OREGON INSTITUTE OF TECHNOLOGY

**Computer Systems Engineering Technology Department**

***CST 204 - Introduction to Microcontrollers***

**Lab 4 – Introduction to Timers**

**Name: Michael Kearton Score \_\_\_\_\_\_\_\_\_\_**

# Introduction to Lab 4

In lab 4, you will generate waveforms using delays; instruction delays and then delays using Timer 1. You have already done this for the “heatbeat” function using the real-time executive (RTE), but this time you will do so without the RTE. The purpose is to only investigate the timer. You will continually generate a pattern, setting the PORTA LEDs to 0xa5 and 0x5a. Pictorially, this is what you generated.

0xa5 (delay) 0x5a (delay) 0xa5

In addition, an *individual* signal, such as RA7 looks like this:

(signal change) (signal change)

(delay) (delay)

RA7

Clearly, the length of the delay will alter the shape of the waveform. Furthermore, the delay time using an instruction-based loop delay is a function of how many instructions are executed in the loop. Since the instruction execution time is dependent on the **CPU clock period and CPU wait states**, i.e. the time per instruction, hence, the overall delay is closely related to the number of loop instructions multiplied by the clock period. Note: The MIPS core in the PIC32 executes at one instruction per clock period, however, by **default there are 7 wait states** added per instruction. As a reminder, the ***clock period is the inverse of the clock frequency***. Yes, there are some “extra” instructions to set up the loop, to perform a subroutine call/return, etc. However, as delay times get longer, the effect of these “overhead” instructions become negligible. We will assume such for this lab.

Another way to generate time delays or time spans with a microcontroller is to use a ***timer***. Conceptually, the timer, 1) is a digital logic counter that has a clocking mechanism, 2) has a way to compare the current count to a preset count value, and 3) indicates a match of the two values. It can also have a structure to “gate” (allow or suspend) the count mechanism. In this lab, we will ignore the *gate* function.

The timer is used for exactly the same purpose as the instruction-based delay case mentioned above – create time delays. The only difference is that the timer-based delay will use a hardware timer indication rather than an instruction loop termination. Also, the time base for the timer’s digital logic counter hardware is the **peripheral bus clock period** (PIC32’s **PBCLK**) which could vary from the **CPU clock period** (PIC32 **SYSCLK**) due to a post-scaler (PBDIV in the OSCCON SFR). This is effectively a clock divider. In addition, the PBCLK to the timer could also be scaled or divided before it is used to count.

# Description for Lab 4

Lab 4 has three parts. You will use three independent source files in three independent projects for each part. They will contain standalone functions without the RTE.

* Part 1 will use **instruction-based** delays.
* Part 2 will use **Timer 1** to generate the time spans with default clock and wait state configurations.
* Part 3 will **scale Timer 1’s clock source** (PBCLCK with adjusted PBDIV).

# Part 1: Instruction-based delays

* 1. Create a **z:\cst204\labs\lab4** folder.
  2. Create a project called **lab4i**. (“i” for instruction)
  3. Create a source file named **lab4i.S** in the **lab4i.X\source** folder. It will contain **two functions**:

**main:**

1. Initialize **LATA<7:0>** latch to **0xa5**. Configure **PORTA<7:0> pins** (these control the LEDs) to **non-open drain outputs**. This requires initializing the **ODCA** and **TRISA** SFRs appropriately.
2. i. Output value **0xa5** to **PORTA**

ii. Call **instr\_delay** function with argument value **62500**.

iii. Output value **0x5a** to **PORTA**

iv. Call **instr\_delay** function with argument value **62500**.

Loop continuously to repeat the above four actions (i-iv)

1. To modify **PORTA<7:0>**, you should use **PORTACLR** to clear all eight bits then **PORTASET** to set the **0xa5** or **0x5a** value. However, *in this case, for this specific data set*, you can also do this by initializing **PORTA<7:0>** to **0xa5** and use the **PORTAINV** to invert the **PORTA<7:0>** bits.

**instr\_delay:**

1. The primary loop code is shown below.

* It shall receive one argument in **a0**, the **loop count** parameter value.
* The primary loop code is shown here. **You need to make this code into a subroutine**.

**loop:**

**nop**

**addi a0, a0, -1**

**bne a0, zero, loop**

**nop**

* 1. Perform the following analysis tasks.

a) Build and program the lab4i project code. Verify that it “blinks” the LED patterns.

b) Determine the **CPU clock rate (frequency)** and **CPU clock period**:

* Refer to **Figure 4-1 in the PIC32 Data Sheet**.
* The default oscillator source is the “**FRC Oscillator 8 MHz typical**” oscillator (Fast Internal RC) that follows the **FRCDIV** path through the tall MUX toward the right. This FRCDIV signal will source the **SYSCLK** (source of the **CPU** clock) as well as the **PBCLK** through the “**Postscaler div x**” block. This selection is indicated by the **COSC<2:0> bits** (Current Oscillator Selection bits) in the **OSCCON Register**, bits <14:12>. Refer to REGISTER 4-1 on page 57 (.pdf pg. 59) of the PIC32 Data Sheet. Furthermore, these COSC<2:0> bits are acquired from the **FNOSC<2:0>** bits in the **DEVCFG1** Device Configuration register that is written at programming time.

**ACTION:** Debug and break anywhere in the main loop. View the “Window/PIC Memory Views/Peripherals” window in MPLAB. Hover over the OSCCON SFR name (the bit definitions should match the PIC32 Data Sheet) and determine the **binary value of the COSC<2:0> bits** (Current Oscillator Selection bits). The bit numbering is starting from the LSB position.

**COSC<2:0>** bits: **0b111** (binary)

* The FRC oscillator frequency (8 MHz) is divided by the “Postscaler” block. This division is selected by the **FRCDIV<2:0> bits** (Fast Internal RC Clock Divider bits) in the OSCCON register, bits <26:24>.

**ACTION:** View the “Window/PIC Memory Views/Peripherals” window in MPLAB. Hover over the OSCCON SFR name (the bit definitions should match the PIC32 Data Sheet) and determine the **binary value of the FRCDIV<2:0> bits** (Fast Internal RC Clock Divider bits).

**FRCDIV<2:0>** bits: **0b001** (binary)

Does it match the default value specified in the Data Sheet? 🞏 YES or 🞏 NO



* Using the information above, determine the **SYSCLK** (and CPU clock) frequency and period.

**ACTION:** Calculate the **SYSCLK** frequency: **4** **MHz**

**ACTION:** Calculate the **SYSCLK** period: **250** **ns**

* Determine the default PBLCK frequency and period. This is determine by the SYSCLK and the “**Postscaler div x**” block. This division is selected by the **PBDIV<1:0> bits** (Peripheral Bus Clock Divisor bits) in the OSCCON register, bits <20:19>.

**ACTION:** View the “Window/PIC Memory Views/Peripherals” window in MPLAB. Hover over the OSCCON SFR name (the bit definitions should match the PIC32 Data Sheet) and determine the **binary value of the PBDIV<1:0> bits** (Peripheral Bus Clock Divisor bits).

**PBDIV<1:0>** bits: **0b11** (binary)

* Using the information above, determine the **PBCLK** frequency and period.

**ACTION:** Calculate the **PBCLK** frequency: **500** **KHz**

**ACTION:** Calculate the **PBCLK** period: **2** **μs**

* CPU instruction execution time depends on the SYSCLK but also the number of wait states programmed. Instruction execution time is (1 + #wait states) \* TSYSCLK. The Cache Control Register (**CHECON**) configures the number of CPU wait states with the **PFMWS<2:0> bits** (Programmable Flash Memory Wait States).

**ACTION:** View the “Window/PIC Memory Views/Peripherals” window in MPLAB. Hover over the CHECON SFR name (the bit definitions should match the PIC32 Data Sheet) and determine the **binary value of the PFMWS<2:0> bits** (Fast Internal RC Clock Divider bits).

**PFMWS<2:0>** bits: **0b111** (binary)

**ACTION:** Calculate the instruction execution time: **2** **μs**

c) Verify that the LEDs “bounce” at 1 Hz. Use a stop watch to measure the delay time.

**Confirmed (Couldn’t record and stopwatch at same time)**

# Part 2: Timer-based delays

* 1. Create a project called **lab4ta**. (“ta” for timer “a” test)
  2. Create a source file named **lab4ta.S**, which will contain **two functions**:

**main:**

* 1. Implement the same as Part 1 with the following additions/modifications:
     1. During initialization/configuration, clear Timer 1 configuration SFR: T1CON = 0x0. This writes ALL bits. Also, clear IFS0<4> = 0 (T1IF).
     2. Curing the loop, call **timer1\_delay\_a 4 times** (i.e. 4 **jal** calls.) with a value of **62500** instead of just 1 **instr\_delay**.

**timer1\_delay\_a:**

1. Take in a 16-bit parameter through a0.
2. Create a subroutine using Timer 1 according to the following algorithm:

* Clear Timer 1 Count value: **TMR1** = 0x0. Write ALL bits at once.
* Load Period Register: **PR1** = a0. Write ALL bits at once.
* Start Timer 1: **T1CON<15>** = 1 (See Register 13-1 T1CON). Set ONLY the ON bit, bit 15.
* Check for Timer 1 status flag to set: Loop and wait for **IFS0<4>** (IFS0 SFR bit 4) to be “1”. This is the **T1IF** bit that indicates that the counter value (TMR1) matched the Period value (PR1). You will need to mask the value read.
* Stop Timer 1: When IFS0<4> = 1 (T1IF), **T1CON<15>** = 0. Clear ONLY the ON bit, bit 15.
* Clear the Timer 1 status flag: **IFS0<4>** = 0 (T1IF). Clear ONLY the T1IF bit, bit 4.
  1. Perform the following analysis tasks.

1. Given the Timer 1 Period Register value to be 62500 and the Timer 1 clock period to be the calculated PBCLK period determined in Part 1, section 4b above, calculate the **timer1\_delay\_a** delay time. Ignore “overhead” instructions and just use the Timer 1 Period Register-to-status-flag-set time.

**ACTION:** Calculated **timer1\_delay** delay time: **0.125** **s**

1. Determine the measured PBCLK frequency and period using the Timer 1 clock data. The Timer 1 clock is determined by dividing the PBCLK according to the Figure 13-1 and the definition for the T1CON register. This division is selected by the **TCKPS<1:0> bits** (Timer Input Clock prescaler Select bits) in the T1CON register, bits <5:4>.

**ACTION:** View the “Window/PIC Memory Views/Peripherals” window in MPLAB. Hover over the T1CON SFR name (the bit definitions should match the PIC32 Data Sheet) and determine the **binary value of the TCKPS<1:0> bits** (Timer Input Clock prescaler Select bits).

**TCKPS<1:0>** bits: **0b00** (binary)

# Part 2: Timer-based delays II

* 1. Create a project called **lab4tb**. (“tb” for timer “b” test)
  2. Create a source file named **lab4ta.S**, which will contain **two functions**:

**timer1\_delay\_b:**

1. Create a subroutine using Timer 1 using the same implementation as **lab4ta.S**.

**main:**

* 1. Configure the Timer 1 clock to be 125 KHz. Do this by configuring the FRC by 64. This can be accomplished by:
     1. Leaving FRCDIV to its default.
     2. Configuring TCKPS = divide by 8. (T1CON SFR)
     3. Configuring PBDIV = divide by 4. (OSCCON SFR) You will need to perform the “unlock sequence” to modify the OSCCON SFR. Do this by using the SYSKEY SFR as follows. You will use **la** and **sw** as usual.
        + 1. SYSKEY = 0
          2. SYSKEY = 0xaa996655
          3. SYSKEY = 0x556699aa
          4. Chance OSCCON
          5. SYSKEY = 0
  2. Implement the same as Part 1 with the following additions/modifications:
     1. Clear Timer 1 configuration SFR: T1CON = 0x0. This writes ALL bits.
     2. Call **timer1\_delay\_b only 1 time** with a value of **62500**.
  3. Verify 1 Hz LED “bounce”.

**Confirmed (noticeably slower than parts 1 & 2)**

# Other Considerations for Lab 4

* 1. Create a “lab4” folder and create all projects, store all source files, within the lab4 folder.
  2. Submission requirements will be specified through Blackboard.