Experiment 6: Timers and Interrupts

Objectives

The purpose of this experiment is to understand the operation of PICTM32 timers so that they can be used to implement a synchronized multi-rate periodic control system by polling the timer interrupt flag. Also to explore detecting events using interrupts.

Equipment List

- ChipKIT^{TM} Pro MX7 processor board with USB cable
- Microchip MPLAB ® X IDE
- MPLAB ® XC32++ Compiler

Overview

PIC32 Timers

All timers have a period register that is used to set the timer's maximum count. The period registers are declared as PR1 for Timer 1, PR2 for Timer 2, and so on. Whenever the time count reaches the value stored in the period register, a timer interrupt flag is set and the timer register is reset to zero. Once the timer interrupt flag is set, it will remain set until a software instruction clears the flag.

Timers can be initialized using the Mplab Harmony Configurator (MHC). The MPLAB® Harmony Timer Driver library provides a high-level interface to the PIC32 Timer peripherals. There are two kinds of Timer Drivers: dynamic and static.

The **Dynamic Timer Driver** is a set of API's allowing you to control any PIC32 timer similar to those offered by the Static timer Driver. The Dynamic Driver offers you more options for application management than a Static Driver.

In this lab, we will use the **static Timer Driver** which provides a basic set of timer access functions such as read, write, start, stop ... etc. When a Static Driver is used, the project is configured with unique APIs to access each timer instance. To access a timer, you will call a function whose name is determined by the timer driver instance number. Harmony generates an **interrupt service routine (ISR)** for each timer with a static driver. You will code the application's ISR functionality into the ISRs. These ISR are set during development and cannot be changed at run time.

As shown in Figure 20, static Driver APIs reference the hardware timers by driver instance (e.g. DRV_TMR0) rather than by the timer peripheral number (e.g. Timer2). You will use the MPLAB Harmony Configurator (MHC) to determine which specific hardware peripheral timer is associated with each timer driver instance. Among the required options to configure the Timer Driver are the prescale and the Timer Period. Timers use the peripheral clock for the input of the prescale dividor as shown in Figure 19.

It is easy to configure the timer in Harmony projects. All you have to do is to follow these steps:

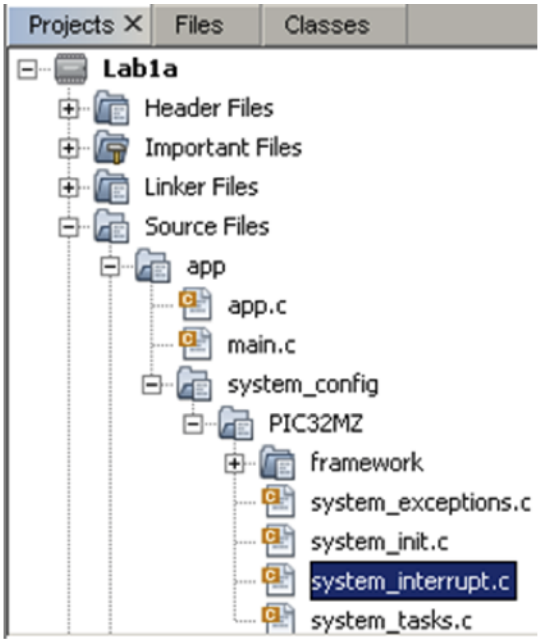

Figure 18: File tree to locate system_interrupt.c

- The first step is to enable and configure the timer driver.
  - In the MPLAB Harmony Configurator (MHC), select the **Option** tab.
  - Expand the Harmony Framework Configuration tree, expand **Drivers**, expand Timer tree.
  - Check the **Use Timer Driver?**
  - Choose **Driver Implementation** : STATIC
  - Check the **Interrupt Mode** box if not already checked
  - **Number of Timer Driver Instances** = 1
  - Select and Expand **TMR Driver Instance** 0 tree option (leave default values for parameters other than below)
  - Set **Timer Module ID** to TMR_ID_2
  - Set **Operation Mode** to DRV_TMR_OPERATION_MODE_32_BIT
  - Set **Timer Period** to an integer value that corresponds to the number of seconds that you desire. Timer2 peripheral is on Peripheral Bus 3 (PBCLK3). Refer to clock diagram on MHC to obtain the PBCLK frequency. The Timer Driver is configured for 32-bit mode and let the prescale setting denoted as PS, we need to calculate a period value that will enable us to achieve the desired interrupt rate (N seconds with a frequency equals to $1/N$) :

$$\frac{PBCLK}{PS * TimerPeriod} = \frac{1}{N} \tag{1}$$

    Set the **Timer Period** to this calculated value.

- The second step is to write code to Start the timer and any other functionalities in your program.
  - For initialization, you can use the following function:

```
DRV_TMR0_Start ( ) ;
```

  - Then, it is necessary to handle the interrupt generated from the Timer each time it ticks. In order to do so, you can use the source file **system_interrupt.c**, as shown in Figure 18.

– Preceding the ISR Handler and below the header files include calls, declare and provide external reference for APP_DATA appData. (The appData declaration is located in app.c file).

– In TIMER3 interrupt function __ISR(_Timer_3_Vector, ...) before the existing function which already clears the TIMER interrupt flag. Add code to Update the application state to design a correct logic of your program (this state will be used in app.c).

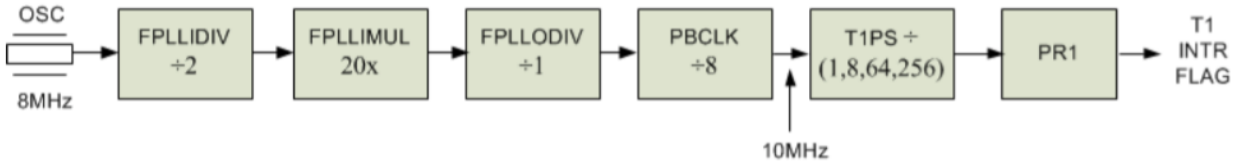

Figure 19: Divider chain from the CPU crystal (oscillator) to the Timer 1 interrupt flag..

Each timer is controlled by its own set of functions. The function names for the API contain the specific timer instance of the timer being accessed. For example, the function to enable the timer x to run is:

DRV_TMRx_Start ( )

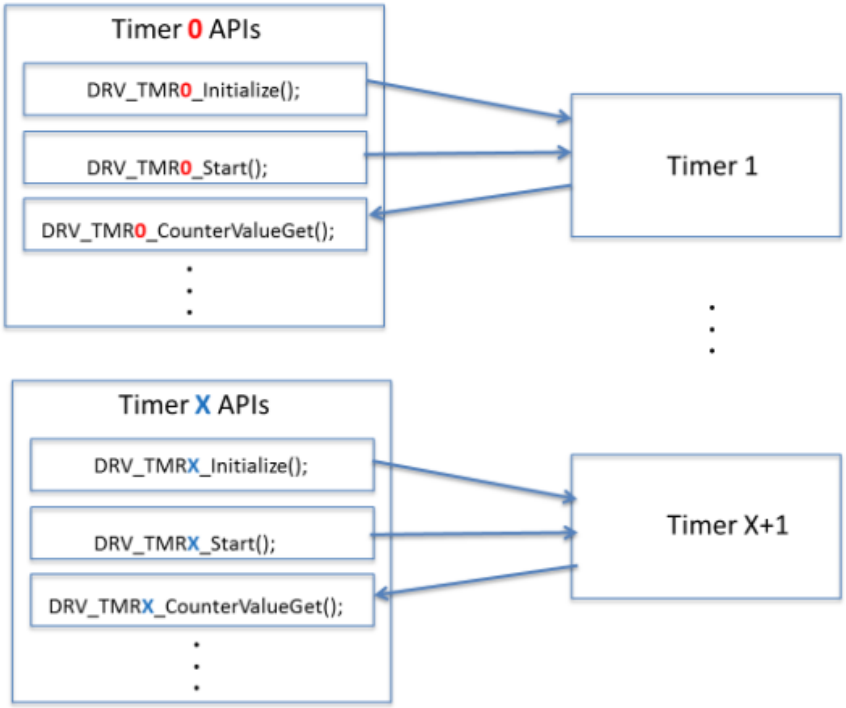

Figure 20: Static Driver API Assignment.

PIC32 Interrupts

An interrupt is a signal triggered by an event. The computer suspends what would normally be the next instruction of a task in one part of a program and begins executing code to complete an entirely different task.

There are four essential code elements required for a program to process interrupts using C:

- the declaration of the functions that will be used to service the interrupts,

- the code to initialize the resources that generate interrupts,
- the ISR code that will be executed in response to an interrupt,
- the instructions that enable interrupts in a global sense.

The initialization and the enabling functions of interrupts are automatically inserted in the system_init.c file by the MHC when you choose to use the interrupt mode of Timer Driver. Functions that have been declared as an ISR cannot be called by any other C function. There are two ways that the ISR code will be executed: either in response to the event that sets the interrupt flag through hardware or by setting the corresponding bit in the interrupt flag register using a software instruction. A function that is declared as an ISR cannot have any variables passed to it (no argument list) and must return a void data type.

For each timer configured with a Static Driver, Harmony inserts code to create an interrupt service routine (ISR) and instructs the linker to place the ISR at the appropriate vector address. As shown in Figure 21, inserted into these ISRs is code to clear the relevant interrupt request flag (Failing to clear the interrupt will cause the processor to repeatedly execute the ISR, thus preventing the processor from executing any other application code). You are responsible for writing the body of the ISR. The code in Figure 21 (Timer 1 ISR) shows how to both declare a function to be an ISR and the general format of an ISR function. For this example, the parameter "ipl4" sets the Timer 1 interrupt level to 4. This method of declaring an ISR eliminates the requirement of a function prototype.

Figure 21: Harmony Generated Timer ISRs for Static Drivers.

Change Notice Interrupts

Change notice (CN) interrupts are generated on selected enabled digital I/O pins whenever the voltage of the pin changes, causing the processor to read a logic value that is different from the previous reading of the PORT register. Pins designated as CN interrupt pins are shown in Table 6. *Only BTN1 and BTN2 on the chipKITTM Pro MX7 processor board have the capability to generate CN interrupts. The I/O pin associated with BTN3 cannot generate a CN interrupt.*

In this code, CN interrupts are turned on for CN8 and CN9, which correspond to BTN1 (RG6) and BTN2 (RG7). You can use MHC to enable CN for these pins and the internal pull-up resistors should be disabled, because these pins have external pull-up resistors on the chipKITTM Pro MX7 processor board.

CNx	**Port**	**CNx**	**Port**
CN0	RC13	CN10	RG8
CN1	RC14	CN11	RG9
CN2	RB0	CN12	RB15
CN3	RB1	CN13	RD4
CN4	RB2	CN14	RD5
CN5	RB3	CN15	RD6
CN6	RB4	CN16	RD7
CN7	RB5	CN17	RF4
CN8	**RG6 – BTN1**	CN18	RF5
CN9	**RG7 – BTN2**	CN19-CN21	NA

Table 6: I/O pins with CN interrupt capability.

Unfortunately, the MHC does not automatically generate the initialization and ISR code for CN when they are enabled, so you have to write the code yourself.

- The ISR code is similar to the Timer ISR (Figure 21) but notice that a single interrupt vector is used for all CN interrupts (_CHANGE_NOTICE_VECTOR). The interrupt does not tell you if the pin went high or low; only that the condition on one of the selected pins has changed. The ISR code must return a type void and have no parameters passed to it. The CN interrupt flag must be cleared prior to exiting the ISR which is INT_SOURCE_CHANGE_NOTICE.
- It is necessary to initialize the CN interrupt in SYS_Initialize function of the system_init.c file. The required code must enable the change notice interrupt source and set its priority and subpriority levels. The following code enables CN and sets interrupt priority to level 1 and its subpriority to level 0:

```
PLIB_INT_SourceEnable(INT_ID_0, _CHANGE_NOTICE_VECTOR);
PLIB_INT_VectorPrioritySet(INT_ID_0, INT_VECTOR_CN, INT_PRIORITY_LEVEL1);
PLIB_INT_VectorSubPrioritySet(INT_ID_0, INT_VECTOR_CN,
INT_SUBPRIORITY_LEVEL0);
SYS_INT_SourceEnable(INT_SOURCE_CHANGE_NOTICE);
```

Experiment

You are asked to implement a system that runs entirely using foreground processes. The Timer is used to set the interrupt flag once each one second. The PIC32 change notice interrupt generates an interrupt when a button is pressed or released. As specified in Table 7, we will use two buttons (BTN1 & BTN2). If BTN1 is selected then the LED will flash as a counter from starting always from 0 to 15. But if BTN2 is selected the LED will flash on and off. (Note: you need to use a delay between each toggle).

When implementing interrupts for Timer and CN:

- Set change notice interrupts to detect activity on BTN1 andBTN2 only, the group priority level 1 and the subgroup level 0.
- Initialize Timer to generate an interrupt once each ms . Set the group priority for level 2 and the subgroup level for 0. And create a delay function using the timer

Button	**State**
BTN1	Counter
BTN2	Flashing

Table 7: Button-controlled.