EE 474: Lab Report 3

Michael Walsh, Josh Shackelford, Alex Finestead

This Page Left Intentionally Peckol

how bout tha 4 creativity points?

TABLE OF CONTENTS

Introduction:

In this third lab assignment we integrated the LCD display, used in previous projects, with a shift register in an effort to reduce the total number of GPIO pins used. A kernel was then written as a driver for the LCD display to further reduce the amount of user interaction with the code. Once the kernel was functioning, a step-sequencer music maker was created using the C programming language. This sequencer utilized the LCD display to give the user visual information on the operation of the step-sequencer. A piezo buzzer was used in conjunction to play the notes as directed by the user.

Design Specifications:

The design specifications for this lab were left intentionally ambiguous. Required was the use of an LCD driven by the BeagleBone Black with the use of an SN54HC595 shift register. Another being that the GPIO's for this lab assigned in Linux kernel space rather than user space. On top of this, the team decided that the kernel space would simply update the GPIO's based on the status of a file in user space. Using that functionality, a library would be built in user space and a project would be developed. A simple step sequencer was the project that would be developed, which is a simple music writing device.

Hardware Implementation:

Before anything could be run, the LCD display had to first be connected to the BeagleBone correctly. To do this, GPIO pins on the BeagleBone were designated for each of the pins of the display. The following table (**Table 1**) was used to direct our wiring. Pins 15 and 16 are unlisted, but are used to control brightness. These were hooked up to ground to prevent any possible fluctuations.

Pin Number	Symbol
1	V_{ss}
2	V_{cc}
3	V_{ee}
4	RS
5	R/W
6	E
7	DB0
8	DB1
9	DB2
10	DB3
11	DB4
12	DB5
13	DB6
14	DB7

Table 1: LCD Pin Assignments

A potentiometer was connected in series with the V_{ee} to give the user some control over the darkness of the LCD display. The potentiometer was set to a value of approximately 3kΩ; this gives the text on the display a dark enough texture to be visible without it being obscured by the background. An 8-bit shift register was placed on the breadboard (**Figure 1**). The eight output data pins of this shift register (Q_A-Q_H) were connected to the eight input pins of the LCD display (pins 7-14). Individual GPIO pins from the BeagleBone were connected to pins 14, 12, and 11 of the shift register as a way of controlling its functionality. With all of the pins of the shift register and the LCD wired, the total number of GPIO pins used was reduced from 12 to 6: a significant downsizing. The piezo buzzer was then connected to a pulse-width-modulator on the BeagleBone for use with the step sequencer. Once all devices were wired, we could beginning testing and initializing the hardware through software means.

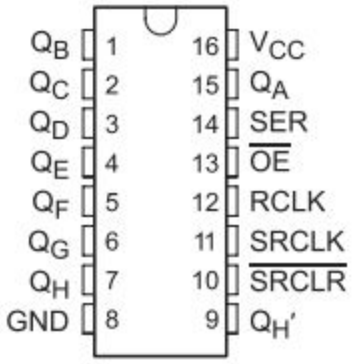

Figure 1: SN54HC595 Shift Register Pin Assignments

The BeagleBone Black's GPIO pins were hooked up in the following manner to the LCD module and shift register:

GPIO_60	DATA
GPIO_49	RCLK (shift clock)
GPIO_48	SRCLK (latch clock)
GPIO_115	RS (register select)
GPIO_20	RW (read/write)
GPIO_112	EN (LCD clock)
ERHPWM1A	PWM for buzzer

Figure 2: BeagleBone Pin Assignments

The following is a full wiring diagram of our BeagleBone Black, LCD module, speaker, and associated circuitry:

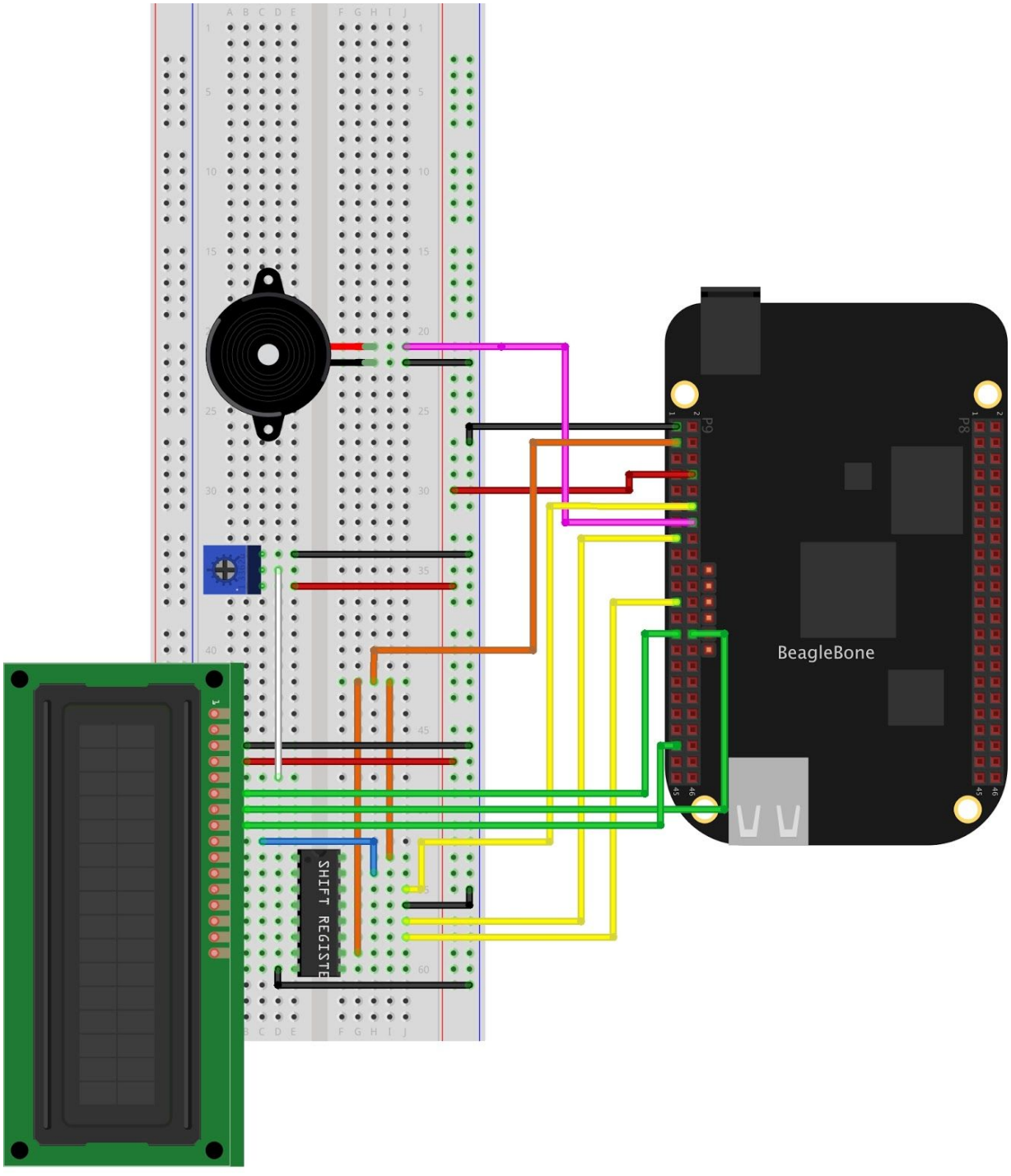

Figure 3: Full Wiring Diagram

The color coding for Figure 2 is as follows: For power, black, red, and orange are used. Black is 0V or ground, red is 5V, and orange is 3.3V. Green wires are LCD controls: RS, RW, and Enable. The white wire is the potentiometer output for LCD contrast. Yellow wires are shift register controls: latch, clock, and data. Blue wires are LCD data, and the pink wire is PWM

output. Note that the Hitachi Standard LCD pin 1 on this LCD is the third pin on this LCD, the wiring diagram reflects how it was wired in the physical world and not the standard across other standard LCDs.

Software Implementation:

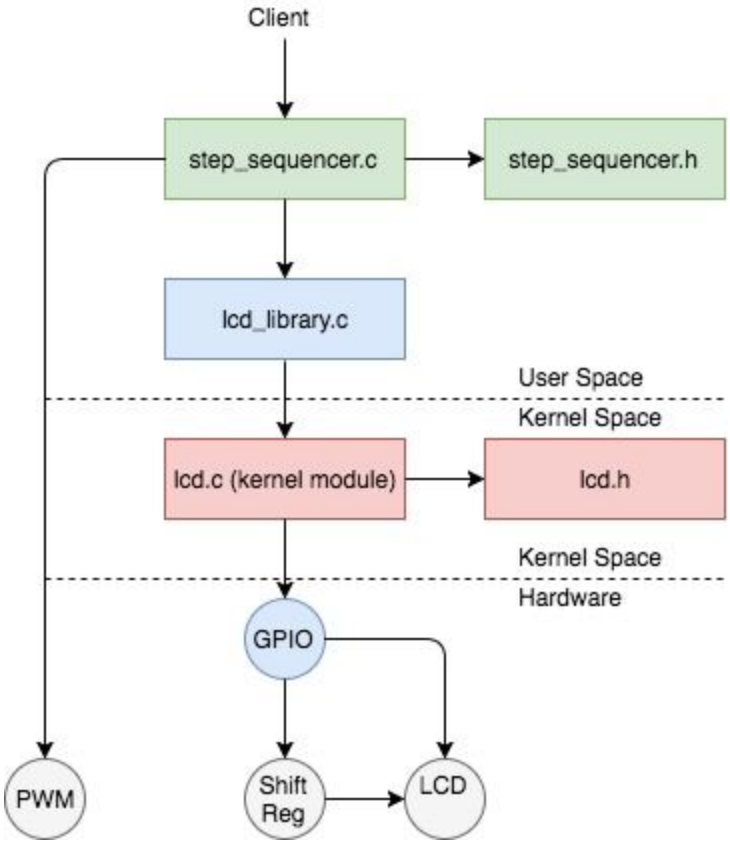

Figure 4: Software and Hardware Diagram

Creating a Loadable Kernel Module

As part of this project, a loadable kernel module (LKM) was constructed in C and loaded into the BeagleBone Black's Linux kernel. This kernel contains functionality for multiple different LCD operations. It runs the initialization process upon loading, presenting the user with a blank screen and a cursor. The kernel also contains functions for writing commands to the LCD and shifting data through the shift register. This kernel allows the user to simply call a *write()* function in user space code, and be able to see the text appear on the LCD. The process for setting up the kernel module is as follows:

- Insert the kernel using *$ insmod lcd.ko*
- Create the device by typing *$ dmesg* into the console and using the major and minor numbers given
- Ensure that the step-sequencer is compiled with *$ gcc step_sequencer.c*
- Run the program with *$ ./a.out*

The user is now in control of the program.

Implementing User-Space Functionality

Once the kernel module was created and loaded onto the Beaglebone, the next step was to create a user-space program that took advantage of the new kernel module. We decided to create a program that utilized the LCD display and allowed a user to input 16 numbers, ranging from 0 to 6, that represented notes on the Blues Scale. The program included several states allowing the user to input a 16-character string all at once, edit input a single note at a time, and even edit the timing for each individual note.

User Manual for step_sequence.c:

step_sequecnce.c is a user-space C program that is designed to utilize the functionality provided by the kernel module for the LCD display. The purpose of the program is to play user specified tones through a Pulse Width Modulator (PWM), displaying all menus and user interfaces onto the LCD screen. The user is able to navigate a series of states using separate keystrokes. Below is a list of all the program states the user can access.

States:

- Home—Boot state of the program. Prompts the user for input, showing all available key input options. Constitutes the reset state for poor input or mistyped commands.

- Input—Standard method of input. The user inputs 16 numbers, each between 0 and 6, with no spacing between them. Upon hitting enter, the program enters the Play state.

- Write—Specific method of input. Allows the user to change each note individually, using ‘l’ and ‘r’ (or ‘L’ and ‘R’) to navigate left and right across the note string. In this state, the default input is all 0, allowing the user to start from scratch.
  Once the user is satisfied with their note string, typing ‘p’ will shift the program to the Play state.

- Timing—Customization method for program. Allows the user to change the timing of each individual note in the input, inputting numbers from 0 to 9, representing the length of time each note will be played. The default state for note timing is 0.2 seconds for each note.
  Once the user is satisfied with all the timings, they can strike ‘i’ to input an entire string of notes, ‘w’ to input an editable string of notes, or ‘p’ to play the current string of notes with the current timing.

- Play—Output state of the program. When in the play state the program will output notes, via a PWM, based on the value of the notes the user input. The program uses the blues scale for its notes, 1-6 increasing in pitch with 0 being a rest, or silent note.
  Once the sequence has finished, the program returns to the Home state.

A library was built on top of this user-space code to further abstract the user’s experience. This library contained functions for writing individual characters and entire string to the LCD display. Also included were functions to set the position and appearance of the cursor, as well as shift

it left and right. All of these functions passed commands through the kernel using the *write()* function to talk to the LCD display.

Testing:

Testing was frequently conducted in this lab, in a variety of ways. In the initial testing of the shift-register's operation, the oscilloscope was extensively used. This allowed us to pass known data to the registers and attempt to shift it through, all while observing the output. Many errors, both hardware and software, were found this way. We also used the oscilloscope probes to test the values entering the LCD display. We were able to determine where the issues were in our wiring and in our code through this method.

The kernel module was significantly harder to debug. The *printk( )* function was particularly helpful for tracking down errors in our code. By running *dmesg* in the terminal, we could view all the statements written by our kernel. We knew that if a message didn't show up that should have, we had found a bug. We could pinpoint specific sections of our code that were causing problems and implement more *printk( )* statements to further understand the bug.

Testing of user-space code was done using *printf( )* statements in conjunction with *gdb*. The *printf()* statements allowed us to check specific values of variables and files without interrupting the flow of the program. GDB gave us the ability to break at specific problem sections in our code and slowly step through them while observing the program's functionality.

Presentation of Results:

All of the hardware and software function exactly as expected. The hardware properly makes use of a shift register to send data bits to the LCD display, which can display a variety of numbers, letters, and characters upon command. The kernel module can be correctly inserted and removed, and appropriately handles functions like *write(), open(),* and *close()*. This kernel runs on insertion and correctly initializes the LCD display. With these functions, the LKM adequately functions as a driver for the display peripheral.

The userspace code also functions correctly. It correctly takes user input and writes strings to the display in a variety of states. This indicates that the kernel module is being utilized correctly from userspace. The step-sequencer gives the user the ability to create a song with notes and timing, and play it back. The LCD shows the user's song and displays the program running through it as it is played back on the Piezo buzzer.

Analysis of Errors:

There was one major hardware issue that was encountered throughout this lab project, and several software bugs. All errors, however, were eventually resolved. The hardware issue involved malfunction of the GPIO pins, and was one of the first errors we had to address. Upon testing all pins with the oscilloscope, it was realized that GPIO 117 produced a consistent

triangle wave regardless of values written to the pin. This error was one that appeared in other lab groups as well, and so after trying to find a work around, we ended up moving to a different GPIO and not using 117.

In intensive testing, it was often the case that running 5V as main supply to the shift register would output data at too high a voltage. The maximum input voltage for controls and data of the LCD is 5V, and the shift register would output anywhere from 4.9-5.5V depending on the condition. Changing the power supply to the shift register from 5V to 3.3V remedied this, and acceptable voltages were achieved.

Debugging kernel code proved quite challenging, as some functions would not be called, and there was no simple way to step through the code. Consequently, all errors were debugged using *printk( )* statements. The kernel acted strangely with the data that was passed to it, as it's internal memory would not be updated until the write function completed. This proved challenging as in order for the kernel module to update GPIO's and the LCD screen, the values had to be updated on a write. A temporary array was called and bypassed the internal values in the kernel's storage holder to achieve the results needed.

Conclusion:

In this third lab, we designed a Loadable Kernel Module (LKM) and appended it to the Linux Kernel on our Beaglebone Black. We then implemented a user-space program that took advantage of the new functionality offered by our LKM. This lab gave the group first-hand experience wrestling with the Linux kernel and operating within the kernel space of an operating system.

This knowledge of kernel manipulation is sure to be useful as we continue work in the field of embedded-system design and implementation. With this lab's material, we have all the skills we need in order to write drivers for any hardware that we have the specifications for. This is a powerful skill as it means that we can implement custom hardware into our systems, and after the initial pain of debugging and applying an LKM, we have simple access to that custom hardware.

References:

[1] DOT MATRIX CHARACTER LCD MODULE USER'S MANUAL
https://class.ee.washington.edu/474/2017spr/lab_specs/Lab2/19988_Optrex_DmcSeries.pdf,
accessed 28, April, 2017

[2] SNX4HC595 8-BIT SHIFT REGISTERS WITH 3-STATE OUTPUT REGISTERS
http://www.ti.com/lit/ds/symlink/sn74hc595.pdf
accessed 28, April, 2017

[3] LAB 1: LOADABLE KERNEL MODULES (LKMS)
https://class.ee.washington.edu/474/ecker/lab_specs/Lab2/LKMDocs.pdf
accessed 28, April, 2017

Block Diagrams created using draw.io
Wiring Diagrams created using fritzing