ECSE 426 - Final Project

Harsh Aurora 260394216

Radhika Chandra 260427945

Clark Gredoña 260458872

Loren Lugosch 260404057

Table of Contents

Table of Figures and Table

1.0 Abstract

Our final project for Microprocessor Systems was mechanical control of one microprocessor board using wirelessly transmitted data and commands from another board. Work done in previous labs was reused and modified to fit the requirements of this project. There are two modes of operation: one where the slave board mimics the movement of the master board and one where angles are directly inputted via a keypad connected to the master board to directly control the movement of the slave board. Almost all of the individual components (Master/ Slave behaviour, servo motor movement, SPI, keypad, LCD) worked as intended except for wireless communication, which only started working the day before the demo after laboriously checking our driver for bugs. Once the wireless communication was fixed, we integrated the pieces of our project into a system which met the requirements in the project specification.

2.0 Problem Statement

The ultimate goal of this project was to create a system involving two communicating microprocessors by combining work from previous labs with new work involving writing device drivers. Two modes of operation were asked to be completed for this project. The first mode requires BoardB to follow the physical movements of BoardA as measured by the accelerometer. The second mode requires angle inputs from the keypad connected to BoardA which then directly control the angles at which BoardB must move in and over what timeframe.

3.0 Theoretical Considerations

3.1 Wireless Communication

The CC2500 wireless chipset is a transceiver intended for 2400-2483.5 MHz operation. The CC2500 is highly configurable. Its main operating parameters and its 64-byte transmit/ receive FIFO buffers are controlled via SPI, typically (as in our case) with a microcontroller.

The base frequency for the frequency synthesizer is determined in increments of :

$$f_{carrier} = FREQ[23:0]$$

where FREQ[23:0] is the base frequency for the frequency synthesizer and f_{XOSC}=26 MHz is the frequency of the oscillator crystal [5].

The chipset has several pins relevant to the SPI. CSn is the chip select for the SPI, SCLK takes the clock input from the SPI, SO (GDI1) is an optional data output on the SPI, GDO2 is the general output, and SI is data input [5].

The CSn pin must be kept low for all transfers on the SPI bus. If the CSn pin goes high during a transfer, the transfer will be cancelled. All transfers begin with an address header byte containing a Read/Write bit, a burst access bit, and a six-bit address (config registers are located on SPI addresses 0x00 to 0x2E, register addresses are located on SPI address 0x30 to 0x3D, and the FIFO is accessed on 0x3F). When CSn is pulled low, the Micro Controller Unit must wait for the SO pin to go low before transferring the header byte. When a header byte, data byte, or command strobe is sent on the SPI interface, the status byte is returned on the SO. Command strobes are sent as single byte instructions by sending the burst access bit to 0 [5].

Because data often contains long sequences of zeroes and ones, performance can be improved by whitening the data before transmitting and de-whitening upon receiving. This helps because data transmitted via air is ideally random and DC free [5].

3.2 SPI (taken from our previous report)

The Serial Peripheral Interface (SPI) communication standard allows a master device to send and receive data to and from slave devices. The MEMS accelerometer used in this lab uses SPI to send acceleration readings to the Cortex-M4.

SPI uses four wires: SCLK, MOSI, MISO, and SS. SCLK controls the speed of data transmission and reception (the master and slave transmit and receive on opposite SCLK edges). MOSI sends data out from the master device and into the slave device; MISO sends data out from the slave device and into the master device. SS enables data transfer for a particular slave. The accelerometer is a slave device and the SPI module is the master device, so the accelerometer sends data via the MISO line [1].

3.3 Accelerometers (taken from our previous report)

MEMS accelerometers measure "proper acceleration," the acceleration of a body relative to freefall [2]. This means that an accelerometer at rest on the Earth's surface will return a value with magnitude 1g, and an accelerometer in freefall will return 0. Our accelerometer uses milli-g units, so any reading from the board when not in motion should return a value with magnitude 1000.

Although no other physical data than proper acceleration is necessary for this lab, we were interested in how we could measure acceleration with respect to Earth's surface rather than proper acceleration. "True" acceleration measurement poses an interesting challenge: a program could subtract the acceleration which would be experienced in freefall from the proper accelerometer's readings to get the desired value, but this would require another sensor which could determine orientation with the board in motion (our program assumes that the board is at rest with respect to the Earth's surface) to calculate the correct values to be subtracted from each dimension. One could use a gyroscope for this, but synchronizing the sampling of the gyroscope and the accelerometer could be difficult.

3.4 Servo Motors (taken from our previous report)

Servo motors move an output shaft to a certain position and stop. The position is dictated by the width of a PWM signal. Servos are built on a DC motor and a negative feedback mechanism [3].

The following is an example of an analog implementation of a servo motor. DC motors turn continuously at a rate proportional to the voltage across their input terminals. To convert the PWM signal to something which can control a DC motor, a lowpass filter smooths the PWM pulse to a (roughly) constant voltage [2]. Once the signal is smoothed to an equivalent DC

voltage, that voltage is applied to the input terminals of the DC motor. The DC motor turns the output shaft as well as the knob f a potentiometer. The output voltage of the potentiometer (a voltage divider dividing the power supply of the servo) is added to the DC motor input. This negative feedback eventually stops the output movement: the farther the potentiometer is turned, the more voltage counteracting the control signal input [4].

3.5 RTOS

A real-time operating system is an operating system that trades performance (throughput) for consistent operation. Such systems are classified as hard or soft depending on the time needed to accept and complete a task - hard RTOS's meet deadlines deterministically while soft ones generally meet them. RTOS's also provide support for multi-tasking (focusing on minimal interrupt and thread switching latency), synchronization, and sharing resources.

The CMSIS-RTOS is a common RTOS interface, abstracting away the implementation of the OS functionality and providing programmers an API for multithreading, signalling, and other useful OS functionality. This project uses the Keil RTX RTOS, which features a round-robin pre-emptive priority-based scheduler. This reschedules on events, prioritizing higher-priority tasks and alternating between tasks of equal priority.

4.0 Implementation

4.1 Master and Slave Concept

We considered one STM board to be the master board which takes keypad input or measures its own tilt and roll using its MEMS accelerometer. It transmits this information in a packet to the slave, which in turn controls the servo motors.

To make coding as a team simpler, we used a single C project and made revisions using Git. Which board is the master board and which board is the slave board is determined by a global variable; during testing, one person would make that variable equal "MASTER" and the other "SLAVE".

4.2 Threading

In addition to the main and OS daemon threads, six threads were implemented, but, depending on the whether or not a board was operating in master or slave mode, only three of these were spawned at a time.

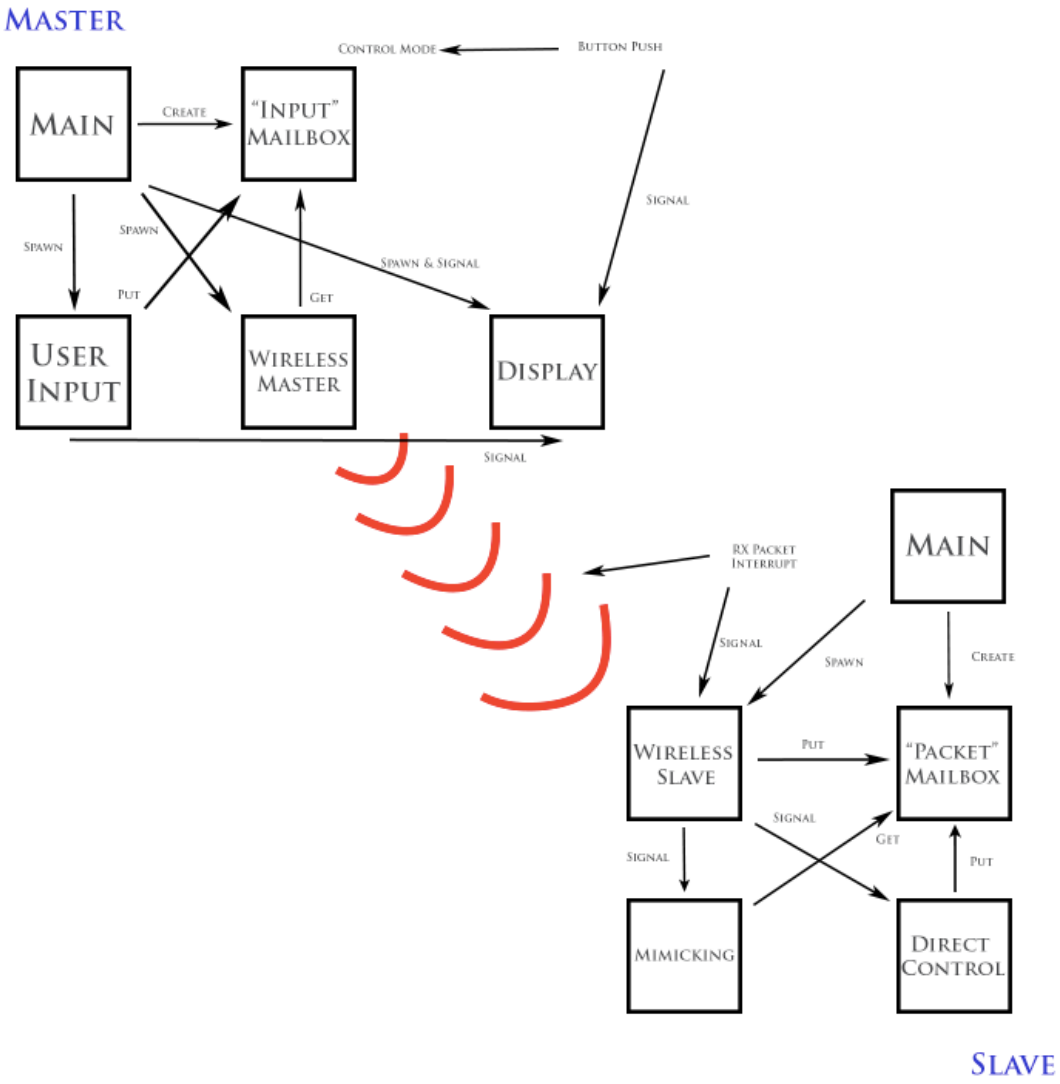

Figure 1: Software Architecture

4.3 LCD Driver

Another design choice was the LCD driver used for the project. Since our group was formed of people from 3 pairs, we had three drivers to choose from, and we had to make whichever driver we picked work with the rest of the project. Loren and Radhika's LCD driver was simple and had only three functions, but Harsh's was more adaptable and interesting: through structs containing information about an LCD driver, his code could allow multiple LCDs, a possibility which we thought was interesting (although we didn't have time to explore it).

We assign 11 GPIO pins to operate the LCD screen - 3 control pins for RS, R/W and EN, and 8 data pins to send character information and configure the settings. In the driver we have two structs as shown in the table below.

Table 1: LCD Driver Structs

Type (struct)	Members	Possible values
LCD_SettingsTypeDef	IncrDecr	INCR, DECR
	DisplayShiftEn	LCD_ON, LCD_OFF
	ShiftSettings	DISPLAY_SHIFT, CURSOR_SHIFT
	ShiftDirection	RIGHT, LEFT
	DataLength	BITS_8, BITS_4
	LineMode	LINES_2, LINES_1
	CharSize	CHAR_5x10, CHAR_5x7
LCD_DisplayTypeDef	DisplayEnable	LCD_ON, LCD_OFF
	CursorUnderline	LCD_ON, LCD_OFF
	CursorBlink	LCD_ON, LCD_OFF
	Address	0x00 - 0xFF

An instance of the LCD_DisplayTypeDef must be assigned desired values and passed of the initialize method to active the screen. After this, an instance of either struct can be passed to the corresponding configure method to configure the desired settings. The driver includes high level methods to clear the display, set the cursor address, write a single character and write a string. The row length of the LCD is defined in a macro in the header file in order to

avoid the displayed information from being stored in the LCD DRAM at an address beyond the physical limit of the screen.

4.4 Keypad Driver

Input from users comes entirely from the blue user button on the discovery board and the keypad. Users switch the system from mimic mode to direct control mode by pushing the blue button and then enter commands through the keypad. The user has to be able to input fairly sophisticated commands using only these two peripherals, so we had to make some clever choices when designing the user input system.

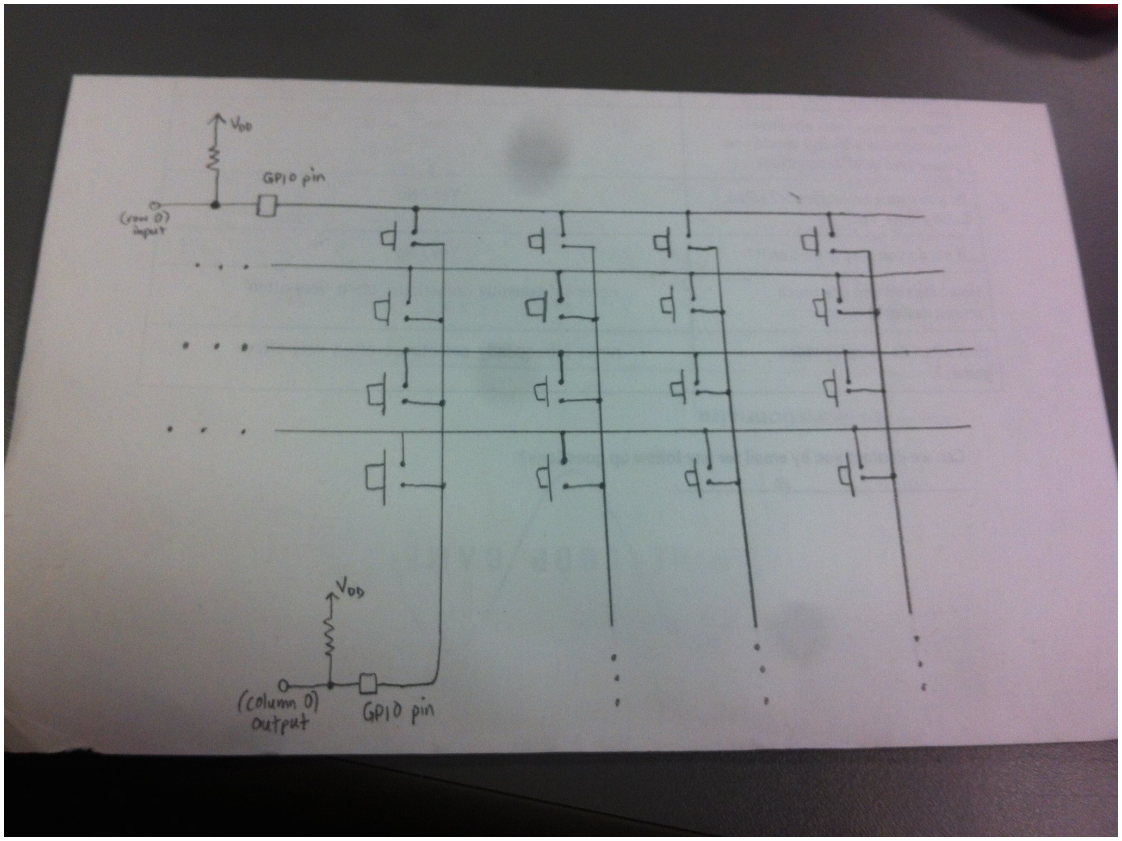

Figure 2: Keypad Buttons

Figure 2 shows how the buttons of a keypad are connected to row wires and column wires. When a key is pushed, the wire corresponding to that key's row is connected to the wire corresponding to that key's column. For instance, when the upper-left button is pressed, the first row wire is connected to the first column wire, and whatever voltage is on the first

column wire will appear at the input GPIO pin connected to the first row wire. If the input GPIO pins are connected to pull-up resistors, they will be “pulled up” to the power supply voltage unless a key is pressed. By configuring the column pins to output zero volts and configuring the row pins as inputs, we can detect when one of the keys in a row has been pressed by listening to the row pin’s voltage (either by polling or interrupts); if it isn’t at the pull-up voltage, a key in that row is being pressed. We decode the key press by switching the outputs to inputs and the inputs to outputs and repeating the above.

Our primary design choice in writing our keypad driver was the method of listening to the input pins on the keypad. We chose to use regular polling rather than another method like “smart polling” or interrupts because of our limited resources. Regular polling is inefficient because the processor does work even when it is unlikely that the peripheral will need to communicate The user’s finger is physically unlikely to press another key a few milliseconds after a key press has been decoded: a smarter method of polling than regular polling would be to “back off” after a press and poll more slowly, then ramp up the polling speed.

Interrupts are preferable to polling, but the keypad would require four interrupt lines, and the EXTI module on our microprocessor board only has a few EXTI lines (and once a GPIO pin with a certain number is used, no other GPIO pin with that number can be used). To use interrupts without having a separate line for each input source on the keypad, we could connect each of the row inputs to a 4-input OR-gate and configure the output of the OR-gate to generate interrupts. Once an interrupt is generated, the MCU would check the rows once and the columns once. We considered this option when we were thinking of using interrupts, but given how compact and unburdened our system is, we decided that it was not too taxing on the processor (and easier to debug) to use polling instead.

4.5 Direct Input State Machine

When operating in sequence input mode, the keypad thread operates within a finite state machine, which is shown in a simplified way below. Beginning in “wait for input” mode, it takes 10 numerical inputs. The first three are stored as pitch, the second three are stored as pitch, and the final four as stored as time (in milliseconds). After 10 inputs, the thread goes to the “Wait for Send Command” thread. This state transitions to the “Sending” state after a “D” press, which was arbitrarily chosen. The sending state resets to the wait for input state after waiting after half a second. An invalid keypress (i.e. a letter in the “Wait for Input”

stage or anything but “D” in the “Wait for Send Command” state) or invalid values of pitch and roll (i.e. angles larger than 180 degrees) forces a transition to an “Input Error” state for half a second.

Not pictured is a “Wait for Button Release” state after each button press. This state ensures that only one character is inputted at a time. As long as the user’s finger is still pushing the button, the keypad won’t return “NO KEY PRESSED”. Once “NO KEY PRESSED” is returned, the program knows that the button isn’t being pressed anymore. This also allows users to hold down the keypad button as long as they want and only input one character. (We could also have harnessed this behaviour to implement the “hold down to repeat character” feature that many PC keyboards have, but this isn’t that useful for a terminal that accepts 3 numbers representing a very specific movement.)

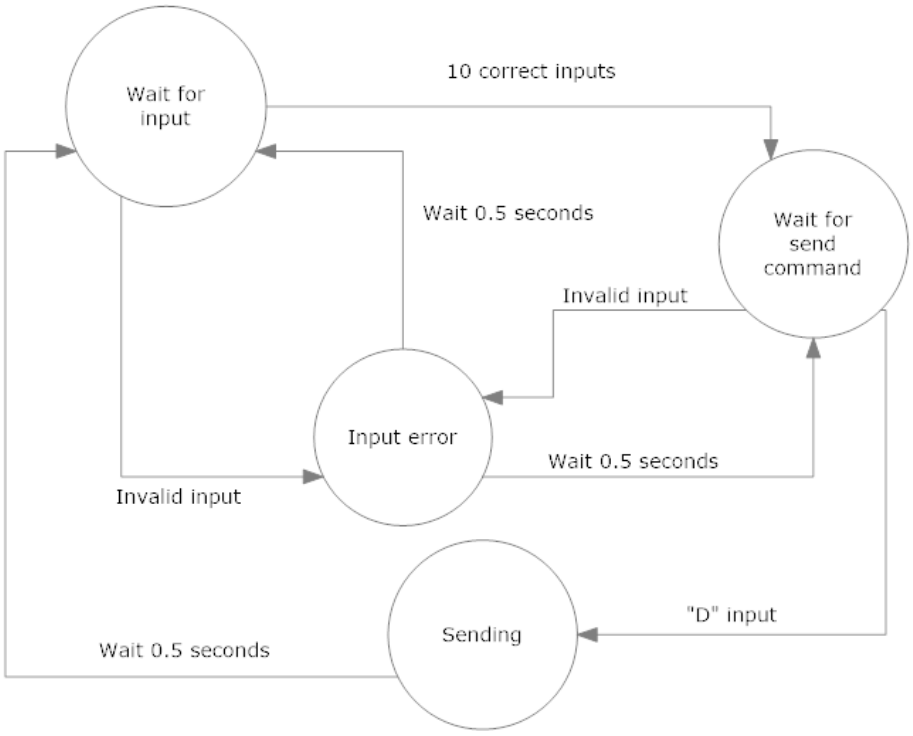

Figure 3: Finite State Machine

4.6 Wireless Driver

The most challenging and interesting part of this lab was writing the wireless driver. The MCU uses SPI to communicate with the chipset in a way very similar to the way it communicates with the MEMS accelerometer, so we adapted much of our code from the

MEMS driver the TAs provided with a few changes (for instance, adding the burst command bit to the right place when reading or writing multiple bytes). Initially, we were having difficulty writing bytes to the SPI's shift register, and it took the use of an oscilloscope to realize that the ARM library function for changing the slave-select line didn't work, so we had to reconfigure the slave-select as a normal GPIO output rather than as an alternate function.

After setting up SPI communication between the MCU and the chipset, we use the SPI reading and writing functions to write configuration codes to the configuration registers. Most of the default values from the project specification we kept, but we changed (among other things):
-the carrier frequency (the TAs recommended that we set our carrier frequency to 16 KHz away from the default, but we added a full MHz)
-the number of preamble bytes (we used the maximum number to reduce the number of "false positive" packets)
-the use of the address byte (a constant other than broadcast)
-the packet length (1 address byte + 25 packet bytes = 26 bytes) + fixed packet length
These changes were primarily steps along the laborious way of debugging; it took a full two weeks to get a single byte transmitted after trying different configurations.

Once the configuration registers are set, the master thread receives input from the user or from the board orientation, wraps the data in a packet struct, duplicates the packet fields, writes the bytes to the TX FIFO buffer, and sends the TX strobe. The chipset transmits the packet once it gets the strobe, and the slave chipset receives the packet and generates an interrupt. The interrupt triggers a handler in the slave board which tells the slave thread to take the data from the RX FIFO buffer. The slave thread does this, demultiplexes the packet, and gives it to the appropriate thread by putting the data in the packet mailbox and signaling the thread indicated by the "mode" field of the packet.

4.7 Servo Motors

As with Labs 3 and 4, the servo motors were controlled via hardware-implemented pulse width modulation. Upon receiving new angles to rotate to, the slave calculated the appropriate duty cycles to set the motors to. In sequence input mode, the change in duty cycle was interpolated over time increments of 20 milliseconds to ensure smooth movement.

4.8 Mechanical Setup

The mechanical design of our system was, in the end, quite simple. We started off thinking of a design that has the receiving board in suspension attached to strings that are attached to the servo motors which wind and unwind to move the board in wanted angles. This design however seemed to be quite complicated since we would have needed four strings for two servo motors. Instead, we came up with a simple design that takes the two motors and attaches them perpendicularly - one rotating the board to the pitch angle and the other the angle of roll. The receiving board was then rested on a styrofoam piece.

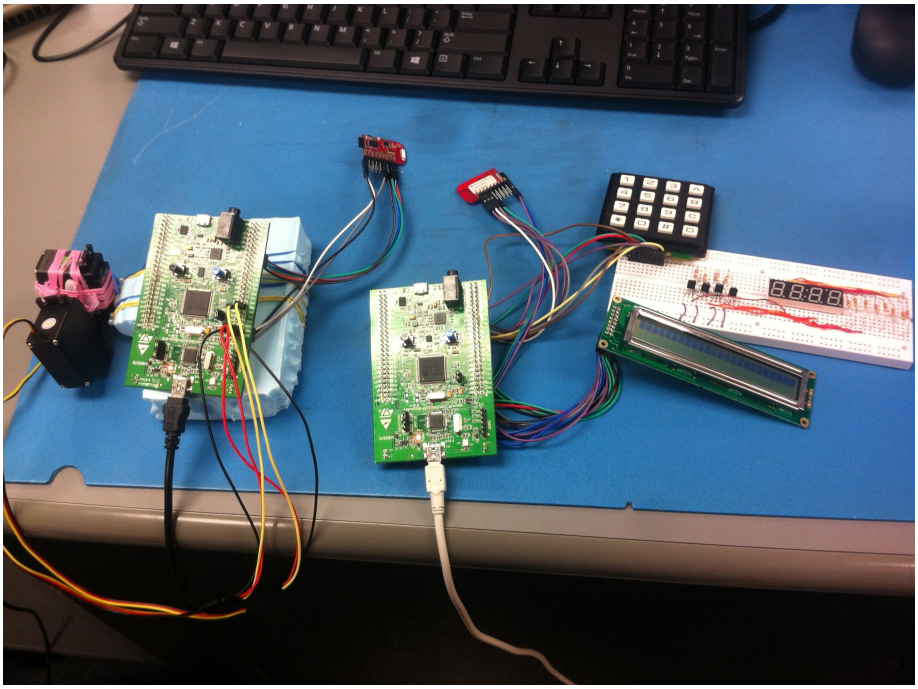

Figure 4: Mechanical Setup for the boards

5.0 Overall Design

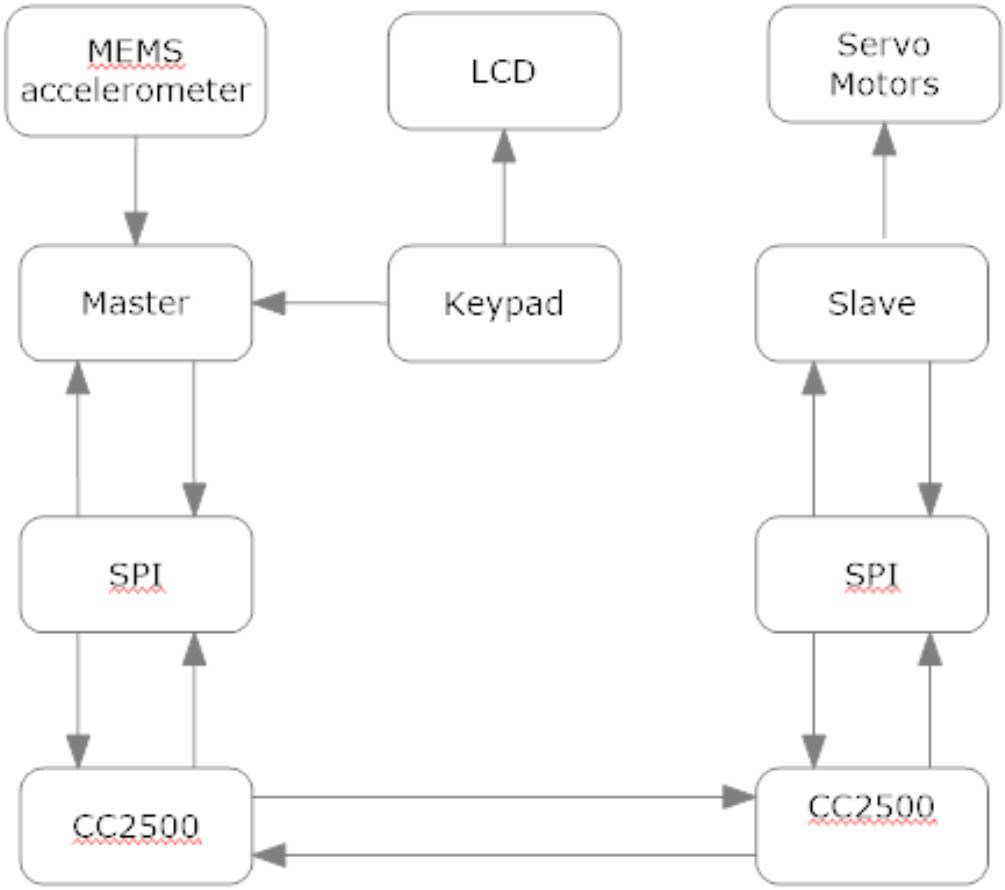

Figure 5: High Level System Architecture

A timeline of work and a breakdown between team members is below:

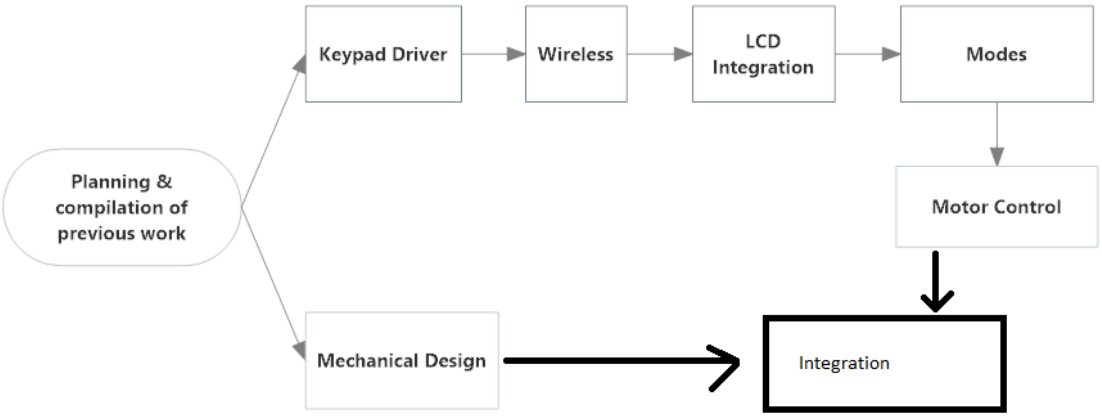

Figure 6: Timeline of workflow

Harsh

- Wireless Connection & Driver
- LCD Driver

Loren

- User Input Handling
- Wireless Connection
- Keypad Driver

Clark

- Documentation
- Servo Motor Setup
- Mechanical Design

Radhika

- Documentation
- Servo Motor Setup
- Mechanical Design

6.0 Testing and Observations:

6.1 Wireless Connectivity

When we first got wireless communication to work, we noticed that it only worked if the boards were very close together. We fixed this when we added some of the packet error checks, but there was still a bit of noise when we moved the boards apart, so we performed simple range tests on the chipsets by increasing the width and sending (as accurately as we could) the same data.

We did the range test for both mimicking and direct control mode. In mimicking mode, we made the same gentle back-and-forth movement with the master board repeatedly and slowly moved the slave board farther away from the master board. As we did, the movements became slightly more jerky (even though the person holding the master board was careful to maintain the same motion). We think that this is because more packets were failing CRC and being dropped the farther away our board moved. In direct control mode, the likelihood that a command would be missed also increased.

7.0 Challenges Encountered

7.1 Communication Between Threads

Designing the communication between threads in a robust way took some thought. In the end, we used a combination of signals, mutexes, and FIFO mailboxes. For instance, both the direct control and mimicry thread used a mailbox filled by the slave thread upon receiving a wireless byte. But only one of these threads was woken up via signal, depending on which mode of operation the slave worked in. The master board's input display thread modified character buffers that were protected by mutexes. After this was done, it signaled the LCD thread to update the display. The inter-thread communication also proved challenging in part because we had no experience using mailboxes before and had trouble getting them to work.

7.2 Finite state machine

As detailed earlier, the finite state machine in the input thread was moderately complex. The sequential input mode of operation required several states, and this was made more complex because we had to ensure that each keypress only inputted one character at a time. To resolve this, each keypress transitioned to a "Wait for Release" state.

We also ran into a few bugs where the state machine would incorrectly "hang" on a certain states rather than transition to the next state. When this happened, it was not obvious whether this was a problem with the state machine, the display thread, or the communication between the two.

7.3 Switching between modes seamlessly

The wireless master can operate in two modes: DIRECT_MODE and MIMIC_MODE. Switching between modes is toggled by a button press which triggers an interrupt. Pressing the button must toggle the mode of operation in the wireless master thread, the input thread, and the LCD thread, and switching between modes seamlessly is a challenge because timing is an issue. In the while loop of the wireless master thread, the mode is first checked before branching into waiting for different signals and mail. If mail from the keypad input thread, for instance, never reaches the wireless master thread because the keypad no longer takes inputs because it is in mimic mode, the wireless master thread can become stuck waiting for the mail that never comes. This was resolved by making sure the master thread only waits for mail for a short period of time before continuing the loop. It was important to find a good length of time that ensured the wireless master did not get stuck and also responded in a timely manner to input from the input thread.

7.4 Wireless

Wireless transmission proved the greatest challenge and only began working the afternoon before the demo. The receiver (slave) initially continuously polled its CC2500. The CC2500 picked up noise, so as a result, the receiver polled a lot of garbage data. This was fixed by replacing polling with an interrupt-driven system where the receiver reads RX after it

receives an external interrupt from the GDO pin, signaling that a complete packet has been received. Sometimes “false positive” packets arrived, so we added CRC checking and preamble bytes.

Occasionally, “real” packets would still have a bit of noise undetected by CRC, so we sent 5 copies of each member of the packet struct and checked them against each other at the receiver end to reduce the likelihood of bit flipping. Bit flipping was especially disastrous if the control field of the packet was changed; the 5-byte method eliminated this problem. In industry, this kind of redundancy would probably not be acceptable, especially in high-bandwidth communication. We would explore better fault tolerance if we had had more time to work with the wireless.

8.0 Conclusion

We met all of the project specifications and learned an enormous amount about microprocessor-based design in the process. We had a lot of trouble getting the wireless communication reliable and functioning, but we were almost happier that we had trouble with it rather than having it work immediately. If the wireless chipsets had worked right from the start, we wouldn’t have learned as much and as deeply about how they work and what they’re capable of as we did during the hours we spent reading through the datasheets and consulting with other project groups.

9.0 References

[1] Accelerometer Application Note, p. 13

[2] Sethuramalingam, T. K.; Vimalajuliet, A., "Design of MEMS based capacitive accelerometer," *Mechanical and Electrical Technology (ICMET), 2010 2nd International Conference on* , vol., no., pp.565,568, 10-12 Sept. 2010

[3] http://www.sciencebuddies.org/science-fair-projects/project_ideas/Robotics_ServoMotors.shtml

[4] http://www.youtube.com/watch?v=tsrAP8EgcbQ

[5] CC2500 datasheet