ECE 315 Lab 2 Report

Section H31

Due: February 27, 2019

By Nathan Klapstein (1449872), Thomas Lorincz (1461567)

Table of Contents

Abstract

In lab 2, 4 exercises were performed to gain experience with interfacing the Freescale MCF54415 microprocessor with a user input peripheral. The exercises introduce the complexities of encoding, multiplexing, and handling interrupts in order to properly issue commands with a 16-button keypad. As the exercises are completed, experience will also be gained in µC/OS programming, static code analysis, and oscilloscope functionality.

In Exercise 1, an oscilloscope is used to perform a single short capture to record timing characteristics of the keypad encoder. Once captured, the timing information is compared to the datasheet values. Completion of Exercise 1 will ensure that the keypad encoder IC is functional (hasn't been damaged by previous use) and will provide experience in using an oscilloscope.

Exercise 2 involves building a new Netburner Eclipse project to use in exercises 3 and 4. Once a new project is built, the boilerplate code will to be run to ensure that the board is connected and working properly.

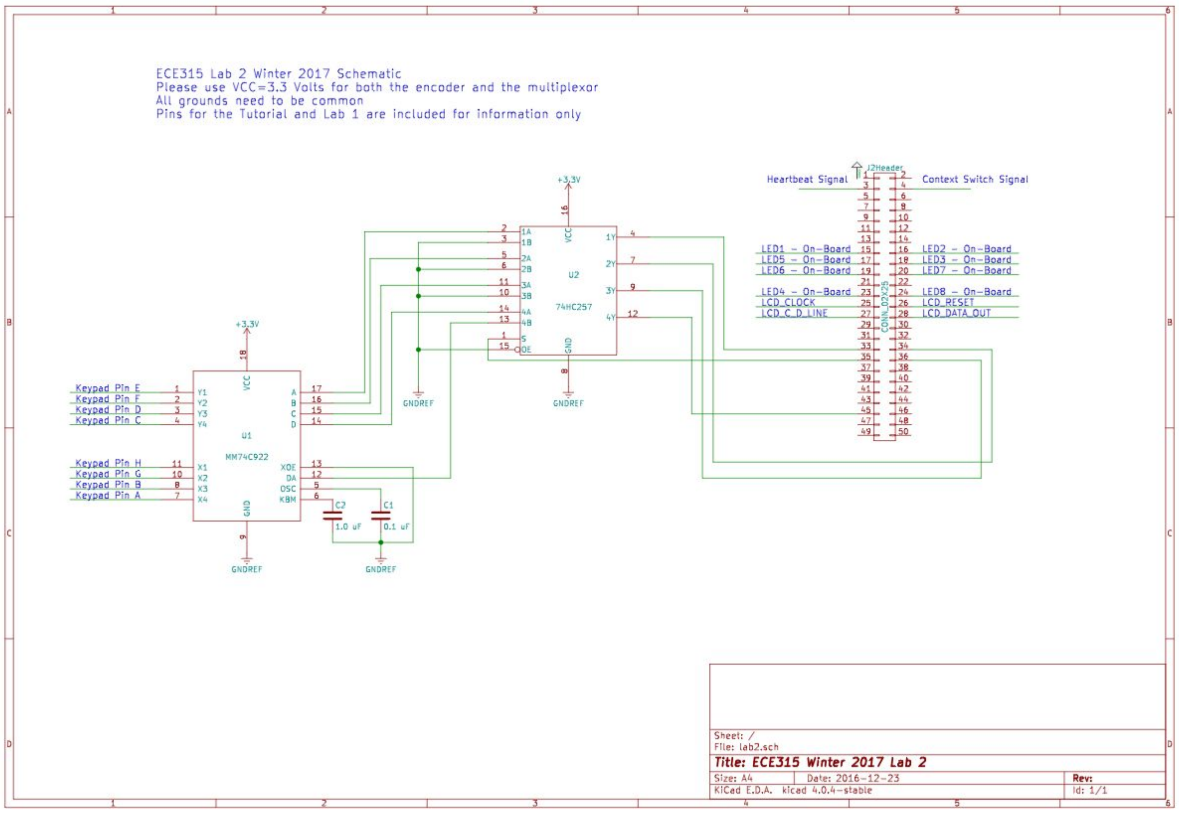
Exercise 3 is focused on setting up an interrupt request (IRQ) and interrupt service routine (ISR) to print keypad inputs to the serial console (MTTTY). Once the interrupt mechanisms are implemented, a mapping should be created such that a unique value can represent each key. To pass these values to the UserMain runtime loop, it is recommended to use a queue so that key presses are not lost and can be processed in order.

Finally, in Exercise 4, functionality is written such that the corner keys of the keypad can move a sprite on the LCD screen. Completion of Exercise 4 will build on some knowledge from lab 1. It will also demonstrate proper usage of IRQs, ISRs, and queues in order to process user input from a peripheral circuit.

Design

Exercise 1

Exercise 1 required the proper construction of a circuit that interfaced the LCD hardware (a Nokia 5110 graphical display which is driven by the PCD 8544 LCD controller) with the MCF54415 microcontroller (similar to lab 1), however, unlike lab 1, additional hardware is added for supporting human input (via a 16-button keypad). This included a MM74C922 keypad encoder integrated circuit (IC) and a SN74HC257 Multiplexor IC to allow key-presses from a 16-button keypad to be properly encoded to later interface and provide input into the MCF54415 microcontroller. As such, a circuit was designed that followed the reference schematic shown below:



Extracted from the Lab2 reference material at https://eclass.srv.ualberta.ca/pluginfile.php/4759492/mod_resource/content/6/ECE315_Winter2017_Lab2_Schematic.pdf on 2019-02-26

Next, the oscilloscope was used to check the MM74C922 keypad encoder IC, comparing the signal characteristics of the IC's Data Available line on various input types. These characteristics were then compared to the IC's datasheet to ensure that the IC's operation was nominal.

Exercise 2

Exercise 2 finished setting up the designed circuit within Exercise 1. And began initial laboratory project initialization. Using the template project code provided `lab2.zip` as a basis to create an Eclipse workspace for later lab 2 exercises.

After setting up the Eclipse workspace for lab 2's project code, the project code was compiled, uploaded, and executed on the MCF54415 microcontroller.

Next a oscilloscope was used to probe both the "Heartbeat" pin (pin `J2[3]`) and "Context Switch" pin (pin `J2[4]`) to ensure that the microcontroller's operation was nominal.

Additionally, visual inspection of the LCD screen was done to ensure it was operating nominally.

Exercise 3

Exercise 3 consisted of modifying the code within `lab2.cpp` to operate the keypad in a interrupting mode rather than a polling mode (this potentially can result into software that uses less CPU cycles busy-waiting for input). As a result modifications were made within the `Keypad::Init` and `Keypad::read_data` methods of `Keypad.cpp` to properly set up the pins of the microcontroller to enable accepting a signal to initiate a interrupt.

These code modifications are as follows:

Within `Keypad::Init`:

```
// other code omitted…
} else if (mode == KEYPAD_INT_MODE) { // KEYPAD_INT_MODE
       // interrupt init here for exercise 3
       // initialization for the keypad to be in interrupt mode
       KEYPAD_DO_D.function(PINJ2_45_IRQ1);
       SetPinIrq(PINJ2_45_IRQ1, 1, isr_func);
       EnableIrq(PINJ2_45_IRQ1);
}
// other code omitted…
```

This set and enabled the pin `PINJ2_45_IRQ1` to initiate interrupts requests (IRQ) on a rising edge signal and allow the function pointed to by `isr_func` to handle such interrupt requests.

Within `Keypad::read_data`:

```
// other code omitted...
if(byte_mode == KEYPAD_INT_MODE){
       // exercise 3 code
       DisableIrq(PINJ2_45_IRQ1);
}
// other code omitted...
if(byte_mode == KEYPAD_INT_MODE) {
       // exercise 3 code
       Init(byte_mode, isr_func);
}
// other code omitted...
```

This disabled the pin `PINJ2_45_IRQ1` from initiating interrupts. This allowed the `Keypad` class to proceeded to read the keypress data as originally defined within the given project code without interruption. After the keypress data was read and stored within `last_encoded_data` class variable, the `Keypad` class was re-initialized with its original configuration via calling the `Keypad::Init` method, thus, re-enabling `PINJ2_45_IRQ1` to initiate interrupts once again.

After setting up the pins of the microcontroller to accept signals to initialize interrupts, the method `Keypad::EdgePortISR1` (and the respective keypad driving software) was modified to properly receive and handle these interrupts (which are at this point now triggered by key-presses on the 16-key keypad). This method was further modified to properly map the keypress to strings and relay this information of a keypress into the `UserMain` function's loop within `lab2.cpp`

To allow the 16-key keypad to send keypress information into the `UserMain` function's loop within `lab2.cpp` the `Keypad` class within `Keypad.cpp` was modified by adding a class parameter `isr_func`. `isr_func` is function pointer that points to a function that handles the interrupts spawned from key-presses from the 16-key keypad. This class parameter was set within the `Keypad::Init` method by a additional method parameter and effectively substituted the usage of the `Keypad::EdgePortISR1` method.

Below is modifications in both `Keypad.h` and `Keypad.cpp`:
Within `Keypad.h`:
```
// within Keypad class other code omitted...
/* pointer to the interrupt service routine (ISR) function */
void ( *isr_func )( void );
// other code omitted…
/**
 * Initializes all the GPIO pins that interface to the keypad.
 *
 * @param mode {@code KEYPAD_POLL_MODE} or {@code
KEYPAD_INT_MODE}
 * @param *isr_func pointer to the interrupt service routine
function
 */
void Init(BYTE mode,  void ( *func )( void ));
// other code omitted…
```

Within `Keypad.cpp`:
```
/**
 * Initializes all the GPIO pins that interface to the keypad.
 *
 * @param mode {@code KEYPAD_POLL_MODE} or {@code
KEYPAD_INT_MODE}
```

```
        * @param *isr_func pointer to the interrupt service routine
function
        */
       void Keypad::Init(BYTE mode,  void ( *isr_func )( void ))
       {
           isr_func = isr_func;
           // other code omitted…
```

Note: the `Keypad::EdgePortISR1` method was removed in both `Keypad.h` and `Keypad.cpp` due to these modifications making it redundant.

Additionally a function acting as the Interrupt Service Routine (ISR) handling method called `EdgePortISR1` was defined within `lab2.cpp`. This function set a `volatile bool` flag `buttonPressed` to `true`. Setting `buttonPressed` to `volatile` allows this variable to be accessed between threads, and contextes relatively safely. This function was passed into the `Keypad::Init` method call within the `UserMain` function and was used as the substitute to the `Keypad::EdgePortISR1` method usage.

Below is the `EdgePortISR1` function and `buttonPressed` flag as defined within `lab2.cpp`:

```
       /**
        * Flag that notes that the keypad button is pressed.
        *
        * {@code EdgePortISR1} **will** set this to {@code true}
        * {@code UserMain} **can** set this to {@code false}
        */
       volatile bool buttonPressed;

       /**
        * Interrupt service request handler.
        *
        * Sets {@code buttonPressed} to {@code true}. This then
proceeds to the change
        * behavior within the main execution loop.
        */
       void EdgePortISR1(void)
       {
           // Insert your ISR code here for exercise 3
           buttonPressed = true;
       }
```

Lastly, the while loop within the `UserMain` function was modified to react to the `buttonPressed` flag being set to `true`. This was achieved by adding a simple if check.

Below is the modification of the while loop within the `UserMain` function to react to the `buttonPressed` flag being set to `true`:

```
	// other code omitted...
	while (1)
	{
		if (buttonPressed)
		{
			// code to execute if the keypad button is executed
			// set the buttonPressed flag to false as we just
	finished handling a button press
			buttonPressed = false;
		}
	}
	// other code omitted...
```

This structure proved to be both effective and straightforward in showing flow of control between the interrupt service request handler setting the `buttonPressed` flag to true and the `UserMain` loop noticing this flag's change, reacting, and finally setting the `buttonPressed` flag to `false`.

As a result the 16-key keypad was able to issue interrupts which were serviced via the `EdgePortISR1` function. The `EdgePortISR1` method then set the `buttonPressed` flag to `true` which then notified the while loop within `UserMain` of a keypress event. Afterwards, method calls such as `myKeypad.GetNewButtonNumber()` could be used within the `UserMain` while loop to obtain info on the button pressed on the keypad, and further process such keystrokes (as done within Exercise 4).

Exercise 4

Within Exercise 4, the `lab2.cpp`, `Keypad.h`, and `Keypad.cpp` project code files were further modified to enable key-presses on the 16-key keypad to move a sprite displayed on the LCD screen. Four sprite movement directions were supported (up, down, left, and right) and the movement of the sprite on the LCD screen was bounded by the outer edge of the LCD screen. Thus, the sprite would always remain visible on the LCD screen (never going off the display). Movement that would move the sprite off of the LCD screen's edge would be ignored, thus, resulting in no change to the LCD display.

Similar to lab 1 a dollar $ sprite was used within the project. The sprite's definition and a helper function for drawing the sprite are displayed below:

```
	// custom dollar $ sprite
	const BYTE dollar[7] = {0x00, 0x24, 0x2a, 0x7f, 0x2a, 0x12,
0x00};
```

```
/**
 * Draw a dollar $ sprite at the current position on the LCD.
 */
void drawDollar(void)
{
     myLCD.DrawChar(dollar);
}
```

The cursor defining the position to draw the dollar $ sprite was initially set to the origin of the LCD screen (position 0, 0) as noted below:

```
/**
 * Position of the dollar $ sprite/
 *
 * Initially set to the top left corner of the LCD screen.
 */
point cursor = {0, 0};
```

The code that initially set up drawing a dollar $ sprite on the LCD is noted below:

```
// other code omitted…
// setup the LCD screen
myLCD.Init();
myLCD.Clear();
// move the LCD cursor to the origin
myLCD.Move(cursor);
// draw the dollar $ sprite initially at origin
drawDollar();
// other code omitted…
```

Definitions noting the integers mapped valid directions to move the dollar $ sprite are noted below:

```
      /* Custom button definitions mapping keypad numbers to a
direction */
      #define UP 0
      #define RIGHT 3
      #define LEFT 12
      #define DOWN 15
```

Note: the numbers for the direction definitions were chosen as they directly map to the numbers given for the corner key-presses on the 16-key keypad. This was used to our advantage later on within Exercise 4.

To enable key-presses to change the position of the dollar $ sprite a additional function `moveDollar` was added as a response to the check of the `buttonPressed` flag being set to `true` within the while loop in the `UserMain` function. This function modified the dollar $ sprite cursor position based on the integer value that was passed in (which was given via the keypress number obtained from the `myKeypad.GetNewButtonNumber()` method call). Additionally, this function ensured that the dollar $ sprites position was bounded by the outer edge of the LCD screen. Finally, after modifying the `cursor` position value, the `moveDollar` function would update the dollar $ sprite on the LCD screen appropriately. Thus, meeting the requirements set out within Exercise 4 without utilizing a queue.

The `moveDollar` function is noted below:

```
/**
 * Move and redraw the dollar $ sprite.
 *
 * @param direction integer that represents a direction to move the dollar $ sprite.
 *          Valid integers representing directions include:
 *          0 - moves the dollar $ spite up
 *          15 - moves the dollar $ sprite down
 *          3 - moves the dollar $ sprite right
 *          12 - moves the dollar $ sprite left
 *
 * The dollar $ sprite's position is protected from being moved off of the LCD screen.
 *
 * Other integers will cause the dollar sprite $ to redraw without changing position.
 */
void moveDollar(int direction)
{
    // modify the cursor position
    switch (direction)
    {
      case UP:
            if (cursor.row > 0)
            {
                  cursor.row -= 1;
            }
            break;
      case DOWN:
            if (cursor.row < 5)
            {
                  cursor.row += 1;
            }
            break;
      case LEFT:
            if (cursor.col > 0)
            {
                  cursor.col -= 7;
            }
            break;
      case RIGHT:
            if (cursor.col < 77)
            {
                  cursor.col += 7;
            }
            break;
      default:
            break;
    }

    // update the LCD screen
    myLCD.Clear();
    myLCD.Move(cursor);
    drawDollar();
}
```

The addition of the `moveDollar` function within the while loop in the `UserMain` function is noted below:

```
    // other code omitted...
    while (1)
    {
        if (buttonPressed)
        {
            moveDollar(myKeypad.GetNewButtonNumber());
            // set the buttonPressed flag to false as we just
finished handling a button press
            buttonPressed = false;
        }
    }
    // other code omitted...
```

Testing

Test	Exercise	Description	Expected Output	Actual Output

1	2	Building, loading, and starting the default program. On success, the "application started" prompt should be displayed.	`Waiting 2sec to start 'A' to abort`
			`Configured IP = 10.0.0.102`
			`Configured Mask = 255.255.255.0`
			`MAC Address= 00:03:f4:06:ae:05`
			`Application started: Nathan Klapstein, Thomas Lorincz`
				`Waiting 2sec to start 'A' to abort`
				`Configured IP = 10.0.0.102`
				`Configured Mask = 255.255.255.0`
				`MAC Address= 00:03:f4:06:ae:05`
				`Application started: Nathan Klapstein, Thomas Lorincz`

2	3	Pressing the keypad in sequential order to ensure that each key is being read properly in the main loop (printing last key pressed every 1 second)	`a`
			`b`
			`c`
			`d`
			`e`
			`f`
			`g`
			`h`
			`i`
			`j`
			`k`
			`l`
			`m`
			`n`
			`o`
			`p`
				`a`
				`e`
				`i`
				`m`
				`b`
				`f`
				`j`
				`n`
				`c`
				`g`
				`k`
				`o`
				`d`
				`h`
				`l`
				`p`

3	3	Retrying previous test with adjusted keypad mapping (columns swapped with rows)	`a`
			`b`
			`c`
			`d`
				`a`
				`b`
				`c`
				`d`

			e f g h i j k l m n o p	e f g h i j k l m n o p
4	3	Pressing the keypad with our first attempt at configuring the IRQ and ISR for handling key presses. Once inside the ISR, we would print “Inside ISR” as a debug statement and the key that was pressed.	Inside ISR a Inside ISR b Inside ISR a ...	Inside ISR a No response
5	3	Retrying the above test after we properly reset the keypad interrupt mode. For this, we used Init(byte_mode, isr_func);	Inside ISR a Inside ISR b Inside ISR a ...	Inside ISR a Inside ISR b Inside ISR a ...
6	4	The 4 corner buttons of the keypad are set to be arrow keys that move a sprite around the LCD display. For testing purposes, the keypad button number is printed as the sprite moves (removed in submitted code).	0 (UP) 3 (RIGHT) 12 (LEFT) 15 (DOWN) 15 (DOWN) 12 (LEFT) 3 (RIGHT) 0 (UP)	0 (UP) 3 (RIGHT) 12 (LEFT) 15 (DOWN) 15 (DOWN) 12 (LEFT) 3 (RIGHT) 0 (UP)
7	4	Repeatedly pressing up (starting at {0,0}) and printing the coordinates of the sprite (to ensure it remains on screen).	0,0 0,0 0,0	0,0 0,0 0,0

8	4	Repeatedly pressing right (starting at {0,0}) and printing the coordinates of the sprite (to ensure it remains on screen).	`0,0`
			`0,7`
			`0,14`
			`0,21`
			`0,28`
			`0,35`
			`0,42`
			`0,49`
			`0,56`
			`0,63`
			`0,70`
			`0,77`
				`0,0`
				`0,7`
				`0,14`
				`0,21`
				`0,28`
				`0,35`
				`0,42`
				`0,49`
				`0,56`
				`0,63`
				`0,70`
				`0,77`

9	4	Repeatedly pressing down (starting at {0,0}) and printing the coordinates of the sprite (to ensure it remains on screen).	`0,0`
			`1,0`
			`2,0`
			`3,0`
			`4,0`
			`5,0`
			`5,0`
			`5,0`
				`0,0`
				`1,0`
				`2,0`
				`3,0`
				`4,0`
				`5,0`
				`6,0`
				`6,0`

10	4	Retrying the above test after adjusting the number of rows (mistake).	`0,0`
			`1,0`
			`2,0`
			`3,0`
			`4,0`
			`5,0`
			`5,0`
			`5,0`
				`0,0`
				`1,0`
				`2,0`
				`3,0`
				`4,0`
				`5,0`
				`5,0`
				`5,0`

11	4	Repeatedly pressing left (starting at {0,0}) and printing the coordinates of the sprite (to ensure it remains on screen).	`0,0` `0,0` `0,0`	`0,0` `0,0` `0,0`

Conclusion

The exercises in lab 2 were effective in providing the learning outcomes described in the abstract of this report. Completion of this lab provided useful experience in timing with an oscilloscope, utilizing interrupt mechanisms, and interfacing with a circuit that supports user input. As well, we were able to apply learnings from the completion of lab 1 to this lab. For example, we found static code analysis and programming to be easier in this lab because of what we learned in lab 1.

No, difficulties were encountered in the lab that were not able to be fixed by reading through the provided materials. We have no recommendations for improving lab 2.

References

Lab2 source code:
https://github.com/nklapste/ece_315_lab_2

Lab2 reference circuit diagram:
https://eclass.srv.ualberta.ca/pluginfile.php/4759492/mod_resource/content/6/ECE315_Winter2017_Lab2_Schematic.pdf on 2019-02-07