# ECE 341 - Lab #5

Collin Heist

March 1, 2019

## Contents



## Listings

# 1 Introduction

The purpose of this lab is to re-implement the previous lab, **Lab #4**, but to use only foreground processes. I'll be using two interrupts for this current lab, one for Timer 1 (set to one millisecond), and the second for the button presses on **BTN1** and **BTN2**. Unlike the previous lab's, this code will not have multi or single-rate processes in the **main()** function, it will instead have all the processing code through the various Interrupt Service Routines.

The only relevant background information is the configuration of the interrupts. I'll add all the necessary interrupt configuration to the **system_init()** function. After configuring both of these interrupts, they will both need their respective interrupt service routines (ISR's). These routines are called by the *system*, and when the necessary interrupt condition is met (either one millisecond passing, or a button press) the ISR is then automatically run by the system. Because of the fact they're executed automatically by the micro-controller, my code will have no direct calls to the ISR functions. In fact, it is actually impossible to call an ISR directly (like a normal function).

The ISR's require special syntax because they require special parameters. In particular, you need to tell the compiler which interrupt vector to tie that particular ISR. This ties each interrupt, so either the button presses or timer, to the respective function execution (so that the right routine is called when a flag is triggered). In the unlikely event two interrupts are triggered at exactly the same time, or one interrupt is triggered during the execution of the another, the priority of those respective interrupts is what determines which takes precedence. Our processor, the **PIC 32** supports 7 levels of priority (plus priority 0 for the background tasks), which a higher level signifying a more important task.

For this lab, no code we'll be executing fits this priority, but if it is essential that a section of code is uninterrupted, interrupts can be temporarily disabled for the runtime of that code's execution. An example of this type of code would be timing-specific code, like responses to incoming signals and all that. Rather than giving that whole chunk of code a higher priority than the other interrupts, disabling and afterwards re-enabling interrupts allows for only the 'protected' bits to execute unimpeded. This type of protection can be implemented by disabling the interrupts with the **INTDisableInterrupts()** function, and then once the protected code has been finished, re-enabling them with **INTEnableInterrupts()**. A possible consequence of this type of code is that very high-priority interrupt routines might be addressed later than you'd like, because their code could not be executed until the interrupts were re-enabled. To apply this example to the lab, if we

protected the motor output code (for example), but then the one millisecond Timer 1 interrupt triggered during that block of code, the delay would be off by however long it took to execute that chunk of code.

# 2 Implementation

The first thing that I needed to change was the **system_init()** function. In previous labs, we just needed to initialize the Cerebot, and then set the buttons and pins as inputs and outputs. The adjustments I've made are attached:

Listing 1: System Initialization

```
void system_init(void) {
  ... // I/O Config. from Lab #4

  OpenTimer1(T1_ON | T1_SOURCE_INT
    | T1_PS_1_1, T1_TICK-1);
  mT1SetIntPriority(2);
  mT1SetIntSubPriority(0);
  mT1IntEnable(1);

  mCNOpen(CN_ON,(CN8_ENABLE | CN9_ENABLE), 0);
  mCNSetIntPriority(1);
  mCNSetIntSubPriority(0);
  unsigned int x = PORTReadBits(IOPORT_G,
    BTN1 | BTN2);
  mCNClearIntFlag();
  mCNIntEnable(1);

  INTEnableSystemMultiVectoredInt();
}
```

I did not include the unchanged code from the previous labs. The above code instead configures the two necessary interrupts for this lab. Timer 1 is opened with the necessary period of one millisecond (where **T1_TICK-1** is defined in the header file). The timer's priority is then set in accordance with the lab worksheet, and finally it is enabled.

Afterwards, the change notice interrupt is setup in a similar way. Using the provided change notice table, we setup the interrupt for **CN8** and **CN9**,

which correspond to **BTN2** and **BTN1**. The buttons are given a priority lower than the timer interrupt, and then turned on, finishing the system initialization.

With both interrupts configured, the next bit of code that needed to be written was their respective interrupt service routines (*ISR)*. The ISR for Timer 1 is shown below:

Listing 2: Timer 1 Interrupt Service Routine

```
void __ISR(_TIMER_1_VECTOR, IPL2) Timer1Handler() {
  LATBINV = LEDA;
  if (!(--step_delay)) {
    stepper_state_machine();
    unsigned int del = (control_mode & 12) >> 2;
    unsigned int rev_min  = (del == RPM10) ?
      10 : (del == RPM15) ? 15 : 25;
    step_delay = 1.0 /
      ((float) rev_min * 100.0 / 60.0 / 1000.0);
  }
  mT1ClearIntFlag();
}
```

This code is largely identical to the previous week's **step_delay** if statement, and it is functionally equivalent. The function is first declared as an ISR, hence the abnormal syntax, and then the step delay is checked. As opposed to previous labs, where the current value of step delay was passed *into* a function and then decremented, since an ISR cannot have any parameters passed into it (by reference or by value), **step_delay** is a *global* variable so it can be changed anywhere. If the delay is at zero (meaning the proper time has passed), then state machine is progressed, and then the **step_delay** is reset to the proper value based on the current value of **control_mode** (another global variable). Independent of whether or not a step was taken, the interrupt flag is finally cleared to prevent the ISR from constantly triggering.

Just like how the first ISR follows the same general structure as the **step_delay** if statement from the previous Lab, the button ISR mirrors the **button_delay** statement from that same lab. That's shown in Listing 3:

Listing 3: Button Interrupt Service Routine

```
void __ISR(_CHANGE_NOTICE_VECTOR, IPL1)
  CNIntHandler() {

  LATBINV = LEDC;
  hw_delay(20);
  unsigned int buttons = read_buttons();
  decode_buttons(buttons);
  LATBINV = LEDC;

  mCNClearIntFlag();
}
```

This simple ISR utilizes the hardware-assisted delay from Lab #2 to wait 20 milliseconds for the buttons to 'debounce' and settle on a steady-state value. Once that time has passed, the buttons are read (same way as before), and their value is stored in the local variable **buttons**. The reason I did not use a global variable here is because it was unnecessary as no other functions need access to the status of the buttons themselves, only the corresponding control mode (which is updated inside **decode_buttons()**). Once again, after all the 'logic' has finished, the interrupt flag is cleared to allow the code to resume it's normal operation.

I changed only one line of the **decode_buttons()** function. Rather than *returning* the control mode, since it's value is being used by other functions throughout the program, and is primarily called by an ISR, the control mode is instead stored into a global variable.

Listing 4: Button Decoding

```
void decode_buttons(unsigned int portG_state) {
  unsigned int btns = portG_state >> 6;
  control_mode = (btns < 2 ? 4 + btns :
    ((btns & 1) << 3) + btns); // Global variable
}
```

And one more very small change was made to the **stepper_state_machine()** function. Previously, it took in the control mode as a parameter, and returned the output code that was then sent to the stepper motor. Since the ISRs need to access these variables globally, there is no need for a return or parameter.

Listing 5: Revised Finite State Machine

```
void stepper_state_machine(void) {
  ... // Same FSM code as Lab #4
  output_to_stepper_motor(return_modes[state]);
}
```

As shown above, the code now directly calls the **output_to_stepper_motor()** function, rather than returning the value.

With all the above-shown functions changed to use global variables and interact with the ISR as needed, the while loop was the only thing that needed to be changed. Because the interrupts are triggered automatically by the system when their respective conditions are met, the main **while(1)** loop does not need to contain any code. All the function calls are handled by each interrupt routine, leaving the **main()** function looking like this:

Listing 6: Infinite Program Loop

```
int main() {
  system_init();

  while (1) {}

  return 0;
}
```

# 3 Testing and Verification

All relevant oscilloscope screen captures are listed in the **Attachments** section. In each capture, $D_0$ is **LEDA**, toggled every one millisecond; $D_1$ is **LEDB** that is toggled every time an output is sent to the motor; and $D_2$ is **LEDC**, that is toggled at the beginning and end of the button-press **ISR**. As evident in the captures, $D_2$ triggers when I push the button, and thanks to the interrupt priority level's being configured correctly, $D_0$ does not miss a pulse during the runtime of that **ISR**. This is present in all the captures, and shows that the priority level of the Timer 1 **ISR** is *greater* than that of the change notice button-presses. This is done intentionally to maintain accurate timing schemes throughout the runtime of the lab.

Because the Timer 1 **ISR** has the highest priority in our program, there is literally nothing that will prevent the **ISR** from triggering as soon as the flag

is set. This results in the stepper motor *technically* running more accurately in this lab, relative to before. In the previous lab, there would be instances where the motor's code would execute when the button code did not, or the other way around. This would result in the stepper motor code executing with small variations in the time between pulses, depending on whether or not the buttons were checked, if the button was pressed, etc. Having all the motor triggered through **ISR** results in nothing possibly interrupting the code when it needs to execute, increasing the consistency of the delay between pulses.

# 4 Conclusion

This lab served as an excellent introduction of foreground tasks (interrupts) and how they're useful in implementing various programs. Having already done this lab a few times, one using simple round-robin scheduling, and one using multi-rate processes, this seems (to me) to be the most efficient implementation. The advantage of this kind of real-time control system with interrupts is how all control happens nearly instantaneously. In our previous labs that utilized polling of the timer interrupt flag (rather than configuring an ISR), a significant portion of the processor's computation time was spent just waiting for a flag to trigger. Compare that to this lab where the system is 'free' to do whatever it wants (even though nothing happens in this particular lab), and the interrupts trigger events on their own, this is far more efficient and responsive. Despite the benefits of an interrupt-driven control system, they're not inherently better all the time. An example of this would be our button presses. Handling the button input ASAP is largely useless for this type of application, because the user's ability to delineate a 100 millisecond response time from a 1 millisecond response time is largely negligible. For this reason, the necessary overhead of the interrupt *might* make simple polling a more accurate fit.

In general, interrupt-based control schemes require more overhead, but are far quicker to respond, and great for urgent event handling. On the other hand, polling schemes are very easy to implement with practically no overhead, and great for non-essential tasks, especially if they happen are a fixed rate.

In a polling-based system, the worst-case for response is how often the polling cycle is. For example, if the relevant flag is polled once every cycle, and that cycle takes 10 ms and the flag is set right after it was polled, then the system will take 10 ms to respond.

On the other hand, if an interrupt-based system is used then the system

will at most take as long as all interrupts that have a higher priority (in the rare case that all of those interrupts were triggered at the same time), plus the time an interrupt takes to trigger (essentially negligible). In this lab, for example, the longest possible time it would take for the button-press interrupt to execute would be the length of the Timer 1 interrupt's (as it is high priority, and there is no protected code).

# 5 Attachments

As previously said, $D_0$ is **LEDA**, toggled every one millisecond; $D_1$ is **LEDB** that is toggled every time an output is sent to the motor; and $D_2$ is **LEDC**, that is toggled at the beginning and end of the button-press **ISR**

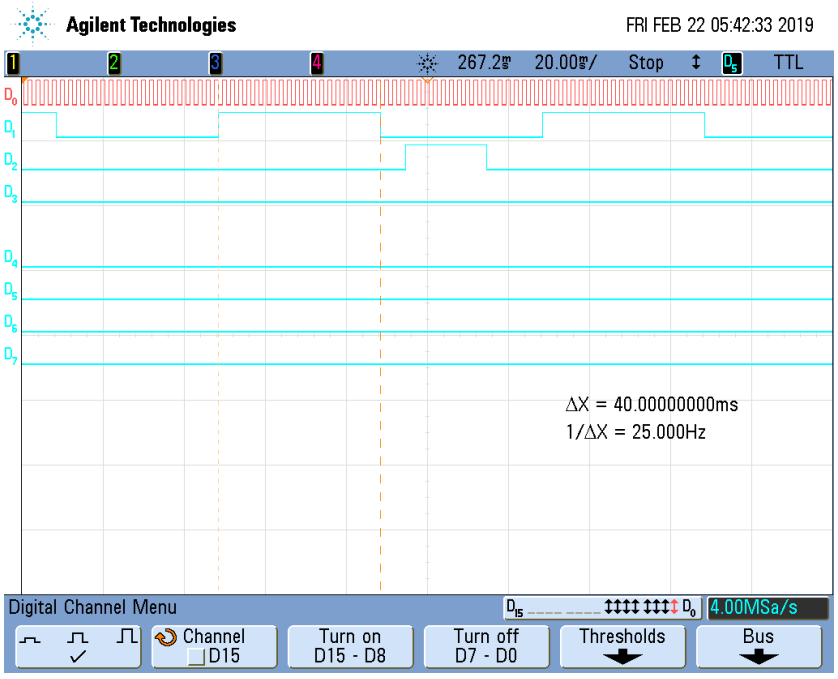


Figure 1: The off-off, 40ms delay with button delay

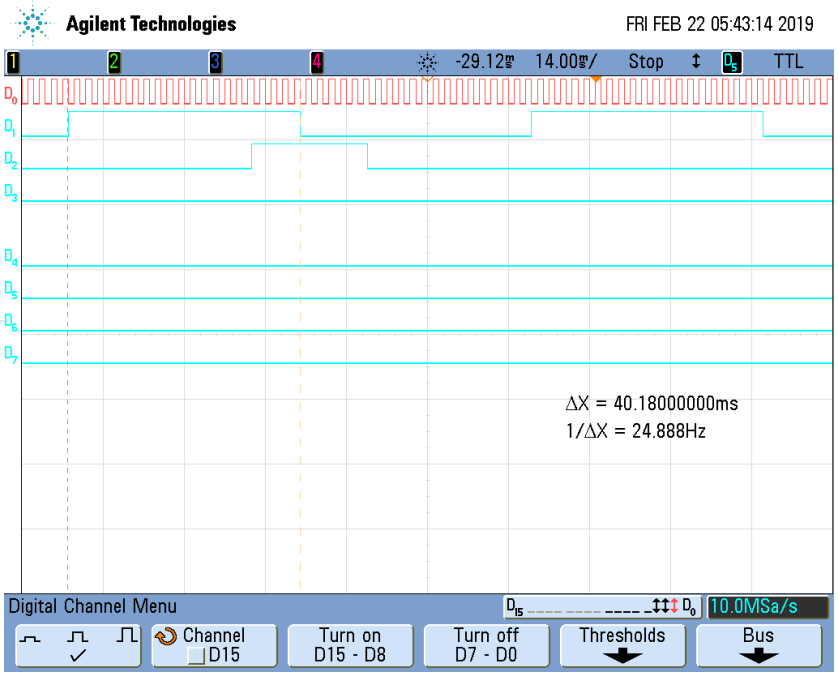


Figure 2: The off-on 40ms delay

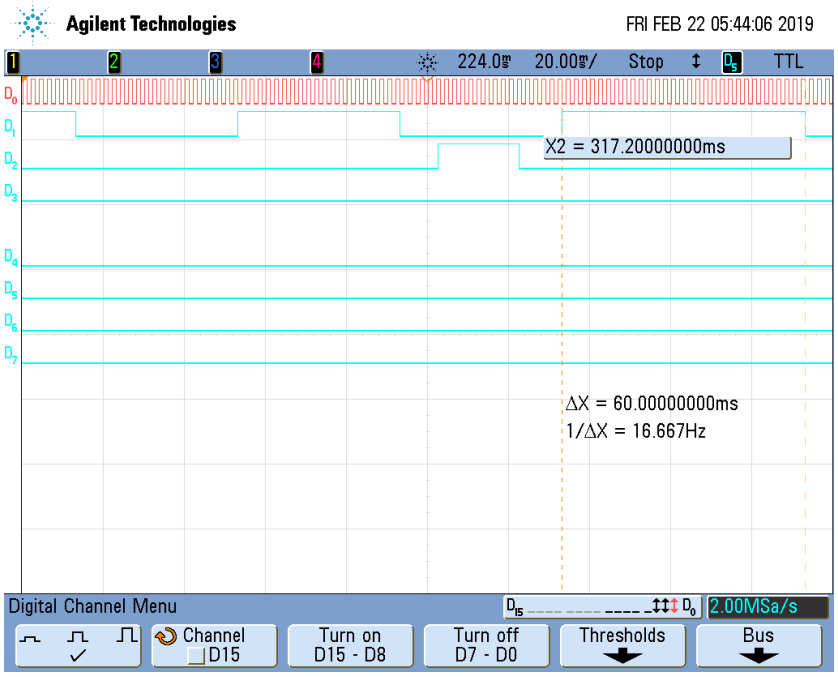


Figure 3: The on-off 60ms delay

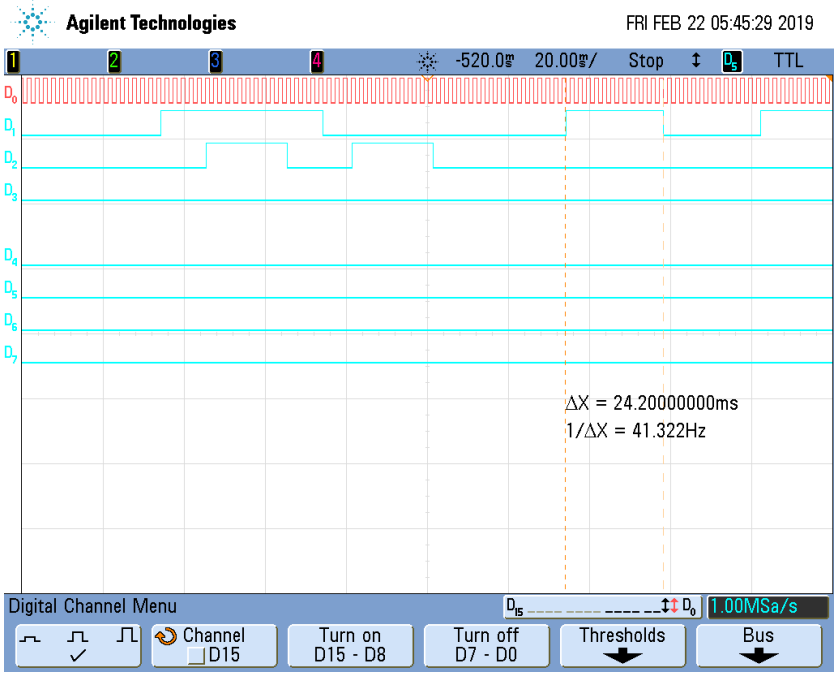


Figure 4: The on-on 24ms delay