Lab Report 3

LKMs and Shift Registers

EE 474 Fall 2016

Dr. Allan Ecker

10/21/2016

Bryan Bednarski (1337864)
Justin Allmaras (1329197)
Colin Summers (1332139)

1.0 Introduction

This report details our designs for EE474 Lab 3: LKM's and Shift Registers. The first step in this process is to compile a kernel and run a kernel on our Beaglebone Black board. This kernel image, once compiled, allows us to construct a series of loadable kernel modules as device drives to be loaded and run as separate procedures. We achieved the first major outcome of this lab by creating this kernel image and converting the code from the Lab 2 LCD driver to proper kernal syntax, compiling and loading it to our kernel. This program allows the user to echo strings directly to the /dev/new_char folder, and have each character within that string displayed on the LCD screen. In previous labs, we acheive this functionality with a series of pipes that took user input for write modes on the LCD (scroll and free write) and passed data into our driver-side code. Due to the kernel module, we were instead forced to instead achieve similar functionality with a series of memcopy() commands. In addition to a number of limitations to our programs structure, we added a number of functions required within all LKMS including licensing information, __init__ and __exit__ functions. Our approach to the design of these functions and other program modifications is described below in our design specification.

With the LCD functional and an understanding of writing and reading information from the user to the kernel space, we began implementing basic motor-control functionality in a separate kernel module. This module, called motor_driva.ko, allows the user to write a series of directional commands to the motor_driva device folder, and have the motors on the tank follow accordingly. Controlling the functionality of the motor H bridges, we provided this module with forward, reverse, left, right and stop functionality.This module can be run in parallel with the LCD driver module, and demonstrates our kernels ability to run multiple LKMs if provided with enough space in memory. Following our description of the LCD kernel module, we will explain to implmentation of this motor driver and its implications in future labs.

2.0 Design Specification

This section details the design process taken in the design of our kernel system for the beaglebone black. This kernel supports the addition of Loadable Kernel Modules (LKMs), which act as individual drivers for hardware and other subsystems that will be implemented throughout the remainder of this course.

2.1 Beaglebone Kernel

A Linux Kernel can support three types of drivers, character, block and network. For the extent of these labs, we will be writing mainly character drivers that support I/O data transfer through byte stream controls. Character drivers act as an interface between hardware and software in the user space. These drivers are crucial to interpreting the input from a user, processing commands and communicating with hardware through a series of define hardware controls. In this lab, our LCD driver interprets user input to the terminal, and parses strings into 16-character blocks. Next, the functions within our new_char driver write those character interpretations to a shift register and the LCD screen. Similarly, the motor control module receives input from the user, interpreting a short list of characters as commands and sets values on the motor H-bridges to control their functionality. This hierarchy within our kernel is shown in **Figure 1** demonstrates how the character drivers that we are writing act as an important part of the kernel datapath.

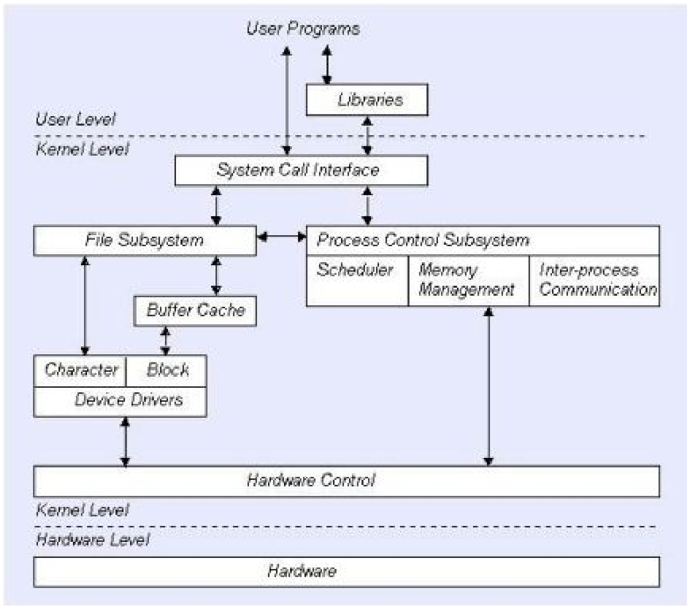


Figure 1: Kernel level hierarchy, module interface between hardware and software systems

After compiling your kernel and inserting it into the Beaglebone, there are a few things to be aware of during the creation of the kernel modules. First, kernel module writing is guided by a number of syntactical changes to non-kernel code printf() statements no longer work, but printk() can be used to write to the kernel's debug log if necessary. Additionally, each module must include a license specification to match module programs with the correct linux kernel version. All interactions with the character driver must be mediated by the character driver folder called /dev/*program*. The programmer can interact with those files by writing text to the \dev*program* file.

Ultimately, the kernel space compilation will only need to be written once for this class. With the kernel set up, we are able to load any number of kernel modules to the kernel space and operate them with a series of module commands. A critical factor in kernel space operation is the amount of memory is available to the processor. On one occasion, we were unable to load both kernel modules simultaneously having run out of computer memory to load a second. Open clearing old files from previous labs, we were able to easily run both modules. Later sections will discuss important applications of character drivers as we detail our design process for the LCD and motor control drivers.

2.2 LCD Driver Module

We reimplemented the LCD hardware from Lab 2, but changed from using a parallel data bus with 8 Beaglebone GPIOS to using a shift register to convert Beaglebone serial data to parallel data using 8 shifts to broadcast the 8 bits of data. We used a SN74HC595 shift register to accomplish the serial to 8 bit transition. See **Appendix A** for the logic diagram of the shift register. The SN74HC595 uses one data pin and two clock pins as well as an output enable and reset pin. We tied the output enable to ground through a pull down resistor to ensure that the device was always outputting. We also tied the reset pin high through a pull up resistor since we had no need to reset the data. This left us with three pins to control from the Beaglebone: serial data, serial clock, and refresh clock. The basic logic of the shift register for sending one byte of data is as follows:

1) Output bit one of serial data
2) Strobe the serial clock high then low
3) Repeat steps 1 and 2 for all 8 data bits
4) Strobe the refresh clock to push the 8 bit data to the output of the shift register
5) Set the control pins of the LCD
6) Strobe the LCD clock to read the data into the LCD

See Figure 2 for the signal patterns for sending the character ‘a’.

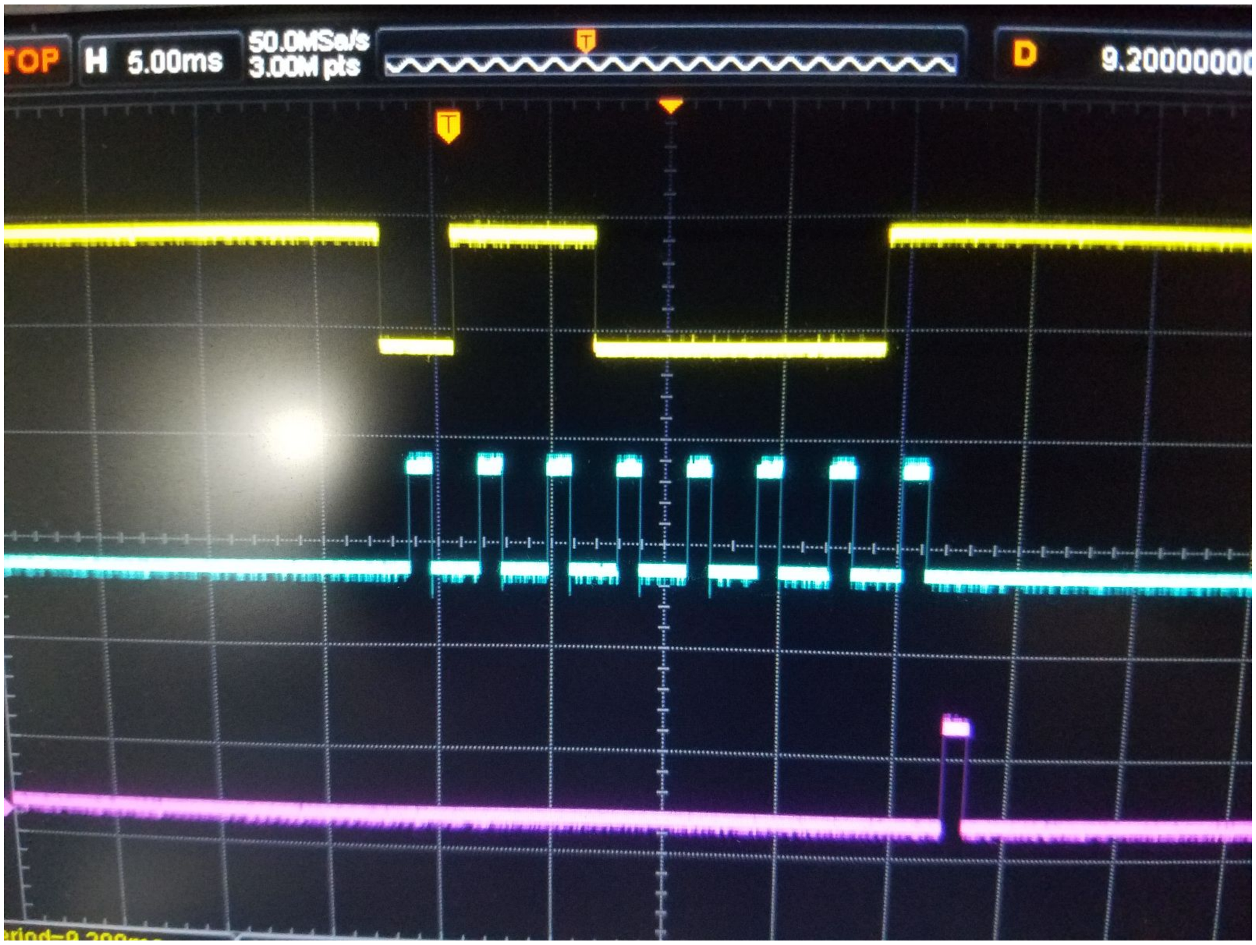


Figure 2: Shift register signals for sending the character ‘a’

In Figure 2, the yellow line is the data line, the blue line is the shift register clock and the pink line is the refresh clock. Due to the pinouts of the shift register and LCD we needed to send the most significant bit first so that it was shifted down to the most significant bit of the LCD screen.

For the LCD functionality, we implemented a scrolling LCD screen much like a teleprompter. Writing a string of characters to the lcd dev file would print 16 characters at a time to the LCD shifting the previous text up a row and eventually off the screen. In our previous lab, we implemented this functionality using a series of pipes to receive input from the user for both commands and strings to be printed to the board. This time around, we used toe \dev\new_char folder to receive input strings from the user space, and decided against using pipes to move data from the user space to the kernel module. Instead, we use a function called device_write, that reads input strings to the \dev\new_char folder and passes characters to a buffer called bufSouce. This buffer also recognizes the total number of characters printed to the console, which was critical for string processing without using the null terminator. (We decided against using the null terminator, ‘\0’ due to its increased risk of introducing a bug in the movement of data and conditions looping based off of the occurrence of these symbols). When a string is received, this function sets a 16 character buffer called bottom, to the first 16 characters in the string. Afetr a 1.5 second wait, it copies that data to another 16 character buffer called top, and prints to the top row on the LCD screen, while writing and printing another 16 characters to the bottom row. This copying and moving of data between these two buffers is done through the memcpy() and memset() functions.

2.3 Motor Driver Module

We also build a simple motor driver kernel module based on the TB6612FNG dual DC motor driver. See Appendix B for the application information for the TB6612FNG driver. Much like the loadable kernel module for the LCD, we controlled the motor driver through a character device file. The following commands were implemented in the kernel module:

F: Drive forward
L: Turn left in place
R: Turn right in place
B: Drive backward
Any other character: Stop

This functionality was implemented through the use of 6 GPIO pins, three for each motor. Each motor pin has two input pins to determine the direction and a PWM input pin to determine the speed. For this lab, we did not implement PWM and instead wrote a 1 (100% duty cycle) when we wanted to drive a motor and a 0 (0% duty cycle) when we wanted the motor off.

The motors are driven using a dedicated 7.5V source that provides power to the motors only. The supply shares a common ground with the Beaglebone as the motor driver breakout has no provisions for isolating the motor ground from the digital input signal ground. For the small motors on our rover we do not expect that this will be a problem.

3.0 Conclusion

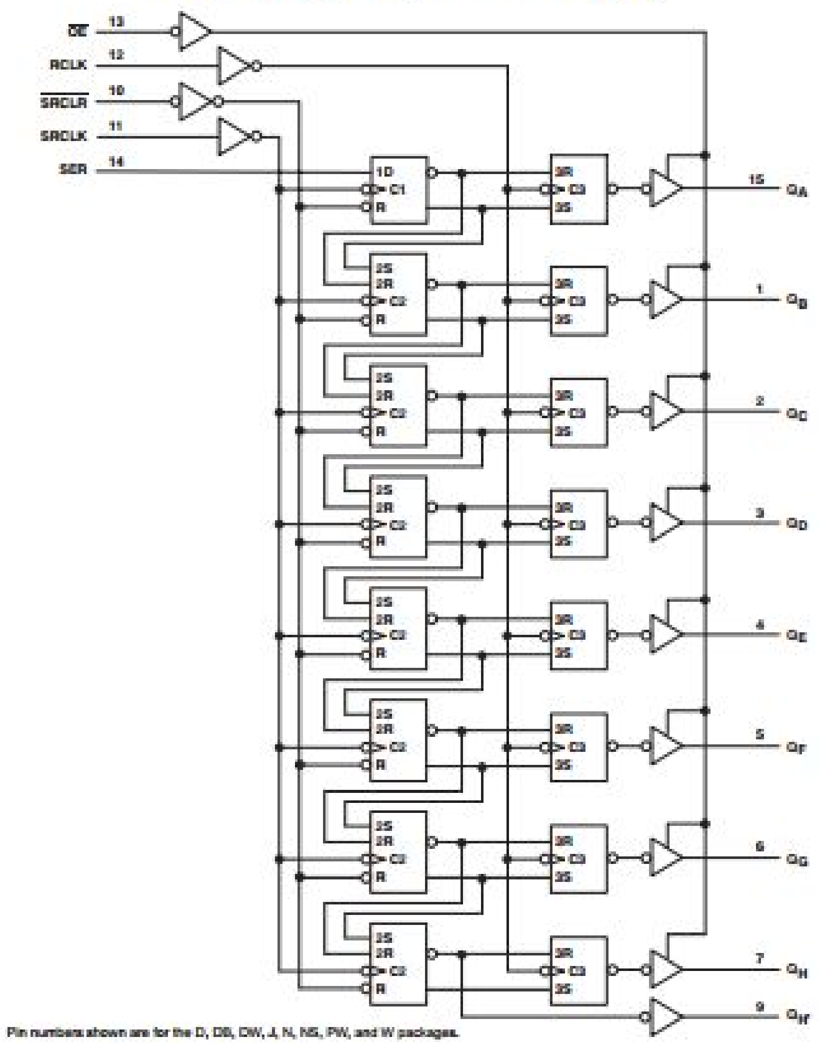
The kernel space and modules developed for this lab represent a huge milestone in our progress in this course. The final functionality of our project, controlling the LCD screen and motors, is completely possible using methods from previous labs. However, in using the kernel module, we have created a true loadable embedded system, that can function independently of other processes on the beaglebone processor. This is shown by having both kernel modules functional at the same time.

In building the kernel, we did run into a few issues that held up our development process. First, we had an issue in the first week of the lab with dependencies of the kernel system. These issues were easily solved once the TAs shows us which files to edit. We also ran into certain issues with the compile kernel when pushing and pulling the files from git. We solved this problem by removing all .git files from the beaglebone kernel code retrieved from our class source, and were then able to package the code to a .tar.gz and work on it at will. As far as the kernel modules go, we really only experienced some difficulties when working with string processing of inputs to the LCD display \dev folders. When printing to the LCD, we continually saw small degree symbols printed to the screen. This issue is not the null terminator, which was originally being used to clear the 16 character line buffers. However, we do believe that the problem was associated with those null terminators, because when we restructured the code to eliminate the null terminators, we did not see the problem again.

While we had to restructure and reformat most of our code from the previous lab to work with our kernel module, we will not have to do that for next week's lab. Rather, we plan to build off these same two kernel modules to build in additional functionality for the motor controls and the H-bridge chip.

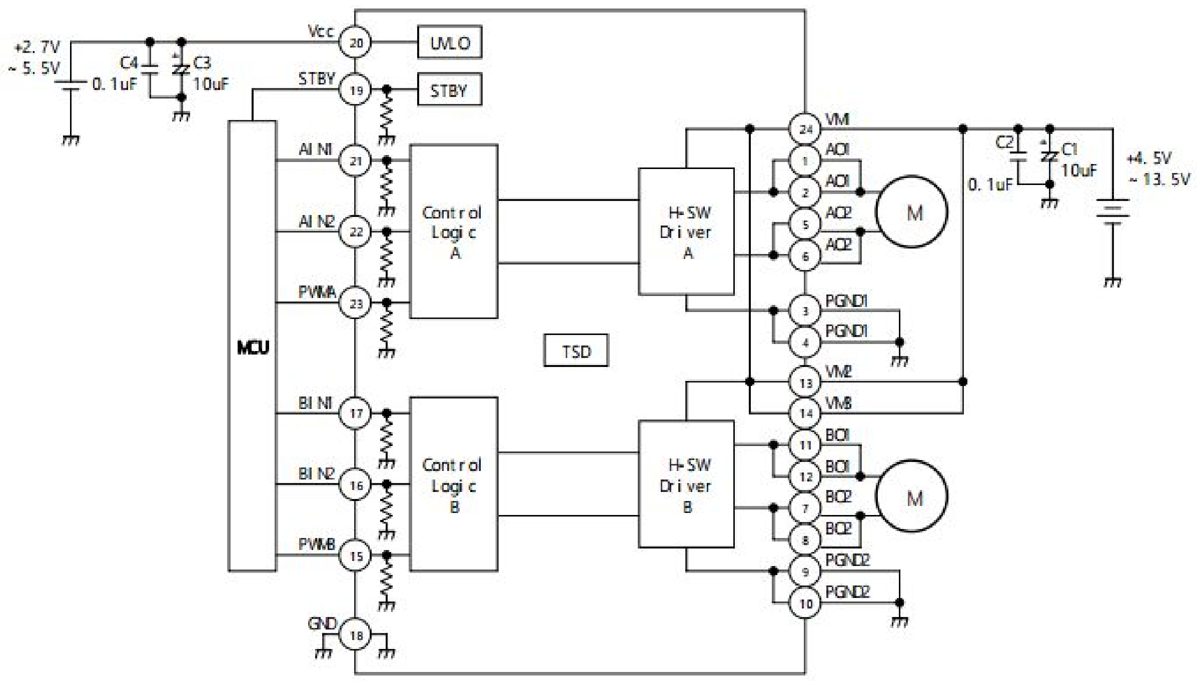
4.0 Appendix

Logic Diagram (Positive Logic)



Pin numbers shown are for the D, DB, DW, J, N, NS, PW, and W packages.

Appendix A: Shift Register Logic Diagram



Appendix B: TB6612FNG Application Diagram