Project #3: Port Re-use and the Matrix Keypad

Jacob Harrison & Benjamin Levandowski

***Abstract*—An MSP430 microcontroller was programmed in assembly to take a number between 0 and 999 from a matrix keypad as input and blink an LED at that frequency. The microcontroller interfaced with the keypad by driving keypad columns low one at a time, and reading the keypad rows as inputs. Though frequency accuracy falls off as frequencies approach 999Hz, the implementation was successful.**

I. Description and Overview

FOR this project, we wrote a program for the Texas Instruments MSP430FR6989 microcontroller that causes an LED to blink at a specified frequency between 0 and $999Hz$. On reset, the program scrolls a simple instruction across the Launchpad LCD display and waits for users to specify a frequency. A flowchart describing the program operation is attached to this report as Figure 1 in Appendix A.

Users input a frequency using a 12-key numeric keypad. Keypad input is read using a port-scanning technique instead of a 'one hot' technique in order to conserve I/O pins on the microcontroller; this reduces the number of I/O pins required to 7 instead of 12. The procedure that processes keypad input is triggered by a timer interrupt approximately eight times a second. Each time the interrupt fires, the three columns of the matrix keypad, which are connected to microcontroller outputs, are sequentially driven lo, and the rows of the matrix keypad, which correspond to inputs on the microcontroller, are read in order to determine whether a button has been pressed. The procedure for reading keypad input is illustrated in greater detail in Figure 2, which is attached to this report in Appendix A, and described further in the *Implementation* section of this report. The hardware diagram for connections with the microcontroller is attached as Figure 4 in Appendix B.

Once a number has been read from the keypad, the number is stored as a binary coded decimal, aggregated with the two previous stored numbers, if they exist, and printed on the LCD. The user can press as many numbers as they would like; the number stored in memory and displayed on the LCD will throw out any digits that the user entered before the three most recent digits until the user presses the $*$ key to submit the trial. Once the user submits the trial, numbers are popped off the top of the stack until one of 2 conditions are met:

- Three numbers have been popped off the stack
- The number popped off the stack was -1, which indicates that the earliest number entered by the user has been popped.

Once one of these conditions are met, a clock cycle count that will yield the keyed-in frequency is calculated using power-of-two approximation and loaded into TA0CCR0, which controls the frequency of a blinking LED.

II. Methods

A. Planning

1) Interface of Assembly and C: We decided to do nearly all of the programming for this project in assembly in order to more directly control the machine instructions executed by the microcontroller and improve the efficiency of our code. However, interfacing with the LCD would have been challenging using only assembly. We decided that we would use the C functions provided by Texas Instruments for interfacing with the LCD to avoid rewriting the LCD drivers.

2) I/O Port Selection: We wanted to use a single I/O port to interface with the keypad to simplify interrupts and polling. The port we chose needed to have at least seven pins available on the GPIO breakout on the launchpad, and could not interfere with the functionality of the LCD.

Two ports had sufficient pins on the breakout section of the launchpad: port 2, and port 9. Port 9 lacked interrupt capability, but port 2 had two pins that are multiplexed into the LCD. Because we wanted to retain interrupt capability for the input ports that interfaced with the keypad, we chose to work around the optional LCD functionality of some of the pins in port 2, as is detailed in the *Errata* section of this report.

3) Timers and Clocks: Three components of our program are driven by timer interrupts:

- Blinking LED
- Keypad polling
- Scrolling Instructions on the LCD

We wanted the LED to blink at the user specified frequency with reasonable precision. If we had used ACLK sourced from the watch crystal, the precision with which the microcontroller caused LEDs to blink at frequencies close to $1kHz$ would have been unacceptably low. For this reason, we decided to source TA0, which was responsible for blinking the LEDs, from SMCLK, which operates at $1MHz$. However, using SMCLK instead of ACLK introduced a different problem for our program - Timer A can only count up to 0xFFFF; this range is not sufficient to cause the LED to blink at frequencies between $1Hz$ and $15Hz$. In order to operate the LED at low frequencies, we used conditional logic that sets and resets a clock divider on TA0 depending on the frequency input by the user.

For keypad polling and scrolling instructions, precision is less important; we want the microcontroller to poll the keypad with a frequency of about $8Hz$ and messages to scroll across the LCD at a reasonable speed, but it is not important that these frequencies be exact. For this reason, we sourced TA1 and TA2, which were responsible for keypad polling and instruction scrolling, respectively, from VLO, which is the lowest power oscillator available on the MSP430.

4) Protecting Against Invalid Input: Our program is designed to read frequencies between $1Hz$ and $999Hz$ (inclusive) from the keypad and output that frequency. In order to protect against invalid input (frequencies of 0 or frequencies greater than $999Hz$) we decided to use a conditional statement to deal with the special case of 0 frequencies, and pop only the last three digits entered off of the stack to ensure that the frequency read from the keypad is $\leq 999Hz$.

B. Implementation

1) Reading Input from the Matrix Keypad: There were two components of our implementation for reading input from the matrix:

- Drive 'columns' lo
- Process interrupt from 'row' inputs

The internal wiring of the matrix keypad is shown in Appendix B. Each button is a switch that shorts a row to a column whenever the key is pressed. Keypad rows were configured as port 2 inputs with pull-up resistors, and keypad columns were connected to outputs on the microcontroller. These outputs were normally driven hi. The Timer A1 interrupt, which fired with a frequency of approximately $8Hz$, would cause the microcontroller to briefly drive each of the outputs low. If a key was pressed, this caused the input associated associated with a given row to drop to zero, which triggered a port 2 interrupt. The I/O interrupt was aware of which column was being driven low when at the time that the interrupt was generated. Since we know the column being driven low and the input that triggered an interrupt, we know the button that was pressed on the matrix keypad. Simple decoding logic would place a numeric value corresponding to the key that had been pressed into R12. LPM3 would then be disabled, and the microcontroller would push the number in R12 to the stack and append it to the frequency displayed on the LCD.

2) Handling Simultaneous Key Presses: If a user presses a key in error, the subroutines responsible for reading input from the keypad allow the user to correct the false key press. A key press is registered only when the key is released, and only the last key released is registered as input to the program. The logic that provides this error is shown in Figure 2 in Appendix A. Any time a key is released and an interrupt is generated, the key press subroutine checks to ensure that no keys are still pressed before it returns the value of the key that had been pressed.

For example, if a user presses '2' in error, then presses '3', releases '2', then releases '3', only the '3' will be registered as input from the keypad.

3) Displaying frequency on the LCD: Each time a number was entered using the matrix keypad, a function was called that would display the currently entered frequency on the LCD.

The user's last three inputs were stored in a CPU register as binary coded decimals; each of the user's keypresses required four bits in the register, so a number between 0 and $999Hz$ could be safely represented using 12 of the 16 bits in the CPU register. Each time a new number was entered using the matrix keypad, the contents of this register would be shifted four times to the left to make room for the new digit, and the numeric value of the new digit (0-9) would be logically OR'd with the value in the BCD register. Once this process was completed, bits 0-3 would contain the ones digit of the entered frequency, bits 4-7 would contain the tens digit of the entered frequency, and bits 8-11 would contain the hundreds digit of the entered frequency.

Once the register contained the last three values entered, an assembly routine was called that displayed each group of four bits on the LCD according to the following procedure:

1) Shift left (4 times for tens digit, 8 times for hundreds digit)
2) Add the ASCII value '0' (0x30) to the BCD digit, which is now in the bottom four bits of a copy of the register holding the BCD values
3) Display the character on the LCD

There was additional logic included in the program that would display the hundreds digit on the LCD as blank (space) instead of as a leading zero.

4) Blinking the LED at the Specified Frequency: Once the submit button was pressed, the

C. Errata

1) Port 2 I/O and the LCD: In order to keep all of the inputs and outputs required to interface with the matrix keypad localized to a single I/O port on the microcontroller, we chose to de-mux two of the I/O pins from the LCD.

Our tests of our program showed that no LCD segments relevant to to the operation of our program have been disabled. Everything displays correctly. However, any modifications to this program should bear in mind that there are likely segments of the LCD that will not function unless the I/O ports used to read the keypad are re-worked.

D. Accuracy of Generated Frequencies

At sufficiently low frequencies ($\leq 400Hz$) the frequency output by the microcontroller on the P1.0 LED is accurate to within hundredths of a Hz. At higher frequencies, however, the error between the frequency output on the LED and the frequency entered by the user will gradually increase. Generally, the frequency output at the LED will be lower than the one entered by the user. This is a consequence of low difference between clock cycle counts required to implement high frequencies. This program should not be used for applications that require high precision of frequencies in the $750Hz - 1kHz$ range.

III. Results

On startup or reset, the program scrolls a message across the LCD instructing users to enter a frequency between 1 and $999Hz$. If a user presses a key, the instructions will exit early. If a number is pressed to early-terminate the instructions, the key press will not be registered. The LCD then changes to display: 'F:' and a blank screen, indicating to the user that it is waiting for input. When a number input on keypad, the entered number is displayed as the least significant digit on the LCD. As more numbers are entered, the numbers shown

on the LCD continuously shift out the most significant digit and cycle the last number entered on the right side. Two digits are shown on the LCD if the most significant number is zero. Otherwise, three digits are displayed.

Once either the '$*$' or '$\#$' keys are pressed, the frequency displayed on the LCD is submitted. Power-of-two division and multiplication is performed to convert the frequency into a clock cycle count that can be placed into TA0CCR0 to control the LED frequency. The LED is controlled using only the timer peripheral, and can be set to blink with high accuracy at low frequencies and acceptable accuracy at high frequencies. The program works as specified for the entire range of frequencies from $1Hz$ to $999Hz$, and prevents the user from entering frequencies outside this range. If $0Hz$ is entered, the LED is turned off.

They keypad allows the user to 'modify' the key that will be submitted if they press the wrong key. If the user presses a key and realizes that they have pressed the wrong one, they can press another key, release the key that was pressed in error, then release the correct key, and only the correct key will register as a keypress. In general, only the last key released will register as a keypress if more than one key is depressed at the same time.

The results of tests for various frequencies between $1Hz$ and $999Hz$ are shown below in Table 1. Output frequencies are measured using a calibrated Digilent Analog Discovery 2.

Input Hz	Output Hz	Absolute Error ($\%$)
1	0.9977	0.23
5	4.9881	0.238
15	14.9643	0.238
16	15.9636	0.228
25	24.9420	0.232
50	49.8829	0.234
100	99.7676	0.232
200	199.521	0.240
500	498.661	2.678
750	747.938	2.749
999	995.771	3.232

Table 1: Frequencies generated by the microcontroller for inputs ranging from $1Hz$ to $999Hz$

IV. Conclusion

We wrote a program using almost exclusively MSP430 assembly that allowed a user to input a frequency between $1Hz$ and $999Hz$ using a matrix keypad, press one of two non-numeric keys on the keypad, and have an LED blink at the frequency they had specified. The subroutines responsible for handling keypad input allowed users to correct errors they made while entering the frequency; only the last key released for any given combination of keys pressed would be registered as valid input. The program handles invalid input by turning off the LED for a frequency of zero, and by using only the three most recently entered digits in the calculations that would determine the LED frequency.

We successfully implemented all of the features that were required for this project. At low frequencies, the accuracy with which we output frequencies is very high. As the frequency requested by the user increases, the actual output frequency begins to lag the desired frequency slightly, and the accuracy drops slightly.

Beyond interfacing with a matrix keypad, this project taught us that it is possible to deal more creatively with devices to save ports if ports on a microcontroller are scarce. The 'grid' layout of the wiring internal to the microcontroller was just one example of how a circuit component might be wired up in order to conserve I/O pins.

Planning the control flow for the logic that would later allow us to interface with the matrix keypad was the most time-consuming portion of this project. Though writing the code to implement our design was trivial, we spent a considerable amount of time planning:

- How the keypad would be physically connected to the microcontroller so as to prevent damage to the I/O pins and allow us to write read input from the keypad with efficient algorithms
- How we could most efficiently poll the buttons in the keypad
- How we could maximize our time in low power modes
- The optimal polling frequency for the columns of the keypad that would balance the need for response to user input with efficiency and power savings

Among the most puzzling parts of this project was finding an input port that was both interrupt-enabled and had enough pins exposed on the launchpad breakout that we could use only one port to interface the LCD. Had we known at the beginning of the project that the pins in port 2 that get mux'ed into the LCD were not essential to the operation of the LCD, we would have wasted less time and effort trying to find some creative way to use the ports.

A. Future Work

Interfacing with a keypad is a significant increase in I/O capability. While this project used the keypad to select an LED's frequency, future projects could use the keypad in a variety of ways such as entering data or controlling a basic GUI. Further, the use of external hardware could always be expanded allowing for functionality beyond what is built into the launchpad.

B. Questions

1) The majority of our program was written in assembly. Which C functions were called from the assembly?

   We called the LCD library functions provided by Texas Instruments, which are written in C, from our assembly code.

2) How did you handle multiple button presses?

   The subroutines responsible for reading input from the matrix keypad to not return control to the remainder of the program until no keys are pressed. When the last key is released, its value is returned as the key that the user entered. This allows the user to correct their error if they press the wrong key.

APPENDIX A
FLOWCHARTS

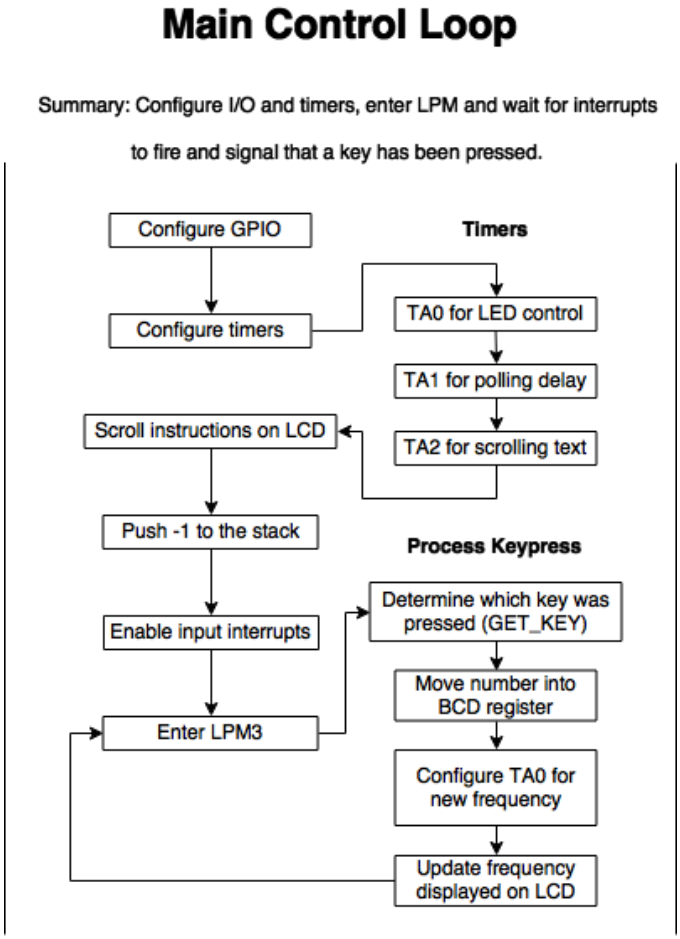

Figure 1: Flow chart for main program logic

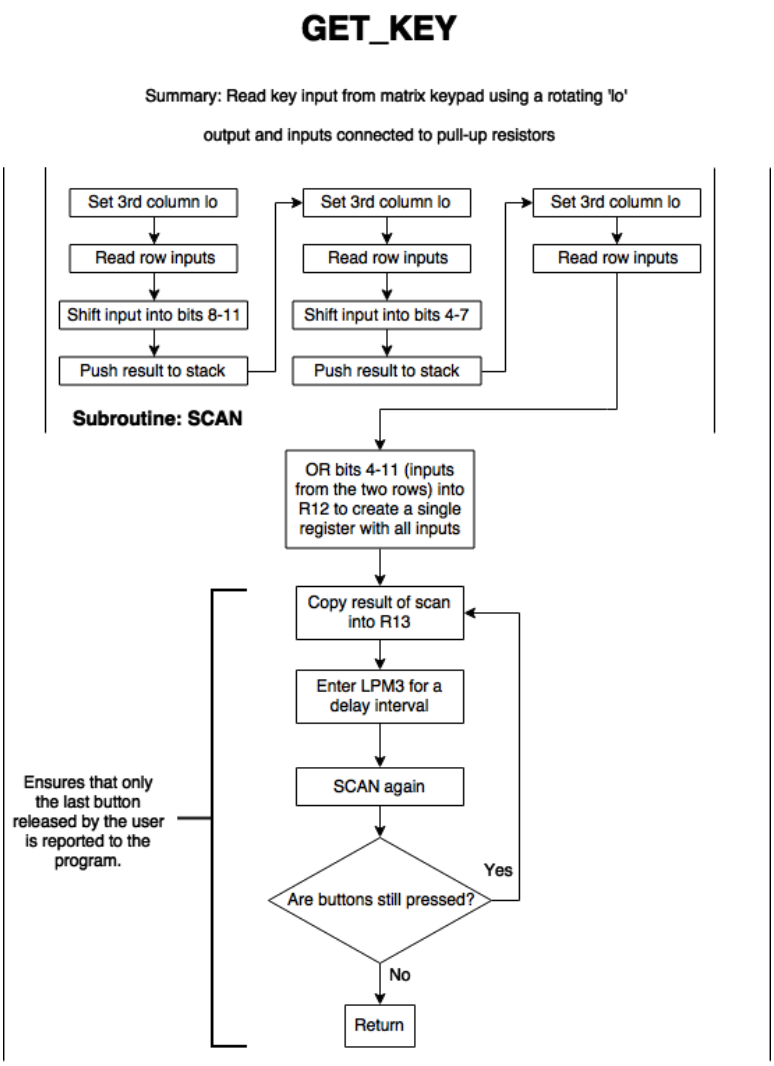

Figure 2: Flow chart for keypad logic

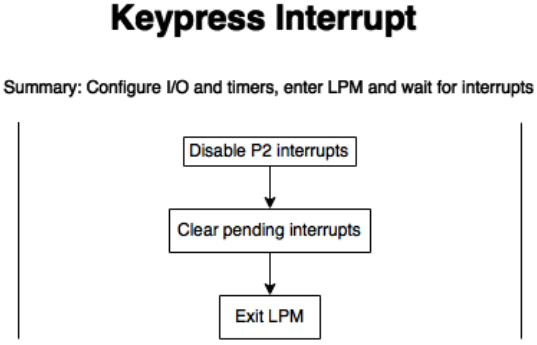

Figure 3: Flow chart for the keypress interrupt

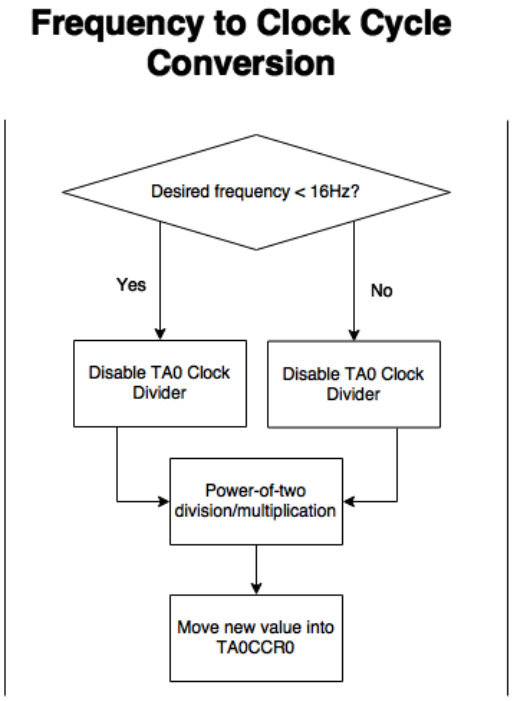

Figure 4: Flow chart for conversion between entered frequency and clock cycle count loaded into Timer CCR

APPENDIX B
HARDWARE DIAGRAM

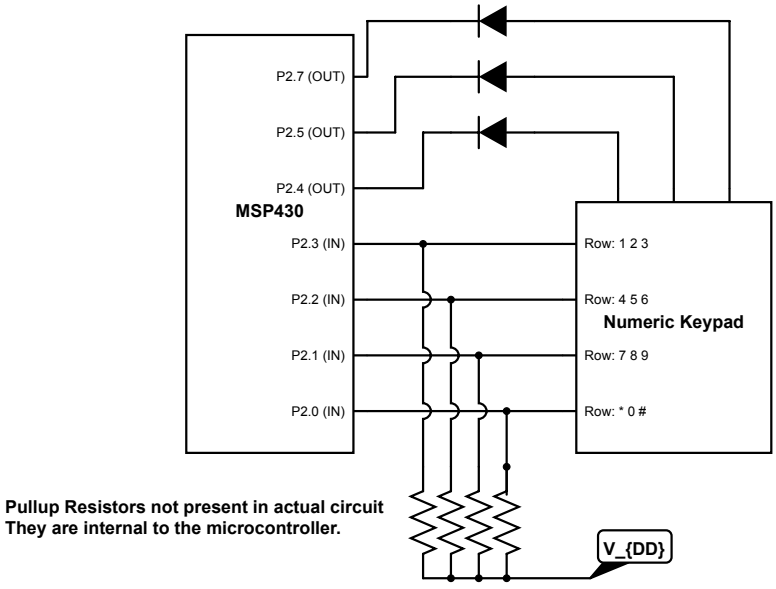

Figure 4: Wiring diagram for our keypad circuit. No external resistors were used - the microcontroller's internal pull-up resistors were engaged.

REFERENCES

[1] Ferreira, Bruno, "Which Schematic Is Better for Keypad Connection to an MCU?", Electrical Engineering Stack Exchange, July 2012
[2] Jameco Electronics, 12 Button Telephone Keypads, Golden Electronics Co., 1994
[3] Texas Instruments, MSP430FR698x(1), MSP430FR598x(1) Mixed-Signal Microcontrollers datasheet (Rev. C), MSP430FR6989 datasheet, Jun. 2014 [revised Mar. 2017].
[4] Texas Instruments, MSP43FR58xx, MSP430FR59xx, and MSP430FR6xx Family Users Guide (Rev. O), MSP430FR6989 Family Users Guide, Oct. 2012 [revised Dec. 2017].