Lab 5

Report

**Owen Blair  
10/12/2021**

Contents

[Goal and Background Information 2](#_Toc84877389)

[Implementation 4](#_Toc84877390)

[Testing and Verification 4](#_Toc84877391)

[Table 1 4](#_Toc84877392)

[Figure 1: Waveform Interrupts 5](#_Toc84877393)

[Conclusion 5](#_Toc84877394)

[Listings 5](#_Toc84877395)

[Listing 1: Timer 1 ISR 5](#_Toc84877396)

[Listing 2: Change Notice ISR 5](#_Toc84877397)

[Listing 3: Timer 1 Interrupt Setup 6](#_Toc84877398)

[Listing 4: Change Notice Interrupt Setup 6](#_Toc84877399)

# Goal and Background Information

The goal of this lab is to implement an interrupt driven management scheme. This is done by utilizing two interrupts: change notice interrupt and Timer 1 Interrupt flag. When an interrupt is set the CPU will be forced to a specified memory location and will execute the user code and then go back to the background tasks. Section of code that the CPU will be forced to is called an Interrupt Service Routine (ISR). These ISRs are coded like functions but are not called like functions. ISRs are dependent on events happening outside the CPU, unlike functions, to trigger the interrupt handler and force the CPU to an ISR. Because ISRs are used differently than functions they require special syntax to identify them to the compiler. If multiple interrupts are triggered the setup priority of each interrupt determines what order the ISRs will be processed in.

At certain times sections of code must be protected from interrupts. These sections are things like setups or a higher priority interrupts. The consequences of entering a protected section of code after the setup portion of code is that the PIC32 might miss an outside event that would have triggered an interrupt. For example, if a change notice interrupt is setup to go high when the input changes and interrupts are disabled then the code within the change notice ISR will not execute. The other option is that a higher priority is running, and a lower priority goes high. This means that the lower priority function will execute after the higher priority interrupt. This may cause ISRs to not operate as intended.

The code the CPU will be forced to when T1IF gets set contains the stepTimer counter variable decrement, stepper state machine, and outputToStepper functions from the previous lab. The difference for this lab is that there will not be a counter for when the button inputs should be sampled as that will be handled by the CNIF and that the if statement that was previously in the while(1) loop will now be executed within the T1IF code and the while(1) loop will be empty. It should also be noted that the stepDelay variable counter and the stepperState variable will need to be a global variable so that the code in the interrupt will be able to access and modify the variable value.

The CNIF code will contain the readButtons and decodeButtons functions from the previous lab and will also include a new delay that will handle button debounce. When the CPU is forced to the change notice section of code by the interrupt handler it will first execute a delay for button debounce, read the buttons, and then decode the buttons and return the global variables to be used in the T1IF code. The global variables

Graphical user interface, diagram, application, Teams

Description automatically generated**Data Flow Diagram**

Diagram

Description automatically generated**Control Flow Diagram**

# Implementation

The implementation of the code for lab 4 went as planned. This was because most of the code needed for the lab was written previously. The modification needed for the lab was to move the code into the needed ISR sections. The read buttons and decode buttons were moved into the change notice ISR and the stepper state and output to stepper function to the timer 1 ISR. An important thing to note is that the change notice ISR has a 20ms delay waiting for the button to debounce before reading the button inputs. Global variables are also needed to pass information to and from ISRs because pointers won’t work.

# Testing and Verification

Two forms of verification were used to test and verify that the code written for this lab was working correctly. The first is visual inspection. This consisted of pushing the buttons on the MX7 board and seeing if the stepper motor acted accordingly. The behavior of the stepper motor should follow the behavior modeled in Table 1. Once visually confirmed to be moving in the correct directions and estimated speed testing moved on to the next form of verification.

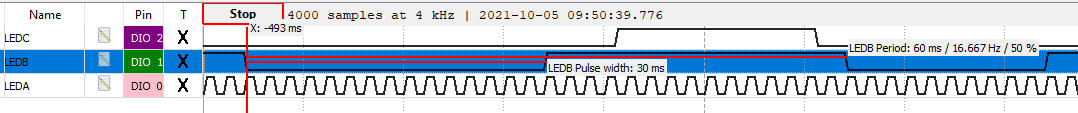
The second form of verification was measuring when the PIC32 would execute a set of instructions related to an ISR. This was implemented in the code by having each ISR to be tested invert a specific LED. This allowed for the testing of the digital waveform using the Analog Explorer module and the Waveform program to see the priorities of each interrupt. The waveform showing interrupt priority can be seen in *Figure 1: Waveform Interrupts.* In the figure, LED A is toggling even though LED B and C are between toggles. LED A is toggled in the highest priority (Timer 1 Interrupt Flag), LED B is toggled every time the stepper motor takes a step, and LED C is high when the change notice ISR is running high and is low when the change notice ISR ends. Further, the interrupt priority can be seen in the figure because LED A toggles even though LED B and C is high. It can also be seen that LED C has a higher priority then the ISR related to LED B because LED C changes states even though LED B is high.

Compared to Lab 4 the stepper motor is more accurately moved using interrupts. This is because an interrupt changes at exactly the time it needs to unlike a delay routine using delay blocking. This can cause timing issues because the time to execute a step is less accurate because the rest of the code needs to execute as well. *Listing 1: Timer 1 ISR* contains the code that runs when the timer 1 interrupt flag goes high. *Listing 2: Change Notice* contains the code that runs every time the change notice interrupt flag goes high. The following listings contain the setup code for the interrupt setups: *Listing 3: Timer 1 Interrupt Setup, Listing 4: Change Notice Interrupt Setup.*

## Table 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Inputs | | Controls | | | |
| BTN 1 | BTN 2 | Step Mode | Speed (RPM) | Desired Step Delay (ms) | Measured Step Delay (ms) |
| Off | Off | Half | 15 | 20 | 20 |
| On | Off | Full | 15 | 40 | 40 |
| Off | On | Half | 10 | 30 | 30 |
| On | On | Full | 25 | 24 | 24 |

## Figure 1: Waveform Interrupts



# Conclusion

The advantages of using a real-time control system that using polling is that it is easy to code and does not utilize peripherals for timing. An advantage of an interrupt-based control systems is that timing events can be more accurately timed, and the CPU does not need to sit idle and continually poll for an event. The disadvantage of an interrupt system is that if there are too many interrupts latency to necessary ISRs can be long. The worst case for latency in a polling system is the time it takes to execute all the other code in the main loop to return to the polling event again. The worst case of latency in an interrupt-based system is the time it takes to go through every interrupt of a higher priority.

# Listings

## Listing 1: Timer 1 ISR

//T1IF!!

void \_\_ISR(\_TIMER\_1\_VECTOR, IPL2) Timer1Handler(void)

{

int testVar = 0;

stepTimer--;

if(stepTimer <= 0)

{

testVar=1; //Used for testing

stepState = stepperStateMachine(direction, stepState, mode);

outputToStepper(stepState);

stepTimer = delay;//If buttons don't change keep at same speed (delay)

}

LATBINV = LEDA; //toggle LED A

mT1ClearIntFlag();

}

## Listing 2: Change Notice ISR

void \_\_ISR(\_CHANGE\_NOTICE\_VECTOR, IPL1) CNIntHandler(void)

{

LATBINV = LEDC;

hw\_msDelay(20);

int btnInput = read\_buttons(); //Read and assign value for later use

decode\_buttons(btnInput, &delay, &mode, &direction);

/\* Required to clear the interrupt flag in the ISR \*/

mCNClearIntFlag(); // Macro function

LATBINV = LEDC;

}

## Listing 3: Timer 1 Interrupt Setup

void timer1\_interrupt\_initialize(void)

{

//configure Timer 1 with internal clock, 1:1 prescale, PR1 for 1 ms period

OpenTimer1(T1\_ON | T1\_SOURCE\_INT | T1\_PS\_1\_1, T1\_INTR\_RATE-1);

// set up the timer interrupt with a priority of 2, sub priority 0

mT1SetIntPriority(2); // Group priority range: 1 to 7

mT1SetIntSubPriority(0); // Subgroup priority range: 0 to 3

mT1IntEnable(1); // Enable T1 interrupts

// Global interrupts must enabled to complete the initialization.

}

## Listing 4: Change Notice Interrupt Setup

/\* Initialization of CN peripheral for interrupt level 1 \*/

void cn\_interrupt\_initialize(void) // Code that is executed only once

{

unsigned int dummy; // used to hold PORT read value

// Enable CN for BTN1 and BTN2

mCNOpen(CN\_ON,(CN8\_ENABLE | CN9\_ENABLE), 0);

// Set CN interrupts priority level 1 sub priority level 0

mCNSetIntPriority(1); // Group priority (1 to 7)

mCNSetIntSubPriority(0); // Subgroup priority (0 to 3)

// read port to clear difference

dummy = PORTReadBits(IOPORT\_G, BTN1 | BTN2);

mCNClearIntFlag(); // Clear CN interrupt flag

mCNIntEnable(1); // Enable CN interrupts

// Global interrupts must enabled to complete the initialization.

}