Revision: 29 JAN 2019 (JFF)
Richard W. Wall, University of Idaho, rwall@uidaho.edu

www.digilentinc.com
1300 NE Henley Court, Suite 3
Pullman, WA 99163
(509) 334 6306 Voice | (509) 334 6300 Fax

Project 2: Software Time Delay

Table of Contents

Purpose

The purpose of this project is to investigate methods of creating software time delays to pace processor operations. The timing methods that control when a process is to be executed can be implemented using polling techniques that sample a free running clock or by executing code that requires a fixed amount of time. This project also demonstrates the use of time delays for signal conditioning.

Minimum Knowledge and Programming Skills

1. Knowledge of C or C++ programming
2. Working knowledge of MPLAB ® X IDE
3. Binary math, Boolean algebra, bit manipulation
4. IO pin control (see Project 1)

Equipment List

1. Cerebot 32MX7cK processor board with USB cable
2. PmodTPH1, PmodTPH2 or PmodSTEP
3. Logic analyzer, or oscilloscope (Suggestion - Digilent Analog Discovery)

Software Resources

1. XC32 C/C++ Compiler Users Guide
2. PIC32 Peripheral Libraries for MBLAB C32 Compiler
3. PIC32 Family Reference Manual Section 14: Timers
4. Cerebot MX7cK Board Reference Manual
5. MPLAB ® X Integrated Development Environment (IDE)
6. C Programming Reference

Time Delay Concepts - *How long does it take?*

Usually the time required to execute software is a detriment to system performance. However, there are instances when the program actually runs too fast and we are challenged with developing ways of slowing it down. Each instruction that the processor executes requires a finite amount of time. Sometimes we can use the inherent code execution time to our advantage by using program loops to generate a delay. Consider the program shown in Listing 1 that adds two integer variables and assigns the result to a third integer variable:

Listing 1.

```
/*
 * File:   P1.c
 * Author: Richard Wall
 *
 * Created on July 5, 2013, 12:56 PM
 */

#include <plib.h>
#include "..\config_bits.h"

int main(void)
{
int a = 5, b = 10, c;

   c = a+b;
   return (EXIT_SUCCESS);
}
```

Each C source code statement is implemented by translating it into instructions that a given processor understands and then acting on those instructions. The actual processor-level instructions that implement this statement can be represented in assembly language. Unlike C, which is considered a “high level” language that is independent of hardware, assembly language is specific to the particular processor on which it runs. The assembly language instructions are subsequently converted to a machine language that the processor actually uses. Although it is beyond the scope of this project to teach assembly language programming, we will consider a little bit of assembly code because it is instructive in terms of timing considerations. You do not need to be especially concerned if some of the details seem a bit mysterious. The C statement, *c = a+b;,* in the program shown in Listing 1 is implemented as a sequence of four assembly language instructions as shown in Listing 2.

Listing 2. Assembly language code generated for the add statement.

```
9D000038  8FC30010      lw     v1,16(s8)  ; Load word Reg V1 indirect
9D00003C  8FC20014      lw     v0,20(s8)  ; Load word Reg V0  indirect
9D000040  00621021      addu   v0,v1,v0   ; Unsigned Integer Add
9D000044  AFC20018      sw     v0,24(s8)  ; Store word indirect
```

The first two columns of Listing 2 are 32-bit numbers expressed in hexadecimal notation. The first column shows the memory address where the machine code is stored. The second column is the actual machine code that the processor uses to implement the instruction shown in third and forth columns. The third column gives the type of assembly language instructions used and the processor memory registers that are involved are given in the fourth column. Any text that follows a semicolon is processed as a comment as illustrated by the fifth column.

To determine how many cycles each instruction takes we need to look at the assembly language for the MIPS architecture that the PIC32MX uses. The conversion of C instructions to assembly code can be different depending on the selection of compiler optimization options. If the PIC32MX is running the core clock at 80MHz, one CPU cycle is 12.5 ns (i.e., the period of one clock cycle is the inverse of the clock frequency).

Due to the pipeline architecture used by the MIPS core, it is difficult to isolate a segment of code and determine its execution time strictly based upon the number of instructions. By using the Stopwatch capability of MPLAB ® X to time the four instructions in Listing 2, we find that 31 CPU cycles are required. (See Appendix C for instructions on how to use the MPLAB ® X Stopwatch feature.) This equates to 387.5 ns when the core clock is running at 80MHz. Setting aside the difficulties with determining time to execute from lines of assembly code, the sheer number of instructions is a relative indicator. The execution time required to complete any particular C instruction is determined by the complexity of the C statement, the assembly code that the C instruction is translated into, and the speed that the processor is running.

It is sufficient to say that each C instruction can require multiple assembly language instructions to implement and each assembly instruction requires time to execute. The bottom line is that the execution of C code requires time and we can use this time to generate predictable time delays. An alternate method to generate a time delay is to use one of the timer resources of the processor. We will investigate both of these two methods of creating a delay function in this project.

Pacing the computer

When the processor executes code faster than the application requires, microprocessor based systems have time to spare. Say, for example, there is an application where an LED is to be toggled between on and off at a one-second rate. The LATGINV instruction, used to toggle the IO pin, only requires four processor clock cycles accounting for 50ns of the desired one second period. Knowing that the microprocessor always wants to be executing the next line of code at a rate of 80 million instructions per second, additional instructions must be executed to implement a time delay operation. In most cases, the time delay function has no other purpose than to kill time.

Two approaches for developing a delay function will be explored in this project: one that requires no special hardware but instead relies strictly upon the execution time of a software loop. We will refer to this approach as the "<u>software</u>" delay method. The other uses one of the processor's built-in timers to generate a delay by counting clock cycles. We call this approach a "<u>hardware-assisted</u>" delay method. Delay functions have two controlling interrelated elements: the resolution and the range. The <u>*resolution*</u> is defined by the smallest possible delay and the <u>*range*</u> by the longest possible delay. To some extent these parameters are determined by the method chosen to implement the delay.

Pure Software Delays

First, let us focus on the software delay method by considering a software loop to generate a delay function. There are three ways to write a software loop: a "for", a "while" loop, and a "do-while" loop. An example of using a "for" loop is: *for (i = 0; i < COUNTS_PER_MS; i++);*. In this program statement, the value of the constant "COUNTS_PER_MS" is chosen to result in a unit delay (delay of one unit) to be one millisecond. Listing 3 shows how to generate a software delay of specified time using a fixed number of "*for*" loop iterations. This "*for*" loop is then repeatedly executed for each desired millisecond of delay using a "while" loop. The speed at which the processor executes code affects value of the constant COUNTS_PER_MS. The larger the nominal value of the constant COUNTS_PER_MS, the higher the resolution of the period hence the more accurately the delay period can be set. The lines of code that are annotated with the comment "// ***SW Stop breakpoint***" identify the lines of code where break points are inserted to allow execution timing using the MPLAB X stopwatch tool. (See Appendix C for information on using the stopwatch.)

Listing 3. Software delay function

```
// LEDA is defined in CerebotMX7cK.h

/* The following define statement should be declared in Project2.h
#define COUNTS_PER_MS   1000        // Initial guess
*/

void sw_msDelay (unsigned int mS)
{
int i;
   while(mS --)         // SW Stop breakpoint
   {
      for (i = 0; i< COUNTS_PER_MS; i++)  // 1 ms delay loop
      {
            // do nothing
      }
      LATBINV = LEDA;   // Toggle LEDA each ms for instrumentation
   }
}                       // SW Stop breakpoint
```

Given that the "*mS*" variable that specifies the number of milliseconds to delay is an unsigned integer, the range or longest possible delay is 4,294,967,295 ms or 49.71 days. The disadvantage of the software delay loop is that the value assigned to "COUNTS_PER_MS" depends upon the processor speed. If the processor speed changes, then the delay function is no longer calibrated correctly.

Calibrating Time for Software Delays

When using the "*sw_msDelay*" function shown in Listing 3 above, the questions that must be answered are "What value should be used for COUNTS_PER_MS and how accurate is the delay?" Subsequently, you must address the issue as to how best to determine the value to be used for the COUNTS_PER_MS constant. As previously suggested, it is difficult to simply accumulate the instruction times hence an alternate method will be used.

We suggest that software code be added to allow the software operation to be observable. One way to accomplish this is by toggling an IO pin on the microprocessor. This requires that external instrumentation be connected to the pin of the board connected to that IO port for the bit being toggled. Suitable instrumentation for this test includes frequency counters, oscilloscopes, and logic analyzers. Now one must consider the appropriate place to insert the code to set the IO pin high or low realizing that inserting of such code may affect the accuracy of the delay function. It is advisable to place the instrumentation code where it will be executed the minimum number of times. It should be noted that the LATxINV instruction is used to toggle specific bits in the LAT register without modifying the other LAT register bits. This is an important consideration when the IO pins for a specific port are used to control independent outputs and different times. A detailed process for determining the value of COUNTS_PER_DELAY is presented in Appendix B.

An effective alternative for measuring delay time is to use the MPLAB ® X Stopwatch tool described in Appendix C.

Hardware-Assisted Delays

The hardware-assisted delay approach to generating a delay function makes use of one of the processor hardware timers. For timer-based delays, the delay amount is often specified as an integer number of CPU clock cycles, timer increments ("ticks"), or time units (e.g., milliseconds). The Cerebot MX7cK board uses an 8MHz crystal. That in combination with programmable PIC32 multipliers and dividers results in the PIC32MX795F512 processor capable of operating at the maximum core frequency of 80MHz. In the *config_bits.h* file, the statement *#pragma config FPLLIDIV=DIV_2* first divides the frequency of the 8 MHz oscillator by 2. The statement *#pragma config FPLLMUL= MUL_20* then multiples the output of the input divider by 20 resulting in 80MHz. Finally the statement *#pragma config FPLLODIV = 1* divides the 80MHz by 1 resulting in the core frequency of 80MHz. Since the power that a processor consumes is related to the processor speed, the combinations of multiplier and dividers allow the developer to set the core frequency that best suits the timing and power requirements of a particular application. Refer to the Section 6 of the PIC32MX5XX/6XX/7XX technical reference for information concerning configuration options for the core oscillator.

The PIC32MX processor family has several internal timers. For this project, we will use the core timer to generate delays of arbitrary periods of time. **The core timer ticks once for every two times the system clock does.** Therefore, the number of core oscillator counts per millisecond equals one half the system clock frequency divided by 1000. The first three *#define* statements in Listing 4 establish the number of core oscillator cycles in a millisecond. The "while" loop in the "hw_msDelay" function shown in Listing 4 implements the total millisecond delay operation. *(Note: #define statements that are already declared in CerebotMX7cK.h should not be repeated.)*

Listing 4. Hardware-assisted delay function

```
/* The following define statements are declared in CerebotMX7cK.h
#define GetSystemClock()       (80000000ul)      // Hz
#define GetInstructionClock() (GetSystemClock()/2)
#define CORE_MS_TICK_RATE      (GetInstructionClock()/1000)
#define LEDA                   BIT_2             // IOPORT B
*/

void hw_msDelay(unsigned int mS)
{
unsigned int tWait, tStart;
     tStart=ReadCoreTimer();   // Read core timer count - SW Start breakpoint
     tWait= (CORE_MS_TICK_RATE * mS); // Set time to wait
     while((ReadCoreTimer() - tStart) < tWait);   // Wait for the time to pass
     LATBINV = LEDA;           // Toggle LED at end of delay period
}                                  // SW Stop breakpoint
```

In Listing 4, the start time is immediately taken directly from the core timer using the *ReadCoreTimer()* function. The variable *“mS”* passed to hw_msDelay() is the total number of milliseconds to wait. The variable, *tWait,* is assigned to the number of ticks per millisecond multiplied by the number of milliseconds to wait, *mS*. The current core timer value is continuously read until the core timer has advanced beyond the value computed for tWait. Repeatedly reading the core timer value is referred to as “polling” the core timer.

The delay period resolution of the hardware-assisted delay method is established by the core oscillator frequency. The range of delay is limited by the largest value that an unsigned integer data type used for the variable, *tWait*, can represent. For an 80MHz system frequency, this results in (2^{32} / CORE_MS_TICK_RATE) milliseconds = (2^{32} /40,000) ms that equals 107,374 ms or 1.79 minutes.

Project Tasks

We will be referencing the instrumentation test points provided on the PmodSTEP module (Appendix A) for this project. Specifically, we will use the test points associated with LEDA and LEDB as identified on the parts layout shown in Appendix A. You are to write a single program with the two methods of time delay. The program is compiled so that only one of the two time delay methods is used at a time. For the program using the code in Listing 3, you must determine an appropriate value for the COUNTS_PER_MS constant that is defined in Project2.h. Hint: try starting at 5000. The Project2.h program will look like Listing 5 and Project2.c like Listing 6. No other bits should be altered when toggling pins RB2 and RB3 for timing purposes. Determine the relative accuracy for delays in different duration of the delay over the range of 1ms to 1000ms.

Listing 5.

```
/********************** Project 2 ********************************
 *
 * File:           Project2.h
 * Author:         Richard Wall
 * Date:           May 22, 2013
 *
*/

/* Software timer definition */
#define COUNTS_PER_MS   5000  /* Exact value is to be determined */

/* Function Prototypes */
void system_init (void);            /* hardware initialization */
void sw_msDelay (unsigned int mS);  /* Software only delay */
void hw_msDelay(unsigned int mS);   /* Hardware-assisted delay */
```

Listing 6.

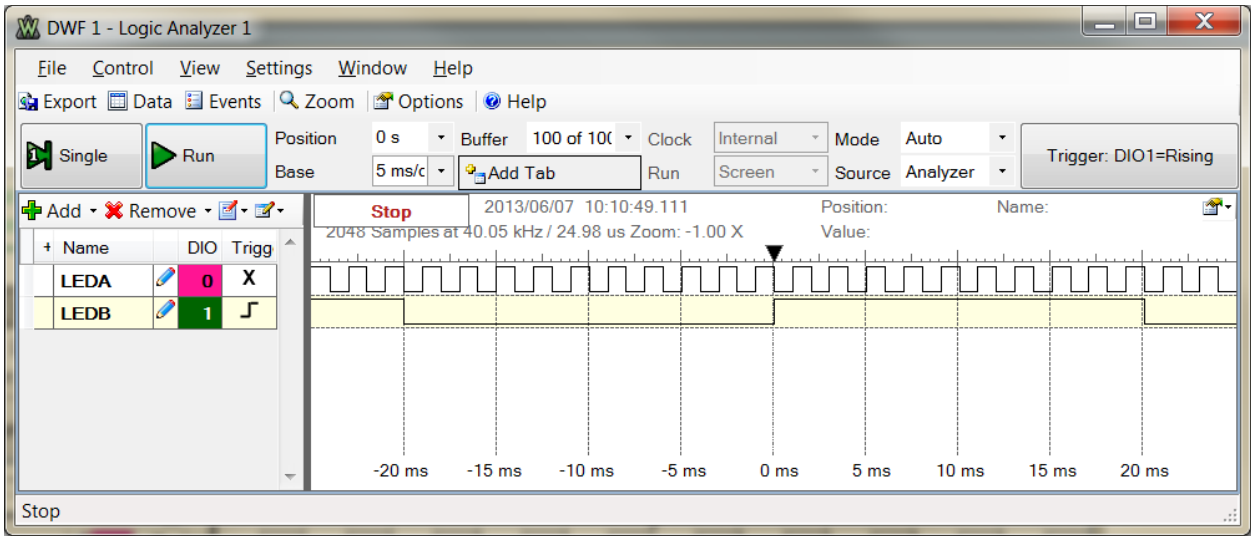
```
/******************************** Project 2 ****************************
 *
 *  File:         Project2.c
 *  Author name:  Richard Wall
 *  Rev. Date:    May 22, 2013
 *
 *  Project Description:  The purpose of this project is to investigate
 *  the characteristics and limitations of two types of polling delay.
 *
 ***********************************************************************/

#include <plib.h>
#include "CerebotMX7cK.h"
#include "Project2.h"

int main()
{
int mS = 1;              /* Set total delay time – change as needed */
    system_init ();      /* Setup system Hardware. */
    while(1)
    {
      LATBINV = LEDB;    /* Toggle LEDB each delay period */
/* Run with only one of the two following statements uncommented */
      sw_msDelay (mS);   /* Software only delay */
//    hw_msDelay(mS);    /* Hardware-assisted delay */

    }
    return 0;      /* Returning a value is expected but this statement
                    * should never execute */
}

/* system_init FUNCTION DESCRIPTION **************************************
* SYNTAX:          void system_init (void);
* KEYWORDS:        initialization system hardware
* DESCRIPTION:     Sets up the configuration for Port B to control LEDA
*                  - LEDH.
* RETURN VALUE:    none
* END DESCRIPTION ********************************************************/
void system_init(void)
{
// Setup processor board
     Cerebot_mx7cK_setup();
     PORTSetPinsDigitalOut(IOPORT_B, SM_LEDS);/* Set PmodSTEP LEDs outputs */
     LATBCLR = SM_LEDS;            /* Turn off LEDA through LEDH */
}
```

```
/* sw_msDelay (mS) Function Description **********************************
* SYNTAX:           void sw_ms_delay(unsigned int mS);
* DESCRIPTION:      This is a millisecond delay function that will repeat
*                   a specified number of times. The constant "COUNTS_PER_MS"
*                   must be calibrated for the system frequency.
* KEYWORDS:         delay, ms, milliseconds, software delay
* PARAMETER1:       mS - the total number of milliseconds to delay
* RETURN VALUE:     None:
* Notes:            The basic loop counter "COUNTS_PER_MS " is dependent on
*                   the CPU frequency. LEDA will toggle at 500 Hz.
*END DESCRIPTION ********************************************************/
void sw_msDelay (unsigned int mS)
{
      /* Use code from Listing 3 */
}

/*hw_msDelay Function Description ******************************************
* SYNTAX:           void hw_msDelay(unsigned int mS);
* DESCRIPTION:      This is a millisecond delay function uses the core time
*                   to set the base millisecond delay period. Delay periods
*                   of zero are permitted. LEDA is toggled each millisecond.
* KEYWORDS:         delay, ms, milliseconds, software delay, core timer
* PARAMETER1:       mS - the total number of milliseconds to delay
* RETURN VALUE:     None:
* END DESCRIPTION *******************************************************/

void hw_msDelay(unsigned int mS)
{
      /* Use code from Listing 4 */
}
// End of Project2.c
```

Project Testing

Use the *LATBINV = LEDA;* and *LATBINV = LEDB;* instructions to toggle the LEDs for to instrument the two different delay programs. Connecting an oscilloscope or logic analyzer to the test point for LEDB (RB3) will allow you to measure the total delay being called for from the main function. Connecting the oscilloscope / logic analyzer probe to the test point for LEDA (RB2) will measure the millisecond *for* loop delay but only for the *sw_msDelay* function. (Remember to connect the oscilloscope or logic analyzer ground to the PmodSTEP ground pin. See Appendix A for the pin locations.)

Figure 1 is an example of the instrumentation to determine the timing for a 20 ms delay. The LEDA trace toggles each 1ms resulting in a frequency of 500Hz. The LEDB trace toggles each 20ms resulting in a frequency of 25Hz.



Position 0 s
Base 5 ms/c
Buffer 100 of 10(
Add Tab
Clock Internal
Run Screen
Mode Auto
Source Analyzer
Trigger: DIO1=Rising

Stop 2013/06/07 10:10:49.111
2048 Samples at 40.05 kHz / 24.98 us Zoom: -1.00 X
Position:
Value:
Name:

Name	DIO	Trigg
LEDA	0	X
LEDB	1	⌐

Stop

Figure 1. Screen capture for 20 ms delay

Complete the data entries for Table 1 using the signal provided by instruments connected to the test point for LEDB.

Table 1. Delay Performance Chart

Design Total Delay – ms	Software Delay- Measured - ms	Hardware Delay- Measured – ms
	LED B	LEDB
1		
10		
100		
1000		

Appendix A: PmodSTEP Parts Layout

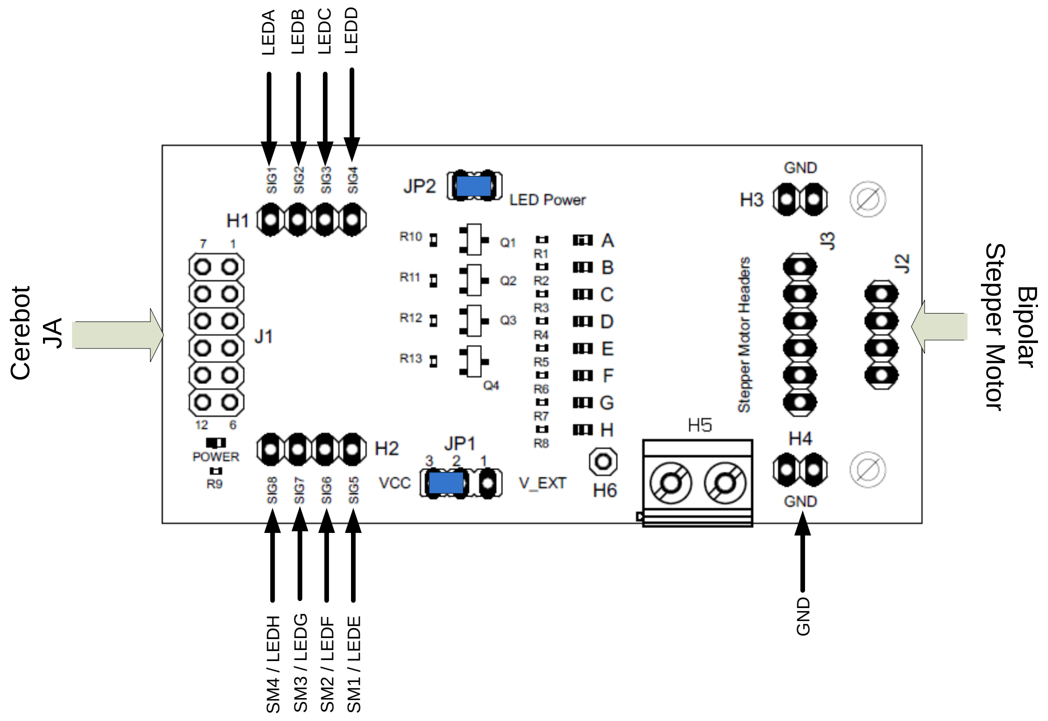


Figure 2. PmodSTEP Stepper Motor Driver Module parts layout

Table 2 LED – Pmod Port JA Pin and PIC32MX7 IO Port B Assignments

LED	JA Pin #	PIC32MX7 Port B Pin #
LED A	1	RB2
LED B	2	RB3
LED C	3	RB4
LED D	4	RB6
LED E / SM1	7	RB7
LED F / SM2	8	RB8
LED G / SM3	9	RB9
LED H / SM4	10	RB10

Appendix B: Process for Determining the Software Delay Constant

Assume that we want to have the ***for*** loop generate a one millisecond delay. The time to execute a software delay loop is proportional to the number of times the loop must be executed as expressed in Eq. 1.

T_{DELAY} = COUNTS_PER_MS * $T_{EXECUTION\_TIME\_SINGLE\_LOOP}$ Eq. 1

Given that the single loop execution time is constant then the relationship expressed in Eq. 2 is also true:

T_{DELAY_1} / T_{DELAY_2} = COUNTS_PER_MS_1 / COUNTS_PER_MS_2 Eq. 2

T_{DELAY_1}:	Actual delay time measure by monitoring the test point for LEDA
T $_{DELAY\_2}$:	The desired loop delay
COUNTS_PER_MS_1:	Initial or previously computed number of for loop iterations to generate a one millisecond delay.
COUNTS_PER_MS_2:	Updated computed value of COUNTS_PER_MS

Suppose that using a particular value for COUNTS_PER_MS_1 in our software delay function results in a delay (T_{DELAY_1}) that is not equal to our desired 1ms delay (T_{DELAY_2}). Rearranging Eq. 2 allows us to solve for an updated COUNTS_PER_MS needed to implement the desired delay as shown in Eq. 3.

COUNTS_PER_MS_2 = (T_{DELAY_2} / T_{DELAY_1}) * COUNTS_PER_MS _1 Eq. 3

We can start with any initial guess for COUNTS_PER_MS _1 provided it is in the numerical range of an unsigned integer. After running the program using our initial guess we then take the measurement to determine the actual delay, T_{DELAY_1}. Using the ratio of the desired delay to the actual delay and multiplying by the value we used for COUNTS_PER_MS_1 allows us to compute the new value for COUNTS_PER_MS_2.

In theory the new value computed for COUNTS_PER_MS_2 will result in the desired delay. However, the proportionality of Eq. 1 is only approximate and it may require a few iterations of solving Eq. 3 to achieve an accurate delay. The solution has converged whenever the measured delay matches the desired delay. The following procedure can be used to determine the correct value of COUNTS_PER_MS to achieve a one millisecond loop delay.

Step 1: Generate Project1 using the code provided by Listing 3 through 6. Be sure to have the common header files *config_bits.h* and *CerebotMX7cK.h* as well as *CerebotMX7cK.c* added to the project. Uncomment the *sw_msDelay(mS)* statement in the *main* function and set the initial value of mS to 1 so a one millisecond delay is implemented. Select an initial value for COUNTS_PER_MS_1. (For a one ms delay, a good number to try first is 5000.) Modify

Project2.h to set the initial value for COUNTS_PER_MS equal to the value of COUNTS_PER_MS_1.

Step 2: Compile and execute the C program for Project 2 with no breakpoints set.

Step 3a: Measure the high or low interval of the square wave generated at the LEDA test point using an oscilloscope or logic analyzer. (Remember to connect the oscilloscope common to the project board ground pin.) Record this measured period for the value of T_{DELAY_2}.

Step 3b: An alternate method uses a frequency meter. Measure the frequency of the square wave observed with a frequency meter that is connected at the LEDA test point. (Remember to connect the meter common to the project board ground pin.) Record the measured frequency as $F_{MEASURED}$. Compute the delay period from Eq. 4.

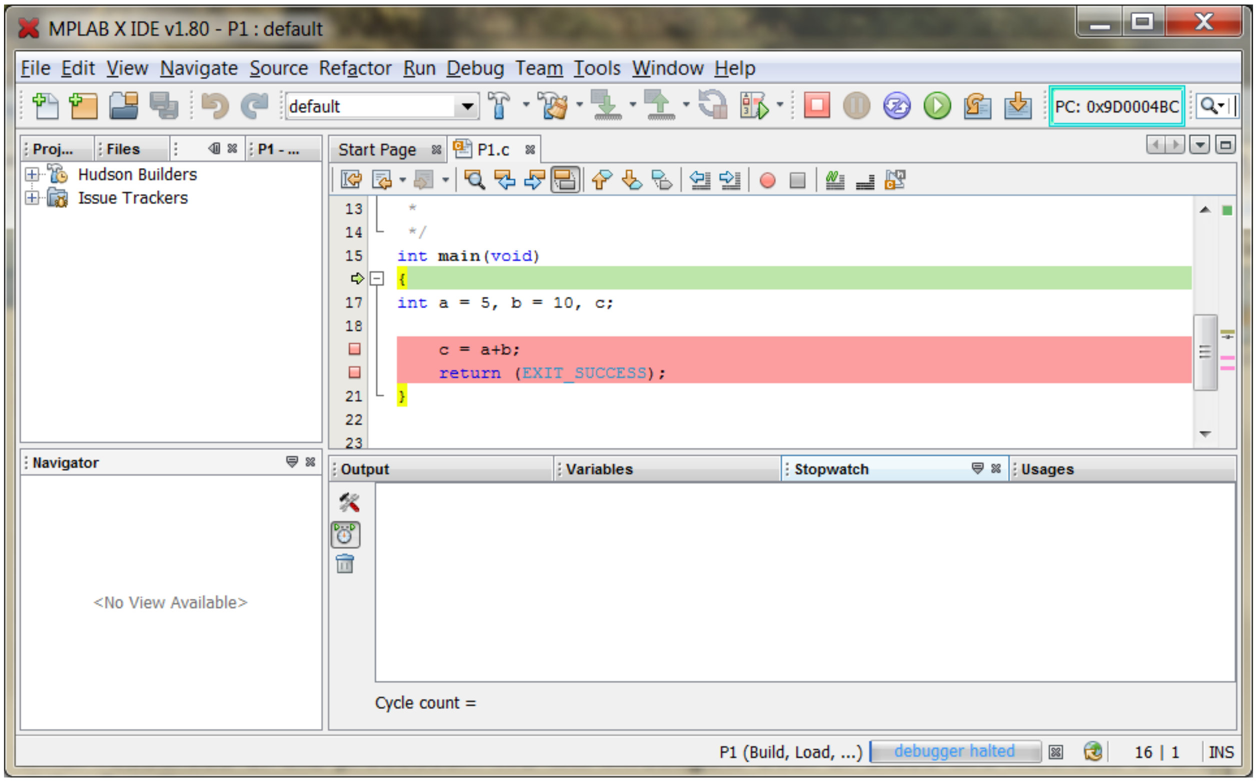
$$T_{DELAY\_2} = 1/(2 * F_{MEASURED}) \qquad \text{Eq. 4}$$

Step 4. Setting T_{DELAY_1} equal to 1ms and using the T_{DELAY_2} found in Step 3a or Step 3b along with the value COUNTS_PER_MS_1, solve for COUNTS_PER_MS_2 using Eq. 3.

Step 5: Modify Project2.h to set the value for COUNTS_PER_MS equal to COUNTS_PER_MS_2. If the T_{DELAY_2} is not equal to 1ms, assign the value of COUNTS_PER_MS_1 equal to the value of COUNTS_PER_MS_2 computed in Step 4 and repeat Steps 2 through 5 again. Continue repeating Steps 2 through 5 until the desired delay is achieved to the desired degree of accuracy possible given the resolution of COUNTS_PER_MS is plus or minus one count.

Appendix C: MPLAB X Stopwatch

Open the stopwatch window by selecting *Window->Debugging->Stopwatch* on the MPLAB ® X task bar. Using the source code window, generate two break points, one on line 19 and the other on line 20, as shown in Figure 3. Next select the properties icon on the top right of the Stopwatch window. (This icon is a hammer and a wrench.)

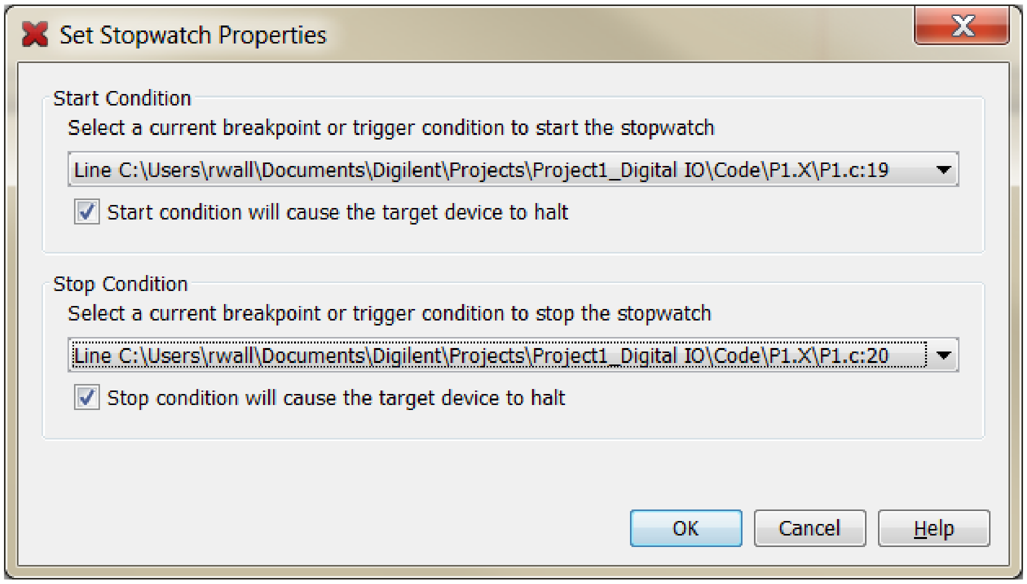


```
13      *
14      */
15   int main(void)
     {
17   int a = 5, b = 10, c;
18
        c = a+b;
        return (EXIT_SUCCESS);
21   }
22
23
```

Cycle count =

Figure 3. MPLAB X IDE with two breakpoints and the Stopwatch window.

On the Stopwatch properties window, the two dropdown boxes on the right allow you to select the line of code that the stopwatch will be started and the line of code where the stop watch will be stopped. Figure 4 show the results of selecting line 19 to start the stopwatch and line 20 to stop the stopwatch. After selecting the start and stop conditions, select the *OK* box. The MPLAB screen will look like Figure 3 once more. Select the *Continue* control until the execution stops at line 20. The stopwatch cycle count is displayed in the Stopwatch window as shown in Figure 5.



Set Stopwatch Properties

Start Condition

Select a current breakpoint or trigger condition to start the stopwatch

Line C:\Users\rwall\Documents\Digilent\Projects\Project1_Digital IO\Code\P1.X\P1.c:19

☑ Start condition will cause the target device to halt

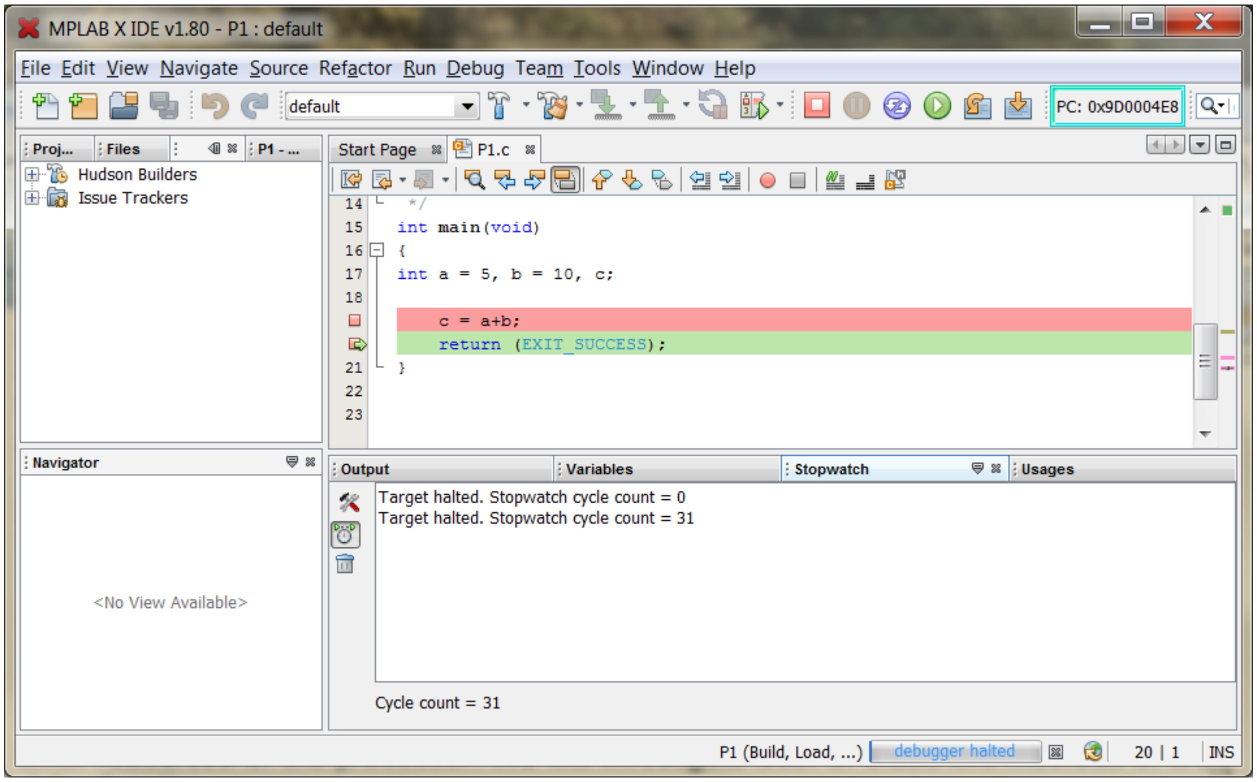
Stop Condition

Select a current breakpoint or trigger condition to stop the stopwatch

Line C:\Users\rwall\Documents\Digilent\Projects\Project1_Digital IO\Code\P1.X\P1.c:20

☑ Stop condition will cause the target device to halt

Figure 4. Setting the Stopwatch properties



MPLAB X IDE v1.80 - P1 : default
File Edit View Navigate Source Refactor Run Debug Team Tools Window Help
default
PC: 0x9D0004E8
Proj... Files P1 - ...
Hudson Builders
Issue Trackers
Start Page P1.c
Navigator
<No View Available>
Output Variables Stopwatch Usages
P1 (Build, Load, ...) debugger halted 20 | 1 INS

```
 */
int main(void)
{
int a = 5, b = 10, c;

    c = a+b;
    return (EXIT_SUCCESS);
}
```

```
Target halted. Stopwatch cycle count = 0
Target halted. Stopwatch cycle count = 31
```

Cycle count = 31

Figure 5. MPLAB X screen after executing code to line 20

Appendix D: An Application of Time Delays

Contact de-bouncing: Momentary contact push buttons are inexpensive and commonly used mechanical switches that either makes connection or breaks connection between its two terminals. Normally-open momentary contact push buttons make an electrical connection between two terminals when pushed or operated. A spring returns the button to the normal state and breaks the connection when released. The opposite action is true for normally-closed momentary contact push buttons. The simple push button is more than adequate for many slowly operating applications such as ringing a door bell or sounding a horn.

If a push button is connected to a microprocessor input pin, then we must consider how the contacts are actually operating. When switch is pressed there is short period of time where oscillation of opening and closing the contact occurs due to the spring-mass characteristics of the mechanical contacts. This phenomenon is called *contact bounce* or *switch bounce*. The duration of the bounce period is random and can range from microseconds to hundreds of milliseconds. Signal conditioning circuits or computer firmware remove these unwanted signal characteristics. Figure 6 shows the typical signal seen by the microprocessor after the internal signal conditioning when the push button switch is connected as shown in Figure 7. Depending on the speed that the IO pin is polled, it is possible that multiple events are detected instead of the single push operation. A search of the literature soon discloses that there are a plethora of digital and analog electronic circuits specifically designed to remove the multiple contact closures when the switch is activated and when it is de-activated. Equally numerous are the segments of VHDL and microprocessor code to provide the signal conditioning using firmware. Fundamentally, the required signal conditioning is that of a low pass filter. This signal conditioning suppresses the high frequencies generated by the contact bounce while retaining the low frequency signal generated by the press and release operations.

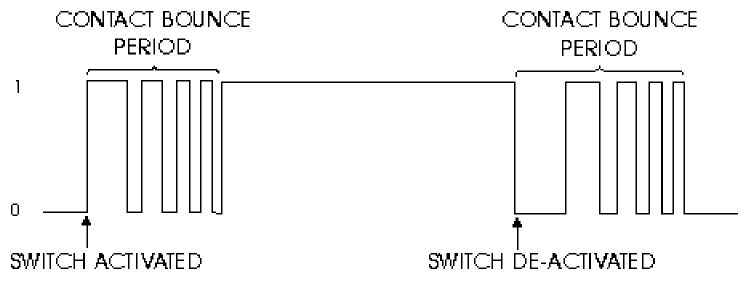


Figure 6. Ideal voltage plot of an active high push button operation showing the results of contact bounce

Buttons that make the connection between two terminals when pressed are called Push-ON switches and buttons that breaks the connection between two terminals are called push-OFF. Figure 7 through Figure 10 show the possible connections for the push-ON and push-OFF switches to generate active high or active low signals. The push buttons on the Cerebot MX7cK processor board are configured similar to the circuit shown in Figure 7.

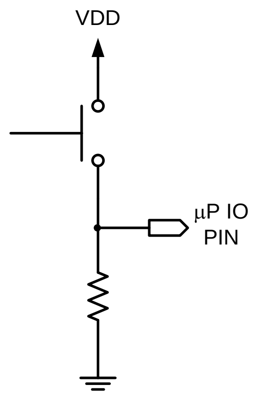


Figure 7 Active high connection of a normally open momentary push button switch

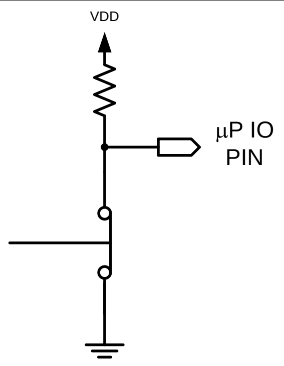


Figure 8. Active high connection of a normally closed momentary push button switch

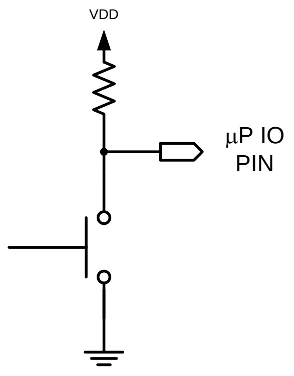


Figure 9. Active low connection of a normally open momentary push button switch

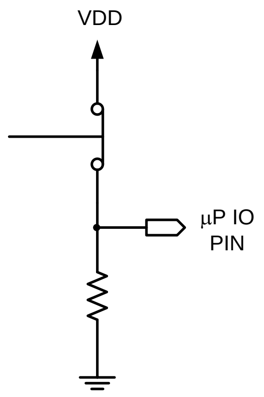


Figure 10. Active low connection of a normally closed momentary push button switch

A computer program can behave differently depending upon the duration of the contact bounce, the speed at which an IO pin is polled, and exactly when the IO pin is polled. Under unfavorable conditions the input can behave as though multiple events occurred when only one event is intended. One approach to avoid this problem is to poll the input pin multiple times over an appropriate period of time that extends beyond the expected contact bounce period. The logic value resulting from successive polling operations can be repeatedly compared until the IO pin readings match. A flow diagram of computer code to implement this operation is shown in Figure 11. While this approach has proven generally effective it is by no means guaranteed to eliminate all of the effects of contact bounce on a computer program. More elaborate schemes must be used if the mechanical switch is to be employed in critical applications where sperious multiple contact closures cannot be tolerated.

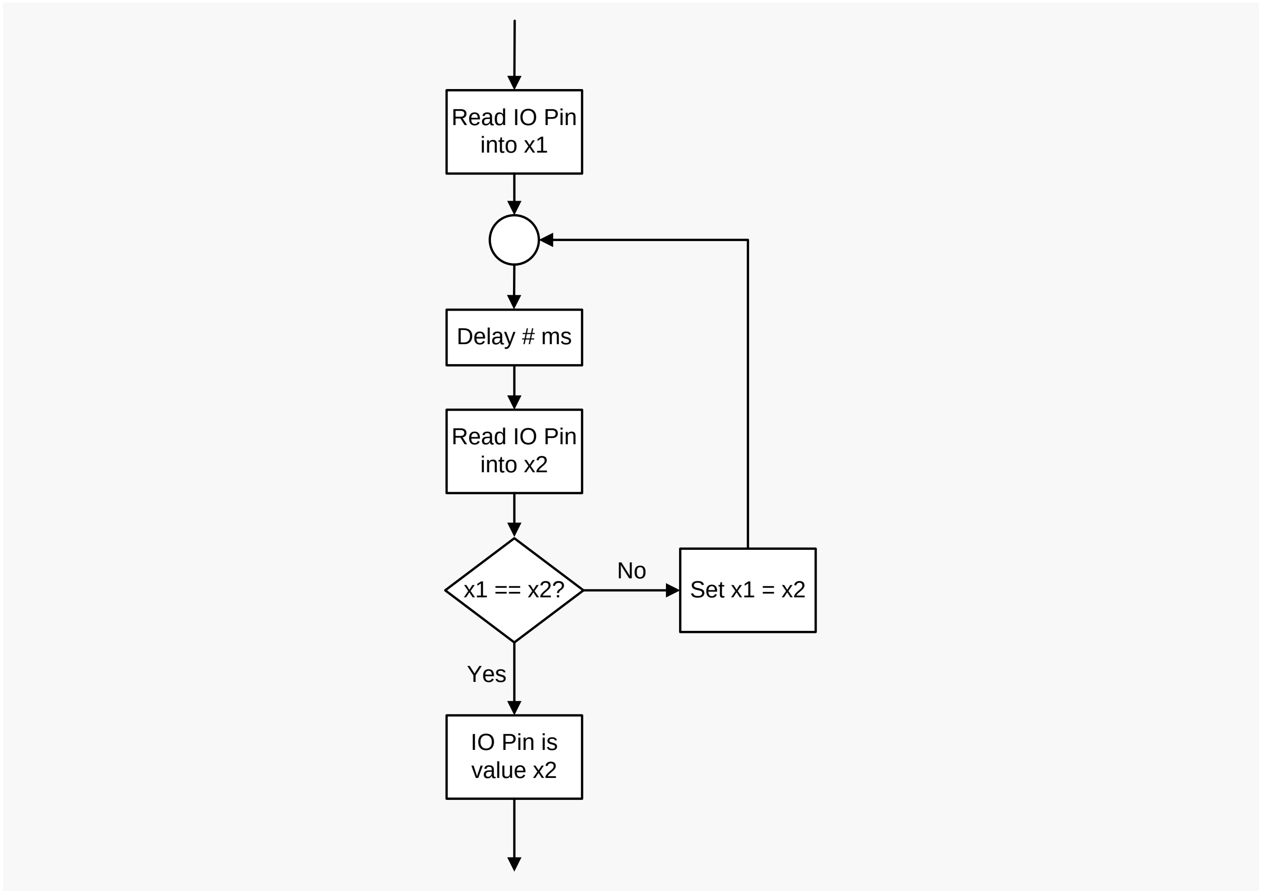


Figure 11. Program control flow diagram for using time a delay to mitigate contact bounce.

There are two problems that must be addressed in the simple solution to contact bounce proposed in the flow diagram shown in Figure 11. The initial problem to be addressed is what to specify as an appropriate delay period to filter out all but an acceptable number of multiple event detections due to contact bounce. Longer time delays are better but if the delay is too long, the event can be missed due to a short button press (lack of signal persistence). The second problem is perceived response time. Ganssle states in his tutorial that 100 ms is within the human response time. A more appropriate computer response is to report the event immediately when the button state change is first detected but not report a button release until it occurs after the appropriate delay time. Such a solution is beyond the scope of this project but is addressed again in Project 5 when we discuss the use of interrupts.

[1] Jack Ganssle, "A Guide to Debouncing, or, How to Debounce a Contact in Two Easy Pages", http://www.ganssle.com/debouncing.htm

[2] http://www.mcuexamples.com/push-buttons-and-switch-debouncing-with-PIC.php

[3] Jack Ganssle, "My favorite software debouncers", http://www.embedded.com/electronics-blogs/break-points/4024981/My-favorite-software-debouncers