Loadable Kernel Modules & Shift Registers
EE/CSE 474 Lab 3

Daisy Xu, Yunie Yi, Abigail Santos

November 4, 2016

Table of Contents:

1. INTRODUCTION

The purpose of this laboratory was to learn/familiarize ourselves with shift registers and loadable kernel modules. For the most part, this lab served to build upon the previous lab and as such, previously discussed design and implementation details will not be as thoroughly explained. There were three core parts to this lab. In the first part, we successfully integrated a serial-in parallel-out shift register from our set up in the previous lab. By using a shift register, we were able to use fewer gpio pins to write to the same 7 data bus lines used by the LCD. In the second part, we also built upon the last lab by developing a loadable kernel module which would replace the complicated software driver we previously built to control the LCD. This kernel managed basic functionalities and the setup of the LCD through the gpios and shift register output. It also controlled the initialization process and writing to the device.

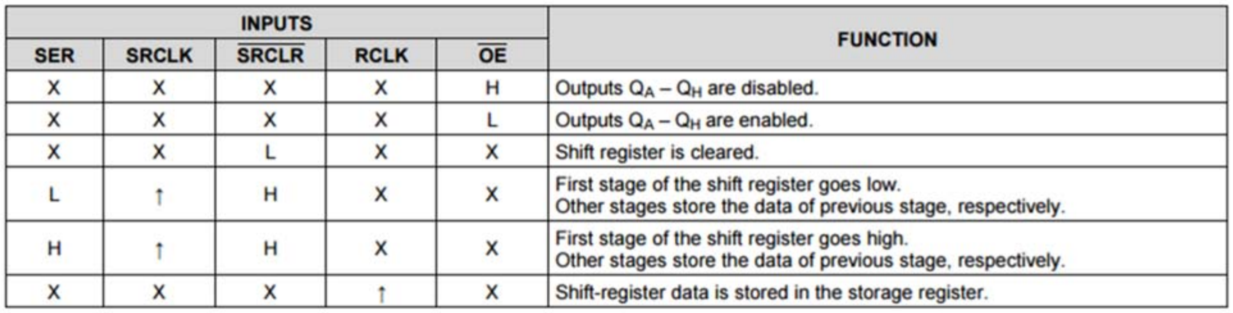
The third part of this lab was integrating our previous game written in userspace on top of the kernelspace code. Although discussed in the previous lab report, in summary, the game is a memorization game similar to Simon Says. The game starts by displaying three 'X' characters on the LCD in succession before disappearing. The user has to memorize the position of the three spaces for the brief moments that they appear. Then, controlling the cursor, the user must find and select the positions in the correct order to win the game. The controls and rules remained the same, however we modified our code to implement extra features. Namely, we implemented a

PWM to play unique musical notes corresponding to the location of the 'X's as they appeared on the screen. The player could then move the cursor and enter ‘p’ to play the sounds at each location without penalty. In this way, if visual memorization was too difficult, auditory memorization could help the player deduce where the 'X's appeared.

2. Discussion Of the lab & Specifications

2.1 Implementing the Shift Register

We used a serial-in parallel-out shift register in order to make our use of gpio pins more efficient. Essentially, the shift register will be able to output 8-bits at a time with only 5 inputs (other than power and ground) in exchange with a slightly longer time to output. These 5 inputs,



INPUTS					FUNCTION
SER	SRCLK	$\overline{\text{SRCLR}}$	RCLK	$\overline{\text{OE}}$	
X	X	X	X	H	Outputs $Q_A - Q_H$ are disabled.
X	X	X	X	L	Outputs $Q_A - Q_H$ are enabled.
X	X	L	X	X	Shift register is cleared.
L	↑	H	X	X	First stage of the shift register goes low. Other stages store the data of previous stage, respectively.
H	↑	H	X	X	First stage of the shift register goes high. Other stages store the data of previous stage, respectively.
X	X	X	↑	X	Shift-register data is stored in the storage register.

including the Serial data input and 4 other control bits are summarized below in **Table 1**.

Table 1: Summary of Function of Inputs of Shift Register

The shift register outputs 8 bits from QA - QH, which we directly hooked into DB7 - DB0 of the LCD respectively, as seen in **Figure 1**.

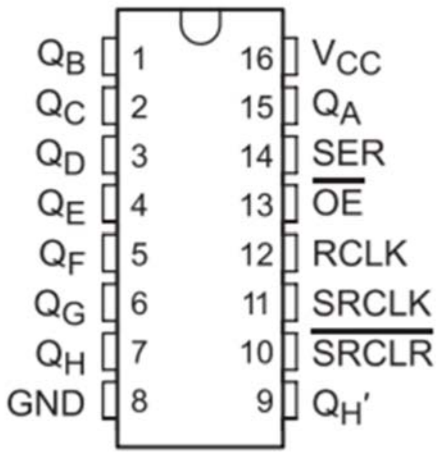


Figure 1: Diagram of Shift Register

The five control inputs are connected into GPIO ports and initialized in gamespace.c as seen in Table 2

INPUTS	GPIO #	Description
SER	26	Serial Data Input
OE	46	Output 8-bit Data from Storage Register (Low True)
Rclk	47	Storage Register Clock
SRclk	69	Shift Register Clock
SRclear	27	Clear Shift Register (Low True)

Table 2: Implementation of Shift Register in 'gamespace.c'

2.2 Converting Driver Software to an LKM

In the previous lab, we wrote all of our initialization in userspace. Instead, we took these basic functionalities and implemented them within kernelspace. In this way, initialization and methods such as read/write that interface with the LCD are done in the LKM. This allows software, such as our game, to run on top of the LKM separately. The methods for our LKM are described below in **Table 3**.

Method Name	**Method Description**
initializeAll(void)	The instructions for LCD setup works same as the previous lab. It initializes LCD by sending a set of instruction.
initGPIO(void)	Initializes all GPIO ports we used for the Shift Register and LCD by exporting the pins and changing direction/initial value of pins.
sendChar(char command)	Takes a valid LCD command as a character input to be sent from the GPIOs to the Shift Register to the LCD.
Ssize_t device_write()	Allows the beaglebone to write to the LCD using a given file. Keeps track of cursor position and how many characters have been written.
Move Direction: moveUp/Down/Right /Left(int place)	Takes current place of cursor on LCD and moves the direction one space over.The functions will not allow the cursor to move beyond visible positions on the LCD.

Table 3: Methods in 'gamespace.c'

2.3 Modifying, Integrating, and Enhancing the Memorization Game

After establishing core functionality in our LCD interface and pipe program, we were able to develop a memorization game. Though discussed in the last lab, the game works by displaying an 'X' in 3 randomized spots on the LCD in brief succession. Each time an 'X' is displayed, a sound unique to the column location of the 'X' is simultaneously played. The goal of the game is to memorize and select where the 'X's appeared using the keys to control the cursor. However, the user can also play the sounds corresponding to any cursor position without penalty. Because every column has a different sound, this provides a second clue as to where the 'X's are. If all 3 spots were correctly found, a “hard mode” with 5 spots will begin.

The controls are all the same as the last lab though the added sounds corresponding to each column can now be played with ‘p’. Likewise, the methods of game.c are much similar to before, and are still designed to work in tandem with a pipe that reads from the terminal. These methods are summarized below in **Table 4.**

Method Name	**Method Description**
showSpots(int difficulty)	Generates and displays random spots on the LCD which the player must memorize. If difficulty = 0, generates 3 spots. If difficulty = 1, generates 5 spots.
closePort()	Closes GPIO ports. Called at the end of the game.
initPWM()	Initializes PWM. Duty Cycle controls sound output. Each column on the LCD is assigned a unique pitch to aid in recognition, controlled by the PWM Period.

Table 4: Summary of Game Methods

Much of the code that controlled the game was written in the main method of ‘game.c’. The main method also contains the logic for reading the user commands, checking if the user successfully guessed or not, and initializing easy/hard modes.

3 TEST PROCEDURE & CASES

3.1 Implementing the Shift Register

Case 1: Signal Verification w/ LEDs Connected to Shift Register Outputs

We first must determine that the Shift Register was correctly set up and and that the 8 outputs match what we desired. We can do this by connecting 8 LEDs in total, one for each Shift Register Output (QA-QH), to verify the values of each output. For example, if our program executes the sendChar(char command) method through the shift register, the 8 LEDs should be lit up to represent DB7-DB0 of the LCD command.

3.2 Converting Driver Software to an LKM

Case 1: Initialization of the LCD in LKM

The initialization process on the kernel module must be run in order to run other complex programs. To start, we must make a basic kernel that would only run this initialization. The command ‘insmod lcd.ko” can be used to run the kernel file and “mknod /dev/ lcd c” to run the linux command terminal connected to it. To confirm each initialization instruction was being properly sent, dmesg would be used to display the current command passed to our sendChar() method each time it was called.

Case 2: Test Read/Write Functions of LKM in Linux Terminal

Once our LKM is compiled successfully, we added in the device_write method to the file to write directly onto the LCD display. This is accomplished by use of the ‘echo [characters] > /dev/[file name]’ command to write the given data from the Linux terminal, and if working successfully, should display whatever characters the user types.

3.3 Modifying, Integrating, and Enhancing the Memorization Game

In this stage, we had to integrate all the functionality tested in 3.1 and 3.2 to further build upon it to run a more complicated kernel. This kernel would support the game written in userspace. The game,‘game.c’, would be developed with the additional function of sound, by using the PWM output unique pitches for each cursor position. Basic functionalities of the game were tested as seen in the cases below.

Case 1: Test Display of Random 'X's

If our game.c is compiled successfully, it should first generate and display 'X's in random locations on the LCD. In easy mode, as in the previous game, the program displays 3 spots, with 5 spots for hard mode. Our C code for this method would be designed to generate random integers corresponding to a visible location index on the LCD. The writing methods should take this location and write the character 'X' to each location, displaying each 'X' for a few seconds.

Case 2: Test Read/Write Operation of Moving Cursor

This case is to test if users can control the cursor direction with the keys ‘a’, ‘s’, ‘d’, or ‘w’ entered in the Linux terminal. If the ‘showSpot()’ method to display the ‘X’s worked successfully, the ‘GAME START’ message is displayed. Then, users can type any of the 4 keys, one at a time, and the cursor on the LCD should move in the corresponding direction. It is possible to use the printf() function to print the current cursor index after each move to the terminal. This can can be checked against the actual index seen on the screen. We also checked how the game would perform when more than one character was entered at a time. Because we designed our code to only read the first character in the buffer, anything beyond one character was ignored. In the instance where a wrong key (i.e. a key that was not ‘a’, ‘s’, ‘d’, ‘w’, or ‘y’) was entered, the program did nothing, just as designed.

Case 3: Test PWM of Cursor Spots

Since our game has additional function of sound via a PMW, the program should be executed such that any spot on each of the 16 columns on the LCD can play a unique pitch. Also, if the user tries to move the cursor beyond the boundaries, the PMW should not generate sound.

4. RESULTS

4.1 Implementing the Shift Register

Through testing, the Shift Register proved to be integrated successfully. Given any 8-bit LCD command, the register was able to read in the command sent with a serial data bit, and properly output the same command to QA-QH.

4.2 Converting Driver Software to an LKM

We successfully made character device in the Linux kernel which is a foundation before running more complexes. All the test cases were success and the device in LKM was ready to run a game

4.3 Modifying, Integrating, and Enhancing the Memorization Game

All functionalities of memorization game worked completely on LKM. The incorporation of the pipe with our LCD interface was successful. Whatever was typed and entered into the terminal correctly was written onto the LCD display. Overflow text on the first line continued onto the second, and vice versa, such that no text was ever lost. And each cursor successfully displayed unique pitch when users pressed ‘p’ in the Linux terminal.

5. ERROR ANALYSIS

Our group tried four different shift registers and all of them did not work; some combinations of output pins are broken when we tested 8 LEDs for each shift register. So, we tried to replace the broken port which is output to DB3 of the LCD with one of GPIO ports available in the beaglebone. In ‘sendChar’ method of our kernel module, we manually assigned the GPIO port of DB3 (index of 3 in the LCD pins) in looping eight time clock cycles of the shift register.

6. SUMMARY AND CONCLUSION

In this lab, we covered basic functionality and logics of character devices for further development of embedded programing. Linux Kernel Module was helpful in that it can incorporate many complex programs written in various language such as Python or C. In device efficiency perspectives, LKM makes the run time faster by having the foundation of LKM and running more than one files over it. Also, we learned that how the shift register we used in this lab; the shift register minimizes the uses of GPIO ports in that it needs only one serial input data and outputs 8 bit data. Although we had troubles in implementing the shift register, we learned all the basic functionalities of it and we were able to optimize our GPIOs of the beaglebone. It will be more useful when we need a lot more input data for more complicated programs. SIPO is the most common and efficient way used in other communication protocols and it also reduces error which can be caused from hardware components controlling I/O ports.

In conclusion, with two basic concepts of LKM and shift register, we were able to develop the memorization game from the previous lab further and to optimize the efficiency in terms of error reduction and run time.