Lab Report 2

Inputs, Outputs, and Time

EE 474 Fall 2016

Dr. Allan Ecker

10/21/2016

Bryan Bednarski (1337864)

Justin Allmares (1329197)

Colin Summers (1332139)

1.0 Introduction

The purpose of this laboratory is to become familiar with operating a 16x2 LCD display using GPIO interfacing on the Beaglebone Black board. The process of controlling the LCD display builds on the progress made during the first laboratory, in which we controlled a series of LED lights by toggling the GPIO pins. For this activity, we built a C driver for the LCD board, by controlling eight data pins and three control flags for the screen. By sending eight data bits in parallel, we are able to send ASCII characters to the display memory addresses on the LCD screen, shift the cursor and write to the subsequent character block. The initial goal of this project was to send a single character to the LCD display through a pipe that allows data to be sent between a user space program and the driver that is running. In addition to being able to send a single character, we created two settings for the driver, a “free” mode and a “scroll” mode. Through the command line, we are able to switch between display modes on command. During “free” mode operation, the user is free to write any strings of characters to the board, of with the first 32 ASCII characters will be displayed. In scroll mode operation, sixteen character strings of the user’s text are rotated from the bottom line of the LCD to the top, when pushed by another line. We implemented this functionality in order to pipe 16 character strings from a text file and proceeds to scroll through the opening sequences of the Star Wars saga.

2.0 Design Specification

2.1 LCD Components

The overall LCD driver is laid out over three c files that define the functionality of the GPIO pin drivers, overall LCD driver, and LCD controls for processing data and running various operations. The subsections below describe the functionality of each major division, and the functions that define their robust and versatile functionality for application in future labs.,

2.1.1 GPIO Initialization

The gpio.c file define the functions used to initialize and operate each of the GPIO pins necessary to drive the LCD screen. In total, 11 GPIO pins are necessary to drive the board. 8 pins are set aside as a parallel data bus for the characters sent to the screen. These 8 bits, 1 byte, account for enough data to define any ASCII character to be printed. Upon initialization of a unique GPIO pin, file points must be defined for the value and direction folders that are later written to control the current state of the GPIO. We also designed a function to quickly set the value on each GPIO, to be quickly called from the LCD function.

2.1.2 LCD Drivers

The LCD driver file contains the bulk of functions and processes responsible for the functionality of the LCD board. This function first defines and initializes all GPIO pins on the beaglebone to be used. In addition to the 8 data pins, this function initializes the read-select, enable, and read/write pins that control processes on the LCD. These pins are necessary because there will not always be data on the LCD data bus, or the data on those pins will not be needed. Therefore, the operation of these three flags makes up nearly the entire control loop for the LCD driver. After initializing each pin, the LCD screen is still not functional and requires a detailed initialization sequence to be executed before it displays a cursor and is ready to receive and display character input. This sequence is determined by the order of values passed to the LCD screen on the 8 data bits, and the time allocated for the LCD screen to process that command. **Table 1** below details the sequence of commands, time delays requires for each and their importance.

Initialization Command	Options Presented	Time Required
POWER ON	n/a	>15ms
Function Set Command	none	>4.1ms
Function Set Command	none	>.1ms
Function Set Command	Character Font	100us
Function Set Command	Number of lines	1us
Display OFF	none	1us
Entry Mode Set	increments/ decrements DD RAM address	1us
Display ON	none	1us

Table 1: Sequence of LCD screen initialization commands

Once the LCD screen is initialized, it is ready to be written to. However, the writing process for each character requires the control pins to be set to and data to be printed to each GPIO value folder. To reduce this complexity, it is necessary to abstract these processes with additional functions in the LCD driver. These functions will later be used by the LCD Controls when characters are piped from the user space to the LCD driver. Descriptions for the most important of these functions are described as follows:

- sendData(int data , int writeNRead , int regSelect) : This function receives an 8 bit data bit as an integer extracts each data bit's value, writing it to the value file for the respective GPIO pin
- sendChar(char C): sends a character to the LCD driver
- cursorShiftRight(int spaces): moves the cursor right by "spaces" on the LCD screen to the location that the next character will be printed
- cursorShiftLeft(int spaces): moves the cursor left by "spaces" on the LCD screen to the location that the next character will be printed
- clearDisplay(): clears the current display for next series of characters to be written

We implemented more functions than above, but these make up the core functionality of our LCD driver.

2.1.3 LCD Controls

The LCD_control.c file contains the main, top-level function for our LCD driver. This function is responsible for calling initialization sequences for the LCD and defining the client-side functionality of the pipe stream. In this function, two pipes are created. One pipe is responsible for setting the mode of the LCD and the other is for passing data to the LCD driver. We implemented two modes and have the ability to easily add more by mapping more special characters to different modes. The two modes we designed were "scroll" and "free." Free mode allowed the user to print up to 32 characters to the LCD. The driver would print the characters passed to the data pipe to the LCD starting in the upper left. Scroll mode allows for data to scroll vertically on the LCD. When in this mode, data passed to the data pipe will be printed to the bottom row of the screen, up to 16 characters, and the data on the bottom of the screen will be moved to the top row of the screen. Any data in the top row is shifted off of the display.

The driver allows for switching of modes at any time. To switch to scroll mode, a '#' character must be passed to the LCDCommands pipe. To switch to free mode, a '$' character must be passed to the LCDCommands pipe. When the mode is switched, the display is cleared and the cursor and blink functionality is removed.

The two pipes solves multiple problems that a single pipe driver may face when implementing several different modes. Having a dedicated pipe for the commands allows the data to remain uncontaminated with any mode information. A way to implement different modes with a single pipe would be to add a special character to the beginning of any data sent, which is an added constraint on the client as well as an expensive operation to concatenate the strings. Also, the mode is updated before the data, which allows for simultaneous writing to the pipes and the mode will be updated in time to print the data to the screen.

The basic sequence of the LCD control is as follows. The program opens the two pipes and waits for a small amount. It then reads the two pipes to see if either was written to. If either pipe was written to, a flag is set to signal to run the rest of the logic. At this time, the pipes are closed. The program then either changes the mode, or prints to the screen, depending on the information sent. This process repeats until it is ended.

2.2 Userspace

While being able to send single or two line statements to the screen via the terminal is useful, a more flexible and robust method of data delivery is needed for an enhanced user experience. We designed a few tools that allow the client of LCDScreen to parse data from a text file to and pass the subsequent messages to the screen.

The first of these tools is located in the file pipe_writer.c. This file features a function lineWrap(char* str, int wrapLen) which wraps the null-terminated string str into lines of length less than or equal to wrapLen without splicing words. For example:

```
char *str = “Hello world welcome to the jungle”
```

Becomes:

```
“Hello world
Welcome to
The jungle”
```

When lineWrap(str, 12) is called. Note that this modification happens in place by inserting newline characters in the appropriate places where spaces are located. This function relies on words being separated by single character spaces. Once this string is modified, a for loop iterates over the delimited string and writes each string chuck, which is less than 16 characters in length (the maximum number of characters the display can print per line), to the named pipe LCDData which is being read by the driver. The end result is that we can parse nearly any space delimited file and dynamically scroll the content across the screen.

We note that this file library is still a work in progress. Several comments, redundant literal values, inefficient code, and poor design patterns are present in the file. Moreover, one of the functions that we implements, centerText, was not ready to ship in time for our demo. We plan to redesign and refactor the section of code.

2.3 Scheduler Jitter

A CPU scheduler is responsible for deciding which memory processes will be executed at some point in time. The scheduler functions via a frequent hardware clock that interrupts the processes in the task queue and checks whether or not another process needs to be prioritized over the current process. To observe this effect we wrote a simple program that toggles a GPIO data pin high and low with as little delay as the system would allow, 1 us to allow the system to process a sleep command without significant delay. This process allows the scheduler to run as quickly as possible, demonstrating the time difference between regular intervals and the delay of the scheduler. Figure 1 below shows an example of the beaglebones scheduled processor jitter that interrupts the toggling of GPIO pins, and stalls the process determining whether another must be run.

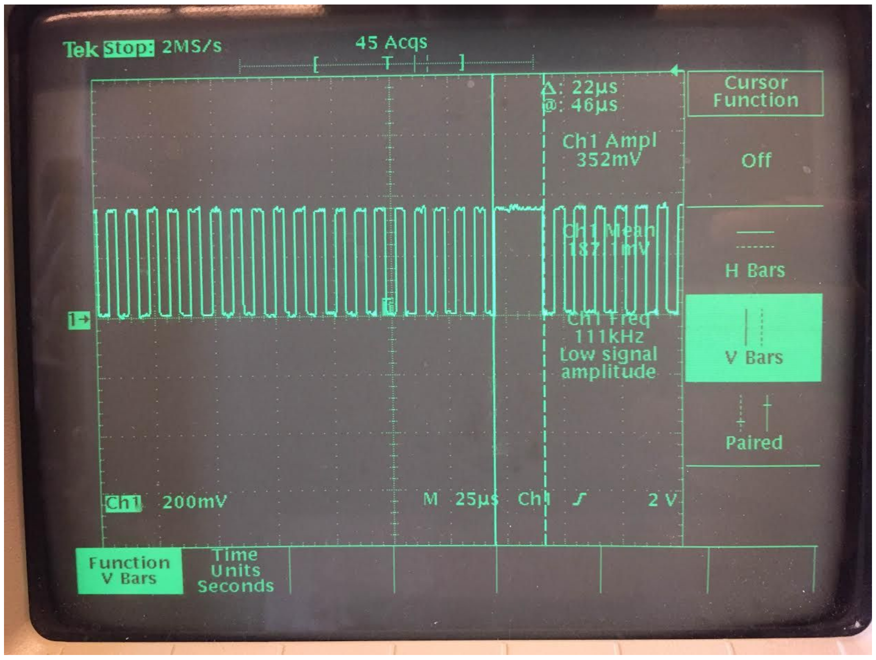

Figure 1: Oscilloscope measurements of scheduler jitter during jitter.c program execution

3.0 Conclusion

This lab was extremely valuable in demonstrating the power of writing abstracted functions to drive hardware processes from a user-interface. While this driver is constantly functioning once executed, the user is given complete control of the system under a specific set of parameters. In this case, the user is given the ability to elect between two different text display functions. But more generally, embedded devices are often expected to operate indefinitely while awaiting user input. We expect these themes to be expanded on in further labs as we continue to develop the hardware extensions to the beaglebone black board and program the hardware-software interfacing between them.