of the information from the Atmega328p datasheets.
- This is a low-level programming assignment. In other words, we aren't allowed to use pinMode(), digitalRead() and digitalWrite(). Instead we use the hardware registers like PINB, PORTB, DDRB etc.
- For setting and clearing bits in the hardware registers, we can use the macro "_BV" and symbolic names such as PINB0, PIND2, PORTB1, etc. For example _BV(PINB3) is the flag for pin PB3 in the PINB register. The symbolic names are defined in the header file "iom328p.h".
- We must try not to use symbolic names as much as we can, i.e. we must try not to use hardcoded values for flags, masks and register manipulations. We must make our program compatible for Arduinos with a similar processor architecture and not specifically for the Uno.
- We are allowed to use the Arduino library operations Serial.begin() and Serial.print().
- We must use the setup() to initialize the microcontroller, e.g. pins and modes.
- We must use the loop() to perform the I/O actions.
- For Assignment A we will not use interrupt handling.

Assignment A:

The essence of this assignment is to build the system described within the practicum report. Use low-level programming to get it all working. Document our findings. Pull conclusions from what we have experienced.

Plan for code:

We must consider two figures before we begin with programming. These figures depict on how we need to wire the entire system. See figure 1 and figure 2 for the depiction of the buttons and LEDs. See figure 3 for the complete system.

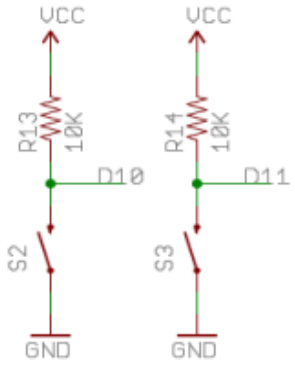


Figure 1: Buttons D10 and D11 diagram

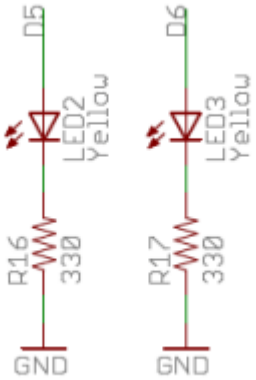


Figure 2: LEDs D5 and D6 diagram.

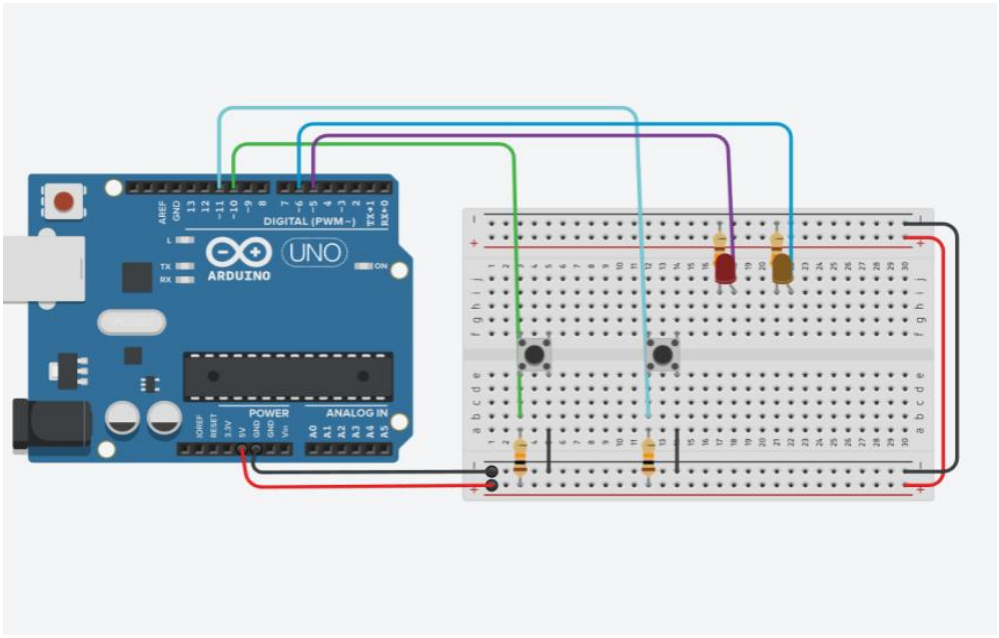


Figure 3: Complete system connection diagram.

You may have noticed that in figure 3 we connect the data wires (blue, pink, purple and green) to these 4 specific pins. For example take the button connected to pin D10. D10 stands for Arduino digital pin 10.

The second step towards making this part of the practicum involved a fair bit of reading. We read chapter “14 I/O ports” from the ATmega328/P datasheets. Indeed we were overwhelmed, but when we started coding the assignment it became easier.

The last step before building our project, required us to look at the requirements. These requirements can be found later in the code. They are the following:

- When the user presses only button D10 then LED D5 turns on (and turns off when the button is released).
- When the user presses only button D11 then LED D6 turns on (and turns off when the button is released). During the press, the string "Hello World!\n" is repeatedly sent over the serial line and stops when the button is released.
- When both buttons D10 and D11 are pressed then both LEDs are blinking with alternating pattern repeatedly every 0.5 seconds
- Use a debouncing solution. It is not allowed to use the debounce library routines.
- When no button is pressed, nothing happens.

Design:

For our implementation, we are thinking about using a switch case to handle all of the different states that the system can be in. one for when button 1 is pressed. One for when button 2 is pressed. One for when both buttons are pressed. Finally, one for when no buttons are pressed. See figure 4 for the state diagram of the implementation.

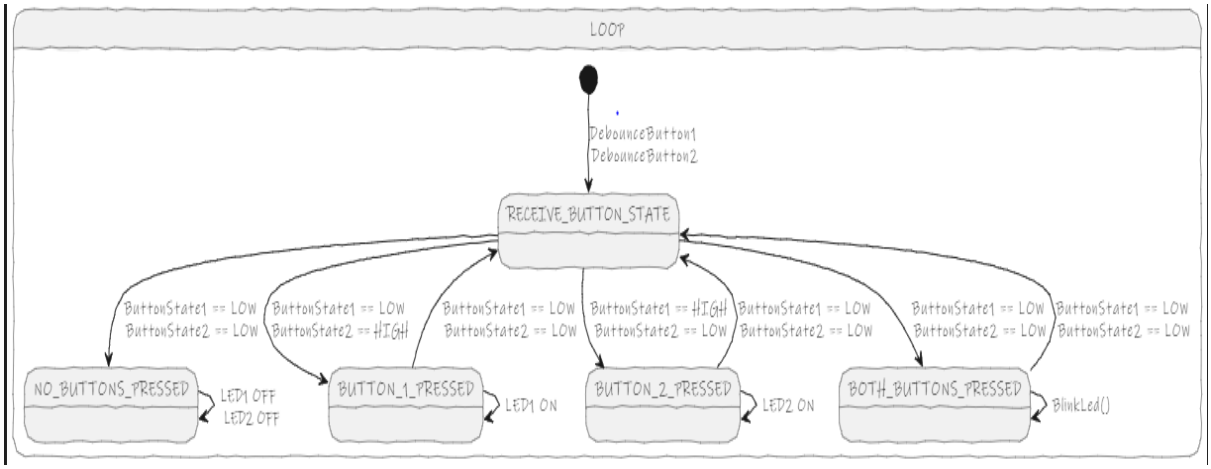


Figure 4: State diagram A

Code:

Now that we have everything we can get started on our implementation. See the code implementation as you read through this report.

We begin by including some libraries to for functionality of this project. These libraries are the Arduino.h library and the avr/io.h file.

We then set up our variables. First, some constants for the pins. Namely the BUTTON_1_PIN, BUTTON_2_PIN, LED_1_PIN and LED_2_pin. We then define the masks for each of the components. Notice how we use to _BV function to shift the register pins and assign them to their respective masks. We then set up some variables for the debouncing function that we implemented. Mainly, the state of the button and the last state of it. Along with this, sum variables for the Millis function implemented in the debounce function. Finally, we then set up the variable that handles the Millis function within our loop function.

We then set up the enumeration, “ButtonState”, to signify the different states within the system. These are the four states mentioned in the design section mention in this assignment.

After this, we began implementing some functions that make this implementation more modular. These functions are the button debounce function and the handle button press function.

The debounce function is self explanatory. It takes the button number, the debounce time, the last button state and the button mask as parameters. The only functionality of the function is to debounce the button and then return the current state of the button. i.e. No LEDs will be turned on or off within the function.

The handle button press function is to handle the button pressed by toggling the bit in the PORTD register that corresponds to the designated LED pin. The function accepts the LED pin as a parameter. Simply put, all the function does is turn the LED on if the correct button is pressed.

We can now work on the setup and loop function now that all of our other functions are implemented. We will begin with the setup.

In the setup function, we use the DDRB register to set our buttons as an input for this implementation. We then use the DDRD register for using the LEDs as outputs. This is done by basically using the OR operator in conjunction with the LED pin.

Finally we have our loop function. This contains the main part of our implementation. First we assign the Millis time to the current time variable. We then get both the button states by using the debounce function twice. One for each button. We then make use of an enumeration variable and assign the current button state to it. Finally we hop in our switch case. See figure 4 in the design section. We use the enumeration variable to jump from case to case. In the case of button 1 is pressed, we use the handle button press function to turn on LED 1 (and then off when button is released).
In the case of button 2 Is pressed, we use the handle button press function to turn on LED 2 (and then off when button is released). Additionally, we continuously print “hello world” to the serial monitor. In the case of both buttons are pressed, we toggle each LED on and off based on the Millis timer. Finally in our default case, we turn all LEDs off. This is because no buttons are pressed.

Assignment B:

The essence of this assignment is to build the system described within the practicum report. We use low-level programming to get it all working here as well. Document our findings. Pull conclusions from what we have experienced. Additionally, we are using interrupts this time around instead of a switch case. More will be explained below.

Plan for code:

The plan for the code in this part remains more or less the same as the plan for the code in part A. We will be using the same systems built (see figure 3) to test our implementation. The difference here is the chapter we have to read. For this part of the assignment, we will read chapter “12. Interrupts” and chapter “13.EXTINT – Eternal interrupts in the ATmega328/P datasheets. Finally, here are the requirements for this project are the following:

- Each button must be handled by the correct interrupt service routine and make sure that the LEDs change state when the button is pressed on the right edge only.
- Do not use “expensive functions” in your interrupt service routines, i.e. an interrupt service routine (ISR) must be quick and short at all times.
- As you know, buttons are nasty things because they cause a bouncing effect. You will need to debounce your buttons by yourself.
- It is not allowed to use the debounce library routines.
- It is not allowed to use digitalwrite(), digitalread(), pinmode(), attachInterrupt() or other library functions that handle GPIO.

Additionally we will be making two separate projects for this part. One with and the other without debouncing to see what the difference may be. For the sake of this report being clean, we will talk about the implementation without the debounce.

Design:

For the design of this system we were thinking of implementing something similar to what we did for part A. The only difference is we will not be using a switch case, but the isr() function. Within the function you will have situations, if the each button is pressed respectively, when no buttons are pressed and when both buttons are pressed. See figure 5 for the state diagram of this implementation.

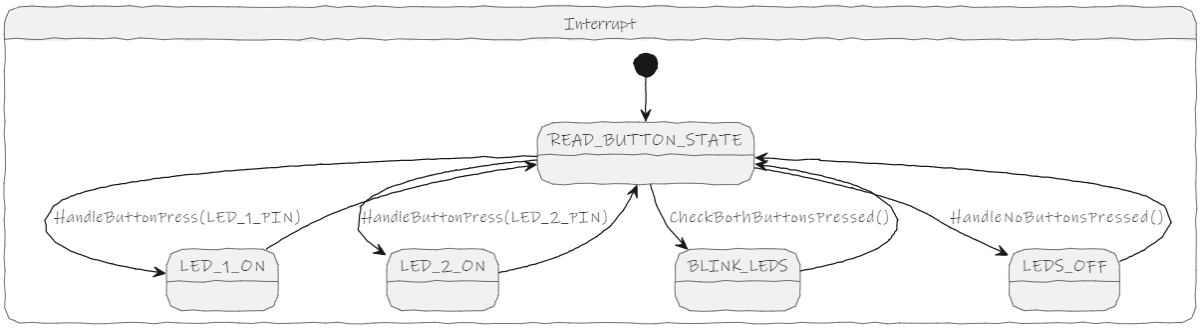


Figure 5: State diagram B

Code:

Now that we have everything we can get started on our implementation. See the code implementation as you read through this report.

We begin by including some libraries to for functionality of this project. These libraries are the Arduino.h library, the avr/io.h file and the avr/interrupt.h file.

We then set up our variables. First, some constants for the pins. Namely the BUTTON_1_PIN, BUTTON_2_PIN, LED_1_PIN and LED_2_pin. We then define the masks for each of the components. Notice how here we use to _BV function to shift the register pins and assign them to their respective masks as well. We then make some Boolean variables to hold the previous state of the buttons but also a Boolean to check if both buttons are pressed. Notice how they are all declared as volatile. The reason is explained in the conclusion section. Finally, We set up the previous Millis timers for each of the three states where one or both buttons are pressed. These are also declared as volatile. We included a constant to act as a delay for the Millis as well.

Now we can work on the functions we will be implementing in this project. We make use of the following three custom functions in this implementation:

- **Handle button press function**: The purpose of this function is to check whether a button has been pressed. The function will turn on the correct LED based on the buttons that is pressed. It will turn the LED off as well if the button is released. The parameters that this function accepts are: int Ledpin, volatile unsigned long &previousMillis and volatile bool &prevButtonstate.
- **handle both buttons pressed function:** The purpose of this function is to check whether both buttons are pressed or not. It takes the Boolean variable declared previously and sets it to HIGH (true) if both buttons are pressed. Else it will just assign the value LOW (false) to the variable. Later in the loop function, we use the variable to determine if we should use the blinking function.
- **Handle no buttons pressed function:** The purpose of this function is to turn off all of the LEDs and set the previous button states back to HIGH.
- **Blink LEDs function:** The purpose of this function is to toggle each LED on and off, in an alternation pattern, based on the constant interval timer we used in the Millis.

We can begin working on the interrupt now that everything else has been handled. Small detail to mention before we begin is that we have to include the sei() function to enable the global interrupts we will later use. The interrupt function (ISR()) will act as our “switch case” for this implementation.

We first check if the button 1 is pressed AND if the previous state of that button is HIGH. If this condition is true we use the handle button function with the parameters: LED_1_PIN, the previous Millis of 1 and the previous button state of 1. We do this twice. Once for each button. Lastly, we call both the function of handling both buttons being pressed and the function of handling o buttons being pressed.

The loop function looks a bit different. Since we are working with interrupts, we don’t really have that much need to work with the loop function. However, as previously stated, we use a variable to check if both buttons are pressed. If that is the case then the blinking function will be called.

The setup functions also looks different. We first set up the LEDs as outputs and the buttons as inputs. We then set up the PCMSK0 variable with the interrupts on both the button pins. Lastly as mentioned before, we need to call the sei() function to enable global interrupts.

Assignment C:

In this part of the assignment we were tasked to combine the knowledge we acquired using interrupts with timers. The task is to build an 8-bit counter using both the timers and interrupts. So we add 2 more LEDs. See figure 6. The buttons will be used to make the counter go faster and slower.

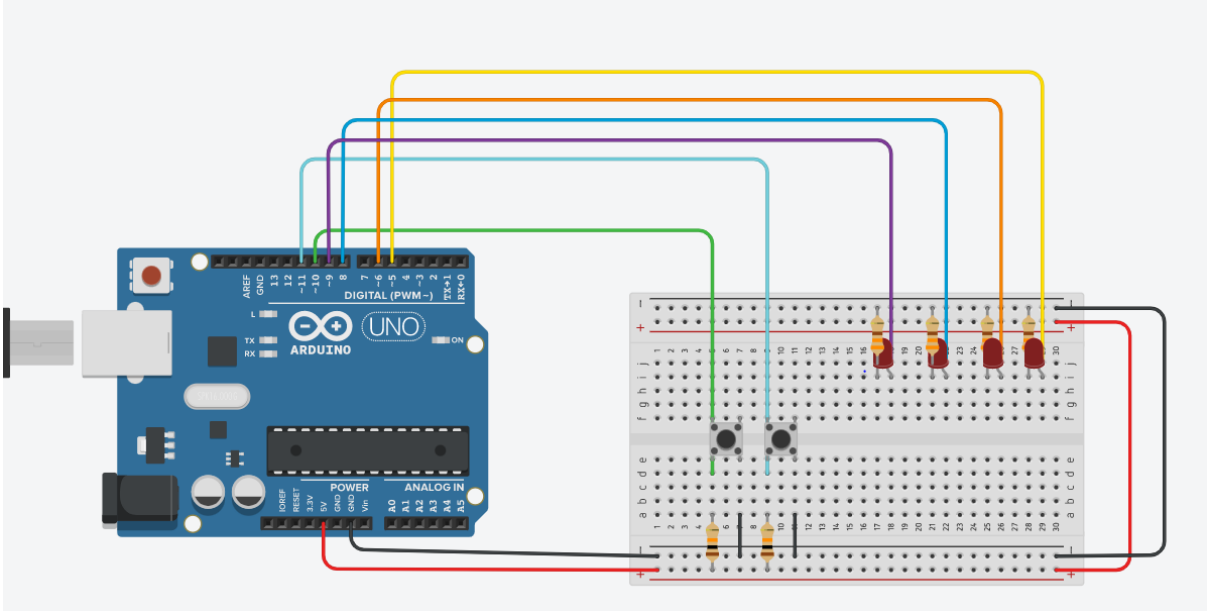


Figure 6: complete system, including 2 extra LEDs.

Plan for code:

You may notice that we reuse a good amount of code of part B for this part. In the code section we will explain how we achieved every aspect of this part. This is done after we gave an explanation of how our code is set in place. Like the previous one, we have a design in place for how we wanted to tackle the problem in the design section. See figure 7. The requirements that need to be met are the following:

1. Timer 1 runs in CTC mode.
2. Timer 1 has a pre-scaler of 64
3. The Compare Match A interrupt routine displays a counter (4-bits binary number), by the four LEDs (0 = off, 1 = on). The speed by which the number is displayed is default approx. 4 times per second.
4. The LEDs are connected as follows: Bit3: D9, Bit2: D8, Bit 1: D6, Bit 0: D5. With four bits we can count from 0x0 till 0xF Hex (0 till 15 decimal).
5. Pressing button D10 yields an interrupt; the interrupt routine slows the update speed of the binary numbers down.
6. Pressing button D11 yields an interrupt; the interrupt routine speeds up the update speed of the binary numbers.
7. The timer 1 interrupt routine must also turn on and turn off the built-in LED at a rate of approx. 1x per second. (we may use an external LED in this section)
8. Things that run at specific frequencies should not be influenced by delays in the loop().

Now that we have every requirement we can start working on the design we want to implement. Once we have an idea, we can get started on the code. Throughout the code

section I will annotate where each requirement falls in.

Design:

For this design, we will implement something similar to what we have in part B. Difference is now we have to keep the timer in mind. This is also another interrupt on its own we feel. We know we need one to handle to counter (timer interrupt) and one for the buttons. So based off of these we went to work and came up with the implementation you see here.

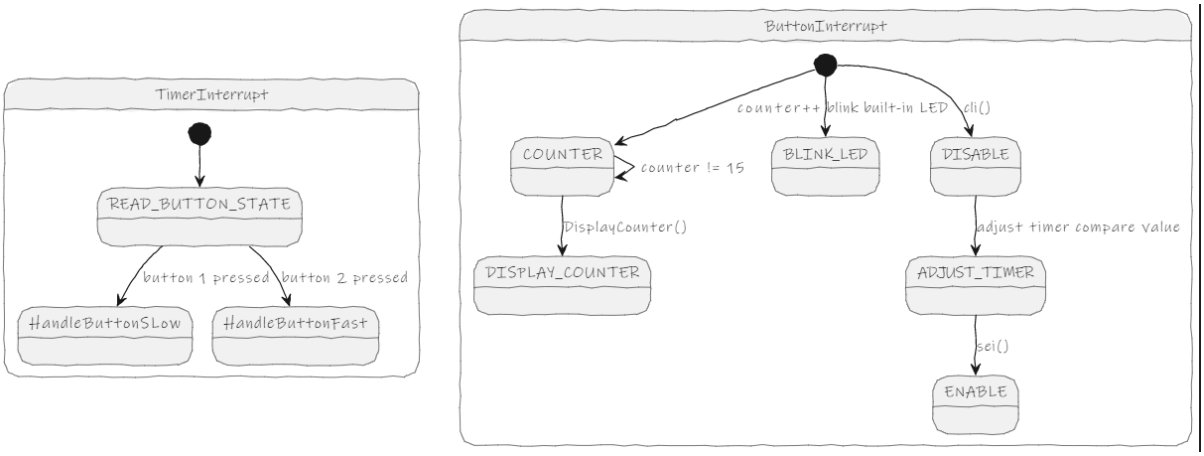


Figure 7: State Diagram C

We will try to base our implementation as much as possible off of this design.

Code:

We begin by including the same important libraries. Mainly the io and the interrupts library (they can be found in the AVR file). We then set up our pins for the buttons and the 4 LEDs. Notice how we used PB0 and PB1 pins for the additional LEDs. We were asked to put the new LEDs on pin d9 and d8. But those pins are not PD pins rather PB pins**[4]**. We then create masks for all of the pins we declared. Again using the _BV function to shift them. We then made some variables for both the counter and the compare values we use for the timer. This is to avoid magic numbers later in our code. We implemented the Millis interval for later use. Finally we have the Boolean value of the buttons. Both for the slow state and the fast state.

We then go on to our setup function. In there we set up both the input and output pins. We then implement the pin interrupt change for the buttons. Then we clear the TCCRI1A register and then proceed to use the TCCR1B to shift the WGM12 variable (what makes the timer run in CTC mode**[1]**. We also set the CS11 and CS10 to it so that we can have a pre-scaler of 64**[2]**. We add the timer compare value to the OCR1A variable. This is used to determine at which speed the timer run. We then enable the timer 1 compare match interrupt. Using the TIMSK1 and the OCIE1A. Finally we enable the global interrupts with sei() function.

Additionally we added two custom functions for the buttons**[5]**. One to slow the timer and one to make it faster. These functions only change the timer compare value on their respective button press. For example, in order for us to slow down the timer, we check if the compare value is smaller than the max compare value. If yes, then we take the timer compare value and increment it with a specific value. The opposite is done for the faster timer**[6]**.

We then use another custom function for displaying the counter on the LEDs**[3]**. The function it self does not increment the counter, but simply displays the value of the counter on the specific LEDs. The way it works is that the counter counts In binary and assigns the binary value to the counter if the if statement within the function turns out to be true.

We then set up our first interrupts service. This acts as the button handler for our system in order to read the value of the buttons and the acting accordingly. The service reads both buttons respectively. Depending on which button mask matches the PINB pin, it will call the custom functions of the button handlers.

We then set up our second interrupt service, This acts as the timer we use in order to display the values of the LEDs and blink the built-in LED. Notice how instead of use the usual PCINT0_vect, we use TIMER1_COMPA_vect. This to show that we do indeed use an interrupt service but with a timer (timer 1 to be specific) vector passed as a parameter instead of the interrupt vector. Within the service is a simple counter variable that increments every timer tick. When the counter reaches 15 it will go back to 0. Again, because an 8-bit counter cannot count above 15. We then call the display counter function. We then implement the blinking for the built-in LED using a simple if-statement implementation**[7]**. Lastly, in order to handle the change in timers we first disable all of the interrupts, set the new timer compare value to the OCR1A register and the enable the interrupts again.

A notable mention within our implementation is that we avoid using the loop function in order to not influence the timing of the interrupts and the timer routines**[8]**.

With this implementation we have fulfilled all of the requirements we needed to fulfil for the part of the Assignment.

Notable anomaly:

As mentioned before, some of our variables within the implementation of part B are declared as volatile. There reason for this is because when a variable is declared volatile, its value can change unexpectedly. In our case, it is related to the interaction between the main program and the interrupt(ISR) service. When a variable is volatile, it can change from external behaviour. That is why all of our variables are declared as volatile. Because they are both used and manipulated by the ISR and the main program. This counts for the timers we use in our custom functions as well.

One of the questions we were asked, were why would we use a pre-scaler of 64 for this implementation. The reason is that the pre-scaler is a division factor for the incoming clock frequency of the timer. We need to use 64 for timer 1 in order to get the desired behaviour of displaying the number 4 times per second. The Arduino functions at a frequency of 16 MHz . If the pre-scaler did not exist, than the timer would also function at 16MHz. So we slow it down. If you do 16MHz / 64, you get 250kHz. If you take the 62500 compare value, and divide it by 250kHz or 250000, you will receive 0.25. meaning ¼, so our comparison of 4 times per second is achieved.

In our application, 1 timer is sufficient because it provides everything we need for the functionality of this project. Its made so that it runs in CTC mode, which allows for periodic interrupts. In our case, its used to generate interrupts at a specific rate to update the counter value and display this on the LEDs.

If we had another task in our project which requires more precise timing, or interrupt handling, then we could implement a separate timer. In other words, you would use different timers if you are working with different timings or multitasking applications.

Conclusion:

The key differences we saw is that interrupts have the potential to work even better than the Millis function used in part A. We would say It's a matter of understanding it better to see how it works. Further assignments/practice need to be done in order to fully understand the concept of interrupts. They are indeed useful because they use what is already built-in the Arduino in order to make it work. So that is even less bytes that is being used to compile the entire program. We came to the conclusion that interrupts are indeed used to handle events quickly. We do not have the amount of knowledge needed in order to make it work flawlessly. For example a blinking LED at 1hz will work fine for quite a while, before it blinks a bit faster for 1 of the seconds before continuing its usual pattern. If we can eliminate that split second, then we can conclude that interrupts are very powerful tools that can be used while programming an Arduino.

As for the timers, we concluded that they function almost the exact way as how you would utilize the Millis function instead of delays. They don't interfere with the rest of the code.

References:

[1] Atmega328P Microcontroller – Atmel, Enabling Unlimited Possibilities (n.d.) https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf