Lab 4: PIC 32 Timers

Report

**Owen Blair  
10/5/2021**

Contents

[Introduction 3](#_Toc84314934)

[Equation 1: T1CLK Formula 3](#_Toc84314935)

[Data Flow Diagram 3](#_Toc84314936)

[Control Flow Diagram 4](#_Toc84314937)

[Implementation 4](#_Toc84314938)

[Testing and Verification 5](#_Toc84314939)

[Table 1 6](#_Toc84314940)

[Figure 1: Button 1 Off and Button 2 Off 6](#_Toc84314941)

[Figure 2: Button 1 ON and Button 2 Off 6](#_Toc84314942)

[Figure 3: Button 1 Off and Button 2 On 6](#_Toc84314943)

[Figure 4: Button 1 On and Button 2 On 6](#_Toc84314944)

[Figure 5: Read Buttons Command Signal 6](#_Toc84314945)

[Figure 6: Timer 1 Interrupt Flag Signal 7](#_Toc84314946)

[Conclusion 7](#_Toc84314947)

[Listings 7](#_Toc84314948)

[Listing 1: timer1Delay Code 7](#_Toc84314949)

[Listing 2: system\_init() Code 7](#_Toc84314950)

[Listing 3: Main Loop Code 8](#_Toc84314951)

[Attachment 1: Limitations of Timer 1 Peripheral 9](#_Toc84314952)

[Attachment 2: Change Delay Period Error 10](#_Toc84314953)

# Introduction

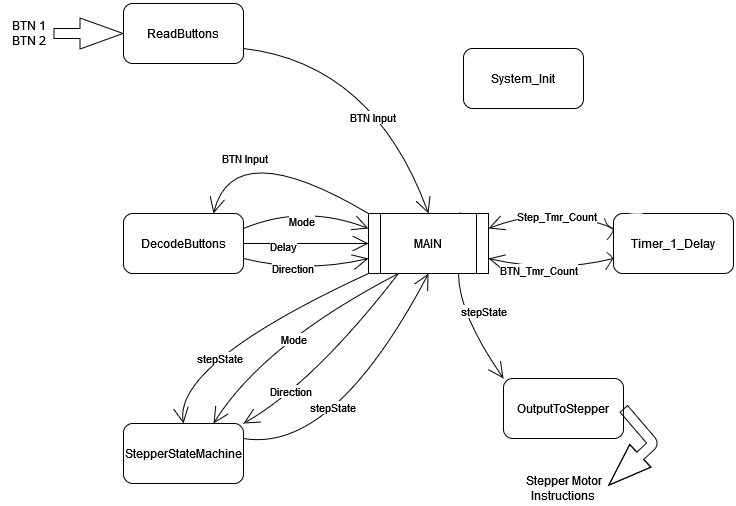
The goal of this lab is to implement a synchronized multi-rate control system by using timer 1 interrupt flag. This interrupt flag will go high every millisecond and us used to decrement two counting variables called buttonTimer, and stepperDelay. When the btnTimer variable reaches zero the program executes the readButtons() function, executes the decodeButtons() functions, and lastly resets the btnTmer to 100. When the stepperDelay variable reaches zero then the program executes the stepperStateMachine() function, executes the outputToStepper() function and sets the stepTimer to the desired delay in milliseconds to reach a specified RPM.

Timer 1 interrupt flag for this lab will be obtained in this lab by using a macro called mT1GetIntFlag(). It is important to note that equation 1 (also shown below) in the lab handout shows how to calculate how long the timer 1 takes to increment once. Also, while deciding on what to make the timer 1 period it is important to remember that the clock will increment the number of times equal to the PRx value +1 because it starts counting from zero. Most of the code from this lab is already written. The main difference between this lab and the last one is that this has a different scheduling method.

## Equation 1: T1CLK Formula

**T1CLK = XTAL \* FPLLMUL / ( FPLLIDIV \* FPLLODIV \* FPBDIV \* TCKPS)**

## Data Flow Diagram



## Control Flow Diagram

**Diagram

Description automatically generated**

# Implementation

The implementation of the code for lab 4 went as planned with a couple problems of trying to use the correct symbol for a pointer in the correct place. Another interesting thing to note is that this program is not entirely interrupt controlled and still used ‘blocking’ style of delays. This is more accurate that a software delay and a hardware delay, but the CPU still needs to sit in a while loop waiting to receive the timer 1 interrupt flag. This can be seen in *Listing 1: timer1Delay Code*. Within the timer1Delay function as it was first implemented there was an error using pointer operators and the shorthand decrement symbol (--). The compiler would give an error and would fail to build the program. I do not know why this happens, but an alternative solution was found by having the \*btnTimer and \*stepTimer equal their current value minus one.

There was little other modification needed for this lab because the previous version did not initialize global variables and instead defined all variables used as local to the main function and used pointers to modify their values within functions. The other modification needed was adding another LED to invert to test how often the timer 1 interrupt flag was set. It is also important to note that a new line was added to the system\_init() function that sets up the timer 1 interrupt flag. The modified system\_init() function can be seen in Listing 2. The tough implementation of the previous lab really helped the implementation of this lab be more efficient. This is because most of the bugs that are contained in the code to gather user input and move the stepper motor were worked out in the previous lab.

The added counters used to control how often the program executed the read buttons function or the stepper motor FSM and outputs functions are of the integer data type. The reason for this was because the value needed to be a number that counted from a discrete starting point to zero in increments of 1 every time the timer1Delay function was executed. This meant that an unsigned integer was not a good choice because there were troubles in the previous lab when an unsigned integer was assigned to zero. A float data type would work but had the possibility to end up as a non-discrete number is something went wrong.

Variables used for timing the stepper motor FSM and the read buttons functions counted down. The reason for this was that the value that was used to compare the countdown values is 0. This way whenever any counter is equal to zero that section of code would be executed. This also allowed the code to assign how long the delay for certain sections of code would be without needing to adjust the value that the button count was compared to when deciding what functions to execute. This can be seen in *Listing 3: Main Loop Code* along with the countdown variables being tested each iteration to see if they have reached zero.

When checking to see if a delay period has expired the code compares the countdown variable to see if it is equal to zero. An advantage with using the <= operator for my code would be if something went wrong and the countdown variable decremented twice instead of once the less than or equal to operator would catch the change and execute the functions necessary to reset to the variable to its needed value where the equal to operator would not. The equal to operator not catching the accidental decrement by two would result in the function always decrementing the countdown variable into negative numbers and would never execute the corelated functions again and reset the countdown variable.

# Testing and Verification

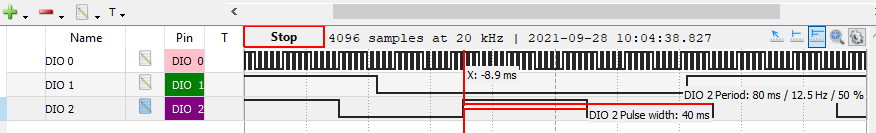
Two forms of verification were used to test and verify that the code written for this lab was working correctly. The first is visual inspection. This consisted of pushing the buttons on the MX7 board and seeing if the stepper motor acted accordingly. The behavior of the stepper motor should follow the 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 how often the PIC32 would execute a set of instructions. This was implemented in the code by having each part 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. Along with visual behavior, Table 1 also contains the measured step delays using the Analog Explorer module and the Waveform program. Figure 1 contains the step delay for the first row of data, Figure 2 contains the measured step delay for the second row of data and so on and so forth. Figure 5 contains the measured digital waveform of how often the PIC32 read the input buttons and Figure 6 is the waveform that toggled an LED every time the timer 1 interrupt flag was set.

## 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: Button 1 Off and Button 2 Off



## Figure 2: Button 1 ON and Button 2 Off

Diagram

Description automatically generated with medium confidence

## Figure 3: Button 1 Off and Button 2 On

A picture containing table

Description automatically generated

## Figure 4: Button 1 On and Button 2 On

Diagram

Description automatically generated

## Figure 5: Read Buttons Command Signal

A picture containing graphical user interface

Description automatically generated

## Figure 6: Timer 1 Interrupt Flag Signal

Application, table

Description automatically generated

# Conclusion

1. See *Attachment 1: Limitations of Timer 1 Peripheral* for the answers to this question.
2. The period register (PRx) affects the accuracy of the timer delay. This is done by making the period register value larger, decreasing resolution, or making the period register value smaller and increasing resolution.
3. See *Attachment 2: Change Delay Period Error* for answers to this question.
4. The differences between how the hardware assisted delay and software delay, and how the Timer 1 peripheral was in one major way. The Timer 1 was an off CPU resource that sets a T1IF signal every time Timer 1 reaches its period and needs to be reset. This means that as the program is running Timer 1 is running as well. This means that the time to execute tasks that are in the background don’t extend the period of the Timer 1 interrupt flag like it would with a hardware assisted or software delay.

# Listings

## Listing 1: timer1Delay Code

void timer1Delay(int \*btnTimer, int \*stepTimer)

{

while(!mT1GetIntFlag());

\*btnTimer = \*btnTimer -1;

\*stepTimer = \*stepTimer -1;

/\*\*btnTimer--; //DOES NOT WORK!

\*stepTimer--; \*/

LATBINV = LEDA; //toggle LED A

mT1ClearIntFlag();

}

## Listing 2: system\_init() Code

/\* initialize\_system Function Description \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* SYNTAX: void system\_init();

\* PARAMETER1: No Parameters

\* KEYWORDS: initialize

\* DESCRIPTION: Sets Registers to default (0) to prevent non zero values

\* from popping up.

\* RETURN VALUE: None

\* END DESCRIPTION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void system\_init(void)

{

Cerebot\_mx7cK\_setup(); // Initialize processor board

PORTSetPinsDigitalOut(IOPORT\_B, SM\_LEDS);/\* Set PmodSTEP LEDs outputs \*/

//Set up timer 1!

OpenTimer1(T1\_ON | T1\_PS\_1\_1, (T1\_TICK - 1));//Set up T1IF

LATBCLR = SM\_LEDS; /\* Turn off LEDA through LEDH \*/

}

## Listing 3: Main Loop Code

int main()

{

system\_init(); /\* Setup system Hardware. \*/

unsigned int btnInput = 0;

unsigned int delay;

unsigned int mode;

unsigned int direction;

unsigned int stepState = 0x0A;

//Set to zero so when passed to the timer1Delay function

//the values will be zero and both if statements should

//run the code within them

int btnTimer = 1;

int stepTimer = 1;

/\*

//Starting point

btnInput = read\_buttons();

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

stepState = stepperStateMachine(direction, stepState, mode);

outputToStepper(stepState);

\*/

while(1)

{

timer1Delay(&btnTimer, &stepTimer);

if(btnTimer == 0){

btnInput = read\_buttons();

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

LATBINV = LEDB; //Toggle Led B

btnTimer = 100; //Reset btnTimer for 1 ms

}

if(stepTimer == 0){

stepState = stepperStateMachine(direction, stepState, mode);

outputToStepper(stepState);

LATBINV = LEDC; // Toggle LED C

stepTimer = delay;

}

}

return 0; /\* Returning a value is expected but this statement

\* should never execute \*/

}

# Attachment 1: Limitations of Timer 1 Peripheral



# Attachment 2: Change Delay Period Error

