ECEC 304: Design with Microcontrollers
Lab 4: Interrupt Processing

July 1, 2016

This lab assignment comprises four problems dealing with interrupt processing and timers. Once you have successfully completed each part, please show the output of the program to the teaching assistant to be checked for correctness. Complete the lab by the end of day on Thursday, July 21st. Your C code must be clearly written, properly formatted, and well commented for full credit.

You may work on this assignment in a team of up to two people. Also, note that the submitted solutions must be the original work of your team. Solutions copied from other teams or from online sources will result in a grade of zero for the entire lab assignment.

Important Submission Instructions: In addition to getting your work checked off by the TAs, upload your solution to Lab 4c. Specifically, a file called `wallclock.c` containing the code for the main processing loop (the `main()` function) as well as the code for the UART and timer ISRs (the `CY_ISR(uartISR)` and `CY_ISR(timerISR)` functions. You can find the submission link on BBLearn under the `Lab submissions/Lab 4` link. One submission by a team is sufficient.

Lab 4a: Interrupt-Driven General Purpose IO (GPIO)

(10 points) Let us start with an example of how to handle an interrupt generated by a device connected to a GPIO pin—in our case, a switch which is onboard the PSoC board. We will develop a simple interrupt service routine (ISR) to service the interrupt as follows: when the switch is pressed, toggle the red onboard LED. Note the following important points:

- The LED is connected to P1[6] with a copper trace on the board, that is to the 6^{th} pin belonging to I/O port 1. The switch is similarly connected to P0[7]. External connections are unnecessary.
- The LED is active low; that is, driving the pin low will turn it on.
- The switch is grounded when pressed.

As a first step, your must configure an input pin—let's call it `SW`—under the "My Design" tab in the .cysch file, as part of your top-level design within PSoC Creator. Note the following points:

- The input-pin parameters must be chosen as follows: digital input, a resistive pull up drive mode, and the initial drive state set to high. Also, set the pin input to generate an interrupt—called `SW_Int`—on the falling edge of the pulse and to trigger an ISR.
- Map the pin component to P0[7] in the .cydwr file under the "Pins" tab.
- Map the LED output pin component to P1[6] in the .cydwr file.

Once you build the code, the system will generate low-level code including the following stub for the ISR called `SW_Int_Interrupt` within the file `SW_Int.c`. (The file may be called differently in your case depending on how you name your interrupt.)

```
CY_ISR(SW_Int_Interrupt)
{
   /*  Place your Interrupt code here. */
   /* `#START SW_Int_Interrupt` */

   /* `#END` */
}
```

We choose not to place our code within the above stub but rather to write the ISR routine within the `main.c` file. To do so, we must change the address of the ISR within the interrupt vector table. Recall that this table specifies the addresses of all the ISRs used by the system; the ISR address vectors become nothing more than a "jump table" which jumps to the real ISR code. The `SW_Int_SetVector` function allows us to do this as follows:

```
int main()
{
    CyGlobalIntDisable;    /* Disable global interrupts. */
    SW_Int_Start();        /* Start the ISR. */
    /* Add custom ISR to interrupt vector table. */
    SW_Int_SetVector(switchISR);
```

```
    CyGlobalIntEnable;     /* Enable global interrupts */

    for(;;){               /* Idle the processor. */
    }
}
```

The ISR called `switchISR` that toggles the LED whenever the switch is pressed is as follows:

```
CY_ISR(switchISR)
{
    /* Simple debouncing to eliminate any switch glitches. */
    CyDelay(50);

    while(!SW_Read()); /* Wait for switch to be released. */
    CyDelay(50);

    RED_LED_Write(!(RED_LED_Read())); /* Toggle the LED. */
    SW_ClearInterrupt();              /* Clear the interrupt. */
}
```

Using the above code snippets as a starting point, write an interrupt-driven program to perform the following task. Begin with the LED blinking once every second. Then, every time the switch is pressed, decrease the blinking frequency by a second until the LED reaches a blinking rate of once every five seconds, after which we increase the blinking rate back up by a second upon each switch press until the rate reaches once every second. The process repeats itself.

Hint: Handle the blinking functionality within the loop in the main program in which the blinking rate is a function of the number of times the switch has been pressed. Your ISR simply updates a global variable shared with the main function that indicates the number of switch presses.

Signature of the teaching assistant: _____ Date: _____

Lab 4b: Handling UART Interrupts

(10 points) In lab assignment 1, we saw how to accept keyboard input from the terminal by continuously polling the UART component, say named `UART_1`, for input data within the main program. The polling-based method can be written as follows.

```
void main(void)
{
   uint32  rxData;
   for(;;){
      rxData = UART_1_UartGetChar();
      if(rxData){
         UART_1_UartPutChar(rxData); /* Echo character. */

         /* Processing code goes here. */
      } /* End if */
   } /* End for */
}
```

This assignment asks you to operate the UART in an interrupt-driven fashion, accept input from the keyboard, and dynamically control the blinking rate of the LED.

Configure the UART component in your top-level design as follows:

- Set the baud rate on the UART to 9,600 bps. Match this baud rate on the terminal running on the host PC.
- Under the "UART Advanced" tab, have the UART generate an external interrupt when the receive FIFO buffer is not empty and associate an interrupt component with this event. Interrupt components are listed in the component catalog under "System." Say, we name this interrupt as `Uart_Int`.
- Designate pin P0[4] as the receive (`rx`) pin and P0[5] as the transmit (`tx`) pin in the .cydwr file under the "Pins" tab.
- External jumper wires are required: connect P0[4] to P12[7] and P0[5] to P12[6] on the board. This effectively transfers the bit stream between the UART component and the USB connector.

When you build your project, the system generates low-level code including an ISR stub for you within the `Uart_Int.c` file. As with the previous assignment, the interrupt vector table entry can be changed to point to the ISR code that we will write within the `main.c` file.

```
CY_ISR(uartISR) /* ISR associated with the UART. */
{
   uint32 rxData;
   rxData = UART_1_UartGetChar(); /* Get character from buffer. */
   UART_1_UartPutChar(rxData); /* Echo back to terminal. */
```

```
   /* Process the user input as appropriate. */

   /* Clear the interrrupt. */
   UART_1_ClearRxInterruptSource(UART_1_GetRxInterruptSource());
}

CY_ISR(switchISR) /* ISR associated with the GPIO. */
{
   CyDelay(50); /* Debouncing the switch. */
   while(!SW_Read()); /* Wait for switch to be released. */
   CyDelay(50);

   /* Process switch state as appropriate. */

   SW_ClearInterrupt(); /* Clear the interrupt. */
}

int main()
{
   CyGlobalIntDisable;  /* Disable global interrupts. */
   Uart_Int_Start();    /* Start the ISR associated with UART. */
   Uart_Int_SetVector(uartISR);  /* Vector ISR address. */

   SW_Int_Start();  /* Start the ISR. */
   SW_Int_SetVector(switchISR);  /* Vector ISR address. */
   CyGlobalIntEnable;   /* Enable global interrupts */

   UART_1_Start();  /* Start the UART. */

   for(;;){
      /* Procesing code. */
   }
}
```

Using the above code as a starting point, write an interrupt-driven program to perform the following task. Begin with the red LED blinking once every second. At run time, accept user input from the keyboard to dynamically change the blinking period anywhere between 1 to 5 seconds. Your code must check if the user input is within the desired range. If not, the LED must maintain the previous blinking rate. Moreover, the dynamic adjustment of this blinking period must be affected only if the onboard switch is ON. The user should be able to toggle the switch between the ON and OFF states at any time. If the switch is OFF, the keyboard input is ignored and the LED continues blinking at the previously set rate. If the switch is ON, the keyboard input is accepted and the LED starts blinking at the newly specified rate.

Signature of the teaching assistant: _____ Date: _____

Lab 4c: Implementation of a Wall Clock

(20 points) In this assignment you will use the timer and UART components in interrupt-driven fashion to implement a running wall clock and display it on the terminal in `hh:mm:ss` format, where `hh` stands for hours, `mm` for minutes, and `ss` for seconds. The wall-clock program must also have the following additional functionality:

- It must allow the user to set the `hh`, `mm`, and `ss` fields prior to starting the clock.
- It must allow the user to reset the various fields during operation. When the user presses the character 'c' on the keyboard, the clock must be stopped, new values for `hh`, `mm`, and `ss`, accepted as inputs from the user, and the clock restarted with these new values.

The timer is available under the Digital/Functions menu in the component catalog. Use the Timer component, not the Timer Counter component. You can configure the timer, call it `TIMER_1`, in your high-level design as follows:

- The internal counter within the timer counts down; starting with an initial value n—which can be chosen to be an 8-bit or 16-bit value, as appropriate—and a clock signal of frequency f, the counter counts down at each rising edge of the incoming clock-pulse train until the counter value reaches 0; upon which the value is reset to n, and the process repeats itself. So, the period, p, of the timer can be calculated as $p = n \times 1/f$ seconds.
- At the end of each period, the timer asserts the line associated with *terminal count*, shown as the 'tc' line in the timer block. Associate an interrupt with this line such that the corresponding ISR, called `timerISR` in the code snippet below, is triggered at periodic time intervals.

The following code snippet may be useful as a starting point for your wall-clock program.

```
CY_ISR(uartISR) /* ISR to handle UART interrupts. */
{
    /* Code to read and handle UART input. */

    /* Clear the interrupt. */
    UART_1_ClearRxInterruptSource(UART_1_GetRxInterruptSource());
}

CY_ISR(timerISR) /* ISR to handle the tc interrupt. */
{
    /* Code to handle tc interrupt. */

    /* Reset the interrupt. */
    TIMER_1_ReadStatusRegister();
}

int main()
{
```

```
    CyGlobalIntDisable;
    Uart_Int_Start();
    Uart_Int_SetVector(uartISR);

    myTimer_Int_Start();
    myTimer_Int_SetVector(timerISR);
    CyGlobalIntEnable;

    UART_1_Start(); /* Start the UART. */

    writeString("Established connection to terminal. \n \r");

    /* Code to initialize wall clock here. */

    TIMER_1_Start(); /* Start the timer. */

    for(;;)
    {
        /* Code to display new clock value on each
           terminal count interrupt from the timer. */

        /* Code to update clock values when user presses 'c' */
    }
}
```

Signature of the teaching assistant: _____ Date: _____

Lab 4d: Measuring Elapsed Time

(20 points) This assignment asks you to develop functions that use the timer component to measure elapsed time between events—in this case, to measure the execution time of a given function. Consider the following function that multiplies two $N \times N$ matrices:

```
void  matrixMult(float A[MATRIX_SIZE][MATRIX_SIZE], \
                 float B[MATRIX_SIZE][MATRIX_SIZE], \
                 float C[MATRIX_SIZE][MATRIX_SIZE])
{
   int8 i, j, k;
   float temp;
   for(i = 0; i < MATRIX_SIZE; i++){
      for(j = 0; j < MATRIX_SIZE; j++){
         temp = 0.0;
         for(k = 0; k < MATRIX_SIZE; k++){
            temp += A[i][k] * B[k][j];
         }
         C[i][j] = temp;
      }
   }
}
```

You are asked to write two functions, `tic` and `toc`, that work in concert to measure the execution time incurred by `matrixMult` in milliseconds. The skeleton of the main program is as follows where the UART and timer components are named `UART_1` and `TIMER_1`, respectively.

```
#define MATRIX_SIZE 5

int tc;
float elapsedTime;

CY_ISR(uartISR) /* ISR to handle UART interrupts. */
{
   /* Your ISR code. */

   /* Clear the interrupt. */
   UART_1_ClearRxInterruptSource(UART_1_GetRxInterruptSource());
}

CY_ISR(myTimerISR) /* ISR to handle tc interrupts from timer. */
{
   tc = tc + 1;

   /* Your ISR code. */
```

```
   /* Reset the interrupt. */
   TIMER_1_ReadStatusRegister();
}

void tic(void)
{
   /* Your code. */
}

void toc(void)
{
   /* Your code. */
}

int  main()
{
   CyGlobalIntDisable; /* Disable global interrupts. */
   Uart_Int_Start(); /* Start the ISR associated with UART. */
   Uart_Int_SetVector(uartISR); /* Update vector table. */

   myTimer_Int_Start(); /* Start the ISR associated with timer. */
   myTimer_Int_SetVector(myTimerISR); /* Update vector table. */
   CyGlobalIntEnable;

   UART_1_Start(); /* Start the UART. */
   TIMER_1_Start(); /* Start timer. */

   srand(rand()); /* Initialize random number generator. */

   float A[MATRIX_SIZE][MATRIX_SIZE];
   float B[MATRIX_SIZE][MATRIX_SIZE];
   float C[MATRIX_SIZE][MATRIX_SIZE];

   for(;;){
      populateMatrix(A); /* Populate matrices with random numbers. */
      populateMatrix(B);

      tic(); /* Time the multiplication operation. */

      matrixMult(A, B, C);

      toc();

      /* Display elapsed time on the terminal. */
```

```
        CyDelay(1000);
    }
}
```

In the above, a random number generator is initialized via the `srand` function and the matrices are populated with random floating-point values during each multiplication run as follows.

```
void populateMatrix(float M[MATRIX_SIZE][MATRIX_SIZE])
{
    int8 i, j;
    for(i = 0; i < MATRIX_SIZE; i++){
        for(j = 0; j < MATRIX_SIZE; j++){
            M[i][j] = (float)rand()/(float)RAND_MAX;
        }
    }
}
```

You are free to implement the `tic` and `toc` functions as you see fit. These functions must measure the elapsed time after each run of `matrixMult` to within one millisecond accuracy, and the result must be stored in the variable `elapsedTime` after the function `toc` returns and subsequently displayed on the terminal. Display the elapsed times when multiplying matrices of the following sizes: 5×5, 10×10, and 15×15.

Hint: Use the `ReadCounter` function, generated by the system as part of the timer API, to obtain the current value of the internal counter. Since we named our timer component as TIMER_1 in the above discussion, this function would be called `TIMER_1_ReadCounter`. See the function declaration within the TIMER_1.c file for more information. Use this function as well as the terminal count information captured by the timer ISR to implement your timing functions.

Signature of the teaching assistant: _____ Date: _____