CALIFORNIA STATE POLYTECHNIC UNIVERSITY, POMONA
COLLEGE OF ENGINEERING

ECE 3301L Fall 2025 **Microcontroller Lab** **Felix Pinai**
Session 2

Experiment 8
External Interrupts

The purpose of this lab is to get the student to be familiar with the concept of system interrupt, and particularly the external interrupts.

PART A) External Interrupts

A hardware interrupt is a signal generated by a special function device to indicate to the CPU that an event has happened and the CPU needs to temporary leave what it is working on so it can go to process the needs of that event. There are some procedures to ensure that the CPU remembers the location where it leaves before it goes away to process the interrupt. The new location where the CPU jumps to take care of the interrupt is called an Interrupt Service Routine (ISR). The CPU will enter the ISR routine when the interrupt occurs and when the task in that routine is completed, the CPU will perform a Return from Interrupt (RETI) to return to the main program at the location where the CPU was interrupted.

The advantage for an interrupt is that the CPU does not have to constantly wait for a hardware event to happen. It just needs to perform any task that it needs to take care of or sit idle doing nothing and then when an interrupt happens, the CPU just jumps to the ISR to handle that interrupt.

To allow interrupts to be able to be processed by the CPU, each hardware device that is capable to generate an interrupt has an 'Interrupt Enable' (IE) bit associated with it. When to set 1, the IE will allow an interrupt to happen. In addition, an 'Interrupt Flag' (IF) bit is set when an interrupt event happens. If the IE is on and when IF is also set, then the interrupt has occurred. When an interrupt is processed by the CPU, the IF must be cleared up to allow the next interrupt event to happen. If IF is not cleared, no further interrupt can be generated.

In addition, a 'Global Interrupt Enable' (GIE) bit is a general control bit that will allow any interrupt to be generated to the CPU when that bit is set to 1. This GIE has to be turned on for interrupts to happen.

Refer to chapter 10 'Interrupts' of the PIC18F460 datasheets to see all the control bits for the interrupt functions of the hardware devices.

External interrupt is a type of interrupt when an input changes from a logic level to the other level. The edge transitioning from a selectable low-to-high or high-to-low transition

will in fact generate an interrupt. There are three pins on the PIC18F4620 that can be used for such operation. They are called by the name INTx where x can be 0, 1 or 2. Here are the registers associated with these pins:

1) Interrupt Flag bits:

```
INTCONbits.INT0IF ;          // INT0 IF is in INTCON
INTCON3bits.INT1IF;          // INT1 IF is in INTCON3
INTCON3bits.INT2IF;          // INT2 IF is in INTCON3
```

2) Interrupt Edge Select bits:

```
INTCON2bits.INTEDG0 ;        // INT0 EDGE is in INTCON2
INTCON2bits.INTEDG1;         // INT1 EDGE is in INTCON2
INTCON2bits.INTEDG2;         // INT2 EDGE is in INTCON2
```

3) Interrupt Enable bits:

```
INTCONbits.INT0IE ;          // INT0 IE is in INTCON
INTCON3bits.INT1IE;          // INT1 IE is in INTCON3
INTCON3bits.INT2IE;          // INT2 IE is in INTCON3
```

4) The Global Interrupt:

```
INTCONbits.GIE=1;            // Set the Global Interrupt Enable
```

The INTxIF bit is set to 1 when the associated INTx signal has made the edge transition that would cause a subsequent interrupt. This bit needs to be cleared once the interrupt has been processed to allow the next interrupt to occur again. Leaving that bit to logic 1 will prevent further interrupt.

An INTEDGx bit, when set to 0, indicates that an interrupt is to be generated when the logic transition goes from high-to-low (falling edge) and the other way when it is set to 1.

The INTxIE must be set to logic 1 to enable the associated INTx signal to generate an interrupt when it transitions with the type of edge specified by the INTEDGx bit (see below).

Once everything is done, we will need to set the Global Enable to 1 to allow the interrupt operation to work.

To test how the interrupts work, we will need to do the following:

a) Copy the lab #7's project to make a new project for lab #8.

b) Make sure to use a separate folder for the new lab and copy all the .h and .c files from the old folder to the new one.

c) Use the main program for lab #7 and rename it Lab8.c.

d) Make sure that the files indicated on both the 'Header files' and 'Source files' use the files that are in the new folder. If you are not sure, remove all the files on those two tabs and add those files again by selecting the ones from the new folder.

e) Now, add the two attached files, 'Interrupt.h' and 'Interrupt.c' into the respective tabs.

f) Open the 'Interrupt.c' and go to the function 'Init_Interrupt()'.

g) Add the following codes:
    1. Look at the Interrupt Flag section above showing the 'IF" flags. Write the code to set those flags to logic 0. We need to clear those flags before we can start any interrupt operations.
    2. Do the same for the 'INTEDGx' flags by setting them with the proper logic. A '1' for low-to-high interrupt versus a '0' for a high-to-low interrupt. Since we are using a push-button, when the switch is pressed down, the logic of the signal to that switch will transition from high to low. Choose the proper value ('0' or '1') to program this 'INTEDGx' flag.
    3. Next is to program all the 'IE' flags to 1 to enable each individual INTx interrupt.
    4. Finally, set the 'GIE' flag to 1 to enable the interrupt operation.
    5. That will conclude that routine.

h) Go to the 'void interrupt high_priority chkisr()'
    1. A line was provided to check if INT0IF flag is set to 1. If so, call the function INT0_ISR() to service that interrupt.
    2. Add additional lines to take care of the other interrupts INT1 and INT2 and by calling the appropriate service routines for those interrupts.

i) The function 'void INT0_ISR()' shows an example of how to service the INT0 interrupt. The first line is to clear the INT0IF flag. Next, a variable INT0_Flag is set to 1. That variable is monitored in the main program later. Add the supporting codes for the other two 'INT1_ISR()' and 'INT2_ISR()' by implementing the same handling done in 'INT0_ISR()'. Be aware that the other IF flags reside in different registers.

j) Open the main program Lab8.c.

k) Add the following lines to define the new variables:

```
char INT0_Flag;
char INT1_Flag;
char INT2_Flag;
```

l) Add the following codes below the line 'Initialize_LCD_Screen(); '

```
RBPU = 0;
Initialize_Interrupt();

while (1)
{                                       // Do nothing,

        if (INT0_Flag == 1)
        {
                INT0_Flag = 0;          // clear the flag
                printf ("INT0 interrupt pin detected \r\n");
                                        // print a message that INT0 has
                                        // occurred
        }
        if (INT1_Flag == 1)
        {
                INT1_Flag = 0;          // clear the flag
                printf ("INT1 interrupt pin detected \r\n");
                                        // print a message that INT1 has
                                        // occurred
        }
        if (INT2_Flag == 1)
        {
                INT2_Flag = 0;          // clear the flag
                printf ("INT2 interrupt pin detected \r\n");
                                        // print a message that INT2 has
                                        // occurred
        }
}
```

Each time when either push-button SW2, SW3 or SW4 (see schematics) is pressed down, the interrupt INT0, INT1 or INT2 will be generated. The associated INT0_ISR(), INT1_ISR() or INT2_ISR() will be executed to service the interrupt. In each routine, the associated hardware INTxIF will be cleared to allow the next interrupt to be generated. Next, the associated variable INT0_flag, INT1_flag or INT2_flag will be to 1.

In the main program, the three variables INT0_Flag, INT1_Flag, INT2_Flag will be constantly monitored. When either one flag is detected to be 1, that variable will be cleared and a message will be printed to show the corresponding push-button switch has been pressed.

Note that the instruction "RBPU = 0" must be asserted in the initialization to force all the three INTx lines to be internally pulled up with no need for external pull-up.

Complete the above program and run it. Push each button associated with the INT line and see on the screen that a message showing that the INT pin has been pressed.

In addition, add the following connections to the logic analyzer:

- Channel 0: INT0 @RB0
- Channel 1: INT1 @RB1
- Channel 2: INT2 @RB2
- Channel 7: TXD @RC6

Activate the Async Serial and assign it to Logic Analyzer 7. Make sure to select the baud rate to be 19200.

Press on any of the three push-button switches. Notice the associated INTx line with that button will go from high to low and the serial port on channel 7 will send out a series of message. This is to illustrate the action of the interrupt line.

PART B) Use of external interrupts in the Traffic Controller Design

We will now use the external interrupts to trigger the requests for pedestrian accesses on the North-South and East-West directions. We will not use the DIP switches for the Pedestrian requests as in Lab 8. We will use instead the two push-buttons SW2 and SW3 that are hooked to the signals INT0 and INT1 (PORT B bit 0 and PORTB Bit 1 respectively).

When a push-button is pressed, this will set a flag to indicate a request for pedestrian access. When it is time to do the handling of pedestrian access, the flag will be checked. If it is not set to 1, then no pedestrian access will be performed. If it is set 1, then the pedestrian process will be performed. When done, the flag will be cleared so that on the next pass no new access will be performed again unless the push-button switch is pressed again in between.

Use the program developed in the section above:

1) Comment out the lined in blue to remove the new while loop.

2) Remove the definitions in the 'main.h' file for the signals associated with the DIP switches in the original Lab #8:

    a. NS_PED_SW
    b. EW_PED_SW

We no longer use these switches to detect pedestrian requests. Create two variables with the same name in your main program:

c. char NS_PED_SW = 0;
d. char EW_PED_SW = 0;

3) Since the same variables above are checked in the 'Main_Screen.c', we will need to add the external references to those variables. Open that file and add the following lines:

```
extern char EW_PED_SW ;
extern char NS_PED_SW ;
```

right below the line 'extern char MODE'.

4) In the 'Interrupt.c' file, replace in the routines INT0_ISR() and INT1_ISR() the variables INT0_Flag and INT0_Flag respectively by those two variables NS_PED_SW and EW_PED_SW.

5) This substitution will allow each pedestrian request to activated when the INT0 or INT1 push-button switch is pressed down.

6) When the traffic program starts, both the NSP and EWP fields on the TFT will show a '0'. When either push-button switch connected to INT0 or INT1 is pressed, the respective NSP or EWP variable will be set to '1' inside the Interrupt Service Routine. During the running of the 'Day-Mode', the pedestrian process 'PED_CONTROL' will handle the pedestrian crossing as usual. However, make sure to add one extra step in this to clear the respective NS_PED_SW or EW_PED_SW variable that was set to 1 back to 0 so that the pedestrian countdown does not get executed again until the associated push-button is pressed in-between.

7) In addition, in the interrupt service routine ISR(), add some codes to check what mode the INTx has occurred. If the mode is in Day mode, then proceed as normal to set the variable NS_PED_SW or EW_PED_SW. However, if the mode is in Night more, do not set the variable to 1 because no pedestrian access is allowed in that mode.

8) When switching from day mode to night mode, make sure to clear both NS_PED_SW and EW_PED_SW variables to 0 to prevent any pedestrian process to be handled in night mode.

If the implementation is done properly, the 'NSP' and 'EWP' display on the screen should have a '0' shown most of the time. When a push-button either at INT0 or INT1 is pressed, then a '1' will show up accordingly and stays at '1' until the pedestrian process

in the proper direction is completed. The '1' should be changed to '0' and stays at '0' until the next time the push-buttton is pressed again.

PART C) Implementation of Flashing Mode

Use the INT2 associated with the push-button SW4 to initiate an RED-light flashing request. When the INT2 occurs, it will set a flag variable called Flashing_Request to logic '1'. When the traffic controller has finished executing a sequence either in Day mode or in Night mode (depending on the Photo-resistor), the program does return back to the main program. The next task is to check the logic of this Flashing_Request flag. If 0, then no request is asked. If 1, then the program will reset the Flashing_Request flag to 0 and then call a routine called Do_Flashing() whereas the following steps must be executed:

1) In the main program, add the following definitions:

   ```
   char Flashing_Request = 0;;
   char Flashing_Status = 0;
   ```

2) In the 'Interrupt.c' replace the INT2_Flag by the variable 'Flashing_Request'. This will make the push-button connected to INT2 the activation for the Flashing operation.

3) In the main program where the codes:

```
if (Light_Sensor == 1)
    {
      Day_Mode();                  // calls Day_Mode() function
    }
    else
    {
      Night_Mode();                // calls Night_Mode() function
    }
```

Add the following codes:

```
if (Flashing_Request == 1)
{
   Flashing_Request = 0;
   Flashing_Status = 1;
   Do_Flashing();
}
```

4) Here are the requirements for the implementation of the routine 'Do_Flashing()':

    a. Create a while loop waiting for 'Flashing_Status' to be 0.
    b. While 'Flashing_Status' is still 1, check if Flashing_Request is 0.

        i. If 0, then then we are still in flashing mode by:

            1. Set all the four directions with the RED color,
            2. Then Wait_One_Second(),
            3. Set all the four directions with the OFF color,
            4. Do another Wait_One_Second(), (). This will create the effect of flashing the RED color in all directions.

        ii. Else (If 1), then there is a request to exit the flashing mode. Clear both 'Flashing_Request' and 'Flashing_Status' to force to exit the while loop.

In addition, to update the status of the 'R' and 'S' fields on the LCD, add the following lines in the 'Main_Screen.c'

a) At the beginning just below the line 'extern char MODE':

```
extern char Flashing_Request;
extern char Flashing_Status;
```

b) In the update_LCD_misc() routine inside the 'Main_Screen.c' file:

```
if (Flashing_Request == 0) FlashingR_Txt[0] = '0'; else FlashingR_Txt[0] = '1';
if (Flashing_Status == 0) FlashingS_Txt[0] = '0'; else FlashingS_Txt[0] = '1';
```

just before the line starting with 'drawline'.