

ECE 150: Fundamentals of Programming

Embedded Systems Self-Directed Project Report

Remote Access Locking System for Student Dorms

Group # 24

Group Members

Kyle Pinto
Efaz Shikder
Hemit Shah

Project Overview and Objectives

The premise of this project was to create a locking system that would bring the functionality of popular commercial smart home systems to a student living in a dorm. All students living in dorms are not allowed to modify their electrical systems, or modify the furniture, walls or locks under their contracts.

As such, the aim of this embedded systems project was to create a comprehensive smart lock system that would be remotely accessible over a wireless network with a remote-like key. The system aimed to be implemented on the existing deadbolt-based locking mechanisms on student dorm doors. Additionally, the system was to be small, self-contained and relatively cheap compared to other solutions.

Overall the system would provide users with several locking features to keep students from forgetting to lock their doors, or remotely unlock their doors to let trusted individuals into their rooms.

Fundamentally, the smart lock system would give the user remote access to the locking mechanism on their door and lock or unlock the door as they please using the remote key. The user would also be given the option of setting the lock system to an auto-lock state which would lock the door within a few seconds when the door is closed.

Additionally the user would have the ability to set predefined auto-lock times at which, if closed, the door would automatically lock on a daily basis. Auto-unlock times could also be defined at which, if locked, the door would automatically unlock on a daily basis. (Not currently implemented, but possible through the use of the configuration file)

The system would also provide the user reliable and consistent data on whether the door is currently locked or unlocked. This feedback would be given in both a plain and clear method, a red or green light letting the user know whether their door is locked or unlocked, and a detailed manner, a file on which the user can see at exactly what time they sent commands to lock or unlock their door and whether those commands were carried out.

As the system controls a critical part of any user's life, failures would reliably be addressed through the use of other mechanisms that would immediately boot the locking service upon system startup and also immediately restart the service if the system has to reboot for any reason. The system would keep track of its lock state before the reboot and immediately reset to that state (locked or unlocked) upon restarting.

This affordable and contract-friendly smart dorm lock would help students transition into the modern era of living.

High - Level System Design

Lock and Key System

The locking system designed and implemented in this project consists of two major hardware components or “nodes.” One node was designed to be the lock mechanism and the other to be the remote access key.

Hardware

- (2) raspberry Pi's
- (1) laser
- (1) photodiode
- (1) servo motor (including popsicle stick structure for mounting)
- (2) red LEDs
- (2) green LEDs
- (2) pushbuttons
- (1) 9V to 5V converter (LM7805)
- (External power outlet adapter)
- (4) 220R resistors
- (1) 1K resistor
- (1) 10K resistor

Lock Mechanism Hardware

The lock mechanism consists of the hardware controlling the lock and providing the user local feedback on the current lock state of the door. It is comprised of a servo motor with a physical structure that allows mounting onto the lock and door, LEDs for user feedback, and a photodiode sensor coupled with a laser that allows the mechanism to determine whether the door is closed. All of these hardware components are connected through a Raspberry Pi W Zero and breadboard dedicated solely to controlling the lock mechanism on the door.

The servo motor is controlled through one GPIO Pin on the Raspberry Pi, however it is powered externally from a wall adapter with a voltage output of 9V. Since the servo requires a 5V power supply, an LM7805 voltage regulator chip was used to step down the voltage to 5V on the breadboard. Using an external power supply is also advantageous as it reduces the risk of damaging the pi by connecting the servo motor directly to a pin. The LEDs on the lock mechanism breadboard are controlled by the Pi through the GPIO Pins. The green LED illuminates when the door is unlocked, and likewise, the red LED indicates that the door is locked. Finally, the photodiode is also connected to a GPIO Pin on the Pi. Together with a laser diode, the photodiode is used to determine if the door is fully closed before attempting to change the state of the door lock. Finally, a single pushbutton is included to allow for the lock state to be changed directly from the door. Pushing the button once toggles the state of the lock between locked and unlocked. All of these components are mounted directly onto the inside of the door that is being controlled by the system. The laser that hits the photodiode is mounted off the door for stability such that when the door closes it is aligned with the photodiode on the breadboard mounted to the door. This is also done for safety reasons, so that the laser does not stray about the room when the door is opened. A depiction of the lock hardware setup can be seen below.

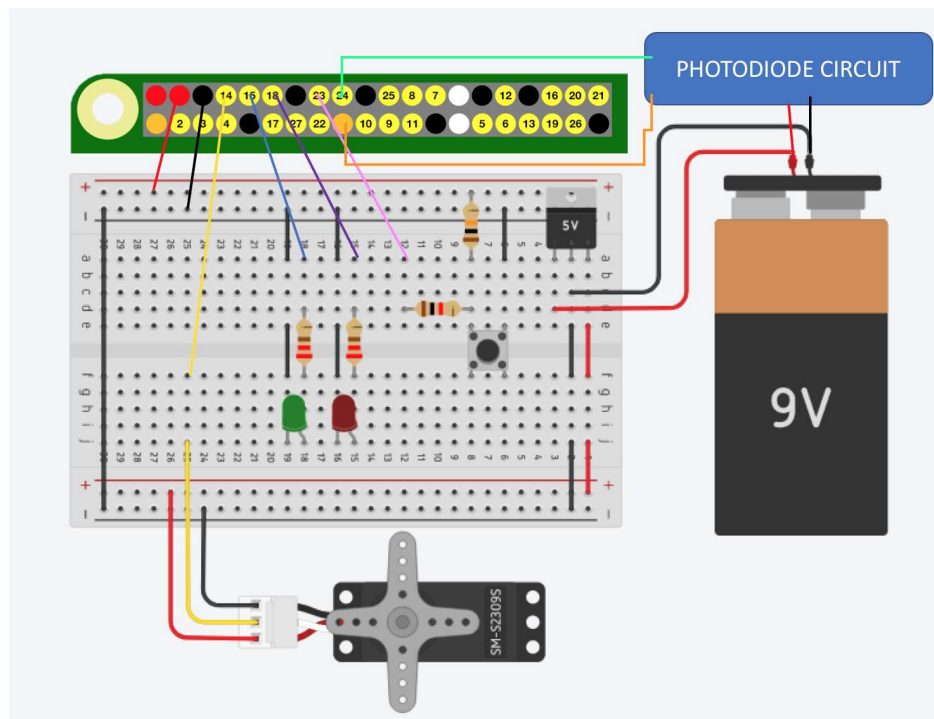


Figure 1: Diagram of the physical circuit implemented on the breadboard and its connections to the Raspberry Pi for the dedicated lock mechanism. The photodiode circuit labelled in the diagram represents an implementation of the circuit from Embedded Lab 3 on the same breadboard, with input from the photodiode going to pin 24.

Remote Access Key Hardware

The remote access key consists of a set of green and red LEDs providing the user feedback on the state of the door (green LED for unlocked and red LED for locked), and a single push button that acts as both the main lock and unlock command point both built in circuits on a separate breadboard. Both hardware components are connected to and controlled by a dedicated “key” Raspberry Pi. Both the LEDs and the pushbutton are connected to GPIO pins on the Raspberry Pi.

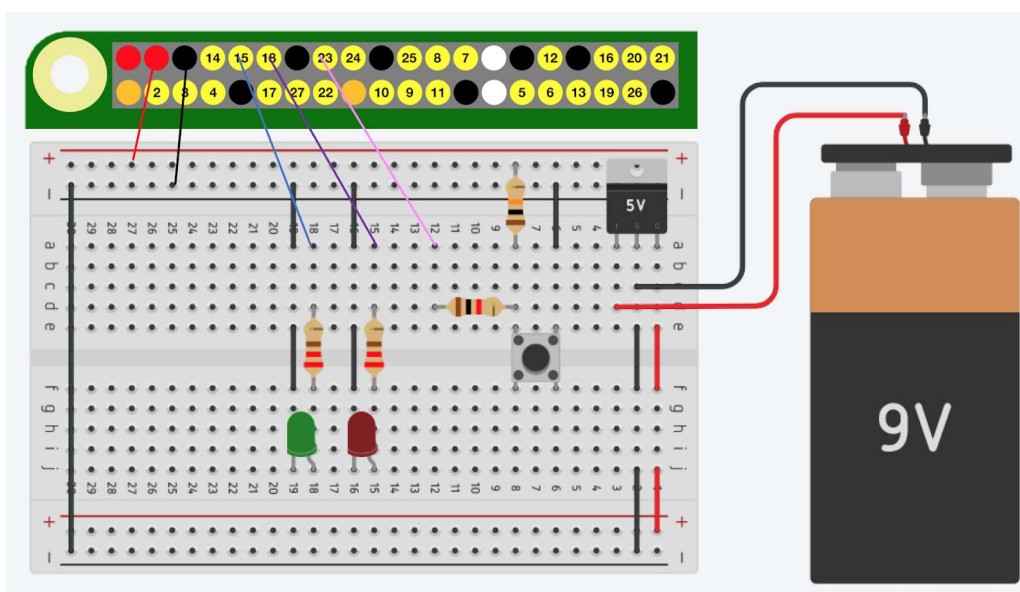


Figure 2: Diagram of the physical circuit implemented on the breadboard and its connections to the Raspberry Pi for the dedicated remote access key system.

Software

Both the software on the “lock” Pi and the “key” Pi implement the following

- Initialization of all hardware components
- Writing to log files
- Unit file to run service on startup
- *Reading from the configuration file* (*Functional, but not shown in demo)
- *Watchdog timer* (*Not shown in demo)

Communication Protocol

The “lock” Pi and “key” Pi communicate primarily through a plain text file (`commFile.txt`) stored in a shared network folder on a windows PC that both Pi’s have mounted to their home directories. Both Pi’s can read and write to `commFile.txt`, although the only information stored in the file is a ‘1’ or a ‘0’ specifying whether the door should be in a locked (‘1’) or unlocked state (‘0’). The use of the pushbutton on either the lock circuit or the key circuit writes the applicable command (lock if currently unlocked and vice versa) to the communication file.

High Level Software Overview

The software infrastructure on the Raspberry Pi controlling the lock mechanism consists of a simple state machine that implements PIGPIO library functions to control the servo motor and a function to read the communication file stored in the shared network folder. GPIO library functions are also used to control the LED and button circuits. Reduced to a few lines, the program running on the lock mechanism reads the current command (a ‘1’ or ‘0’ specifying locked or unlocked respectively) in `commFile.txt` and executes the command through a state machine that consists of four major states: `START`, `UNLOCKED`, `WAITING_TO_LOCK` (waiting for the door to close as detected by the photodiode) and `LOCKED`.

A separate piece of software is used to operate the “key” of the system, consisting of a second Raspberry Pi. This code includes functions to read and write the state of the lock to the `comm.txt` file by means of a simple state machine. It also uses GPIO functions to read the user input via a button and output the current state of the lock to the LED indicators. There are only three states in the state machine: `START`, `PRESSED` and `UNPRESSED`. When the state transitions from `PRESSED` to `UNPRESSED`, if the communication file contains the previous command a command (‘0’) is written to the communication file to unlock the door, and if the door is currently unlocked a command (‘1’) is written to the communication file to lock the door.

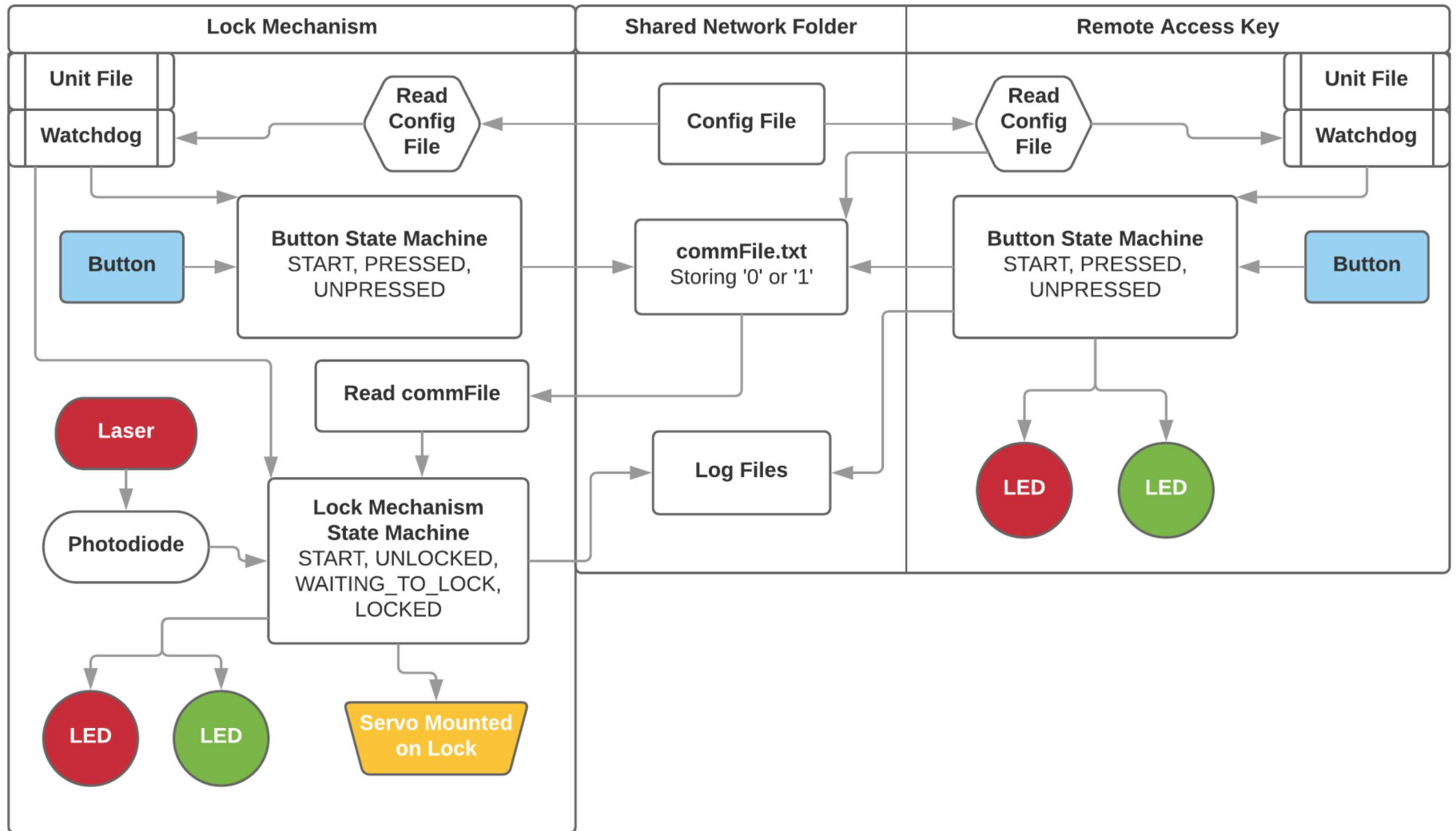


Figure 3: Diagram showing the interaction between main hardware and software components in each node (lock mechanism and remote access key) including interactions with the files shared on the shared network folder. Important state machines and file reading functions are shown rather than all of the code for simplicity. The logical pathways between files, functions and hardware components are shown using the directions of the arrows, but should not be taken to represent the complete interaction between all elements.

Overall File and Code Structure

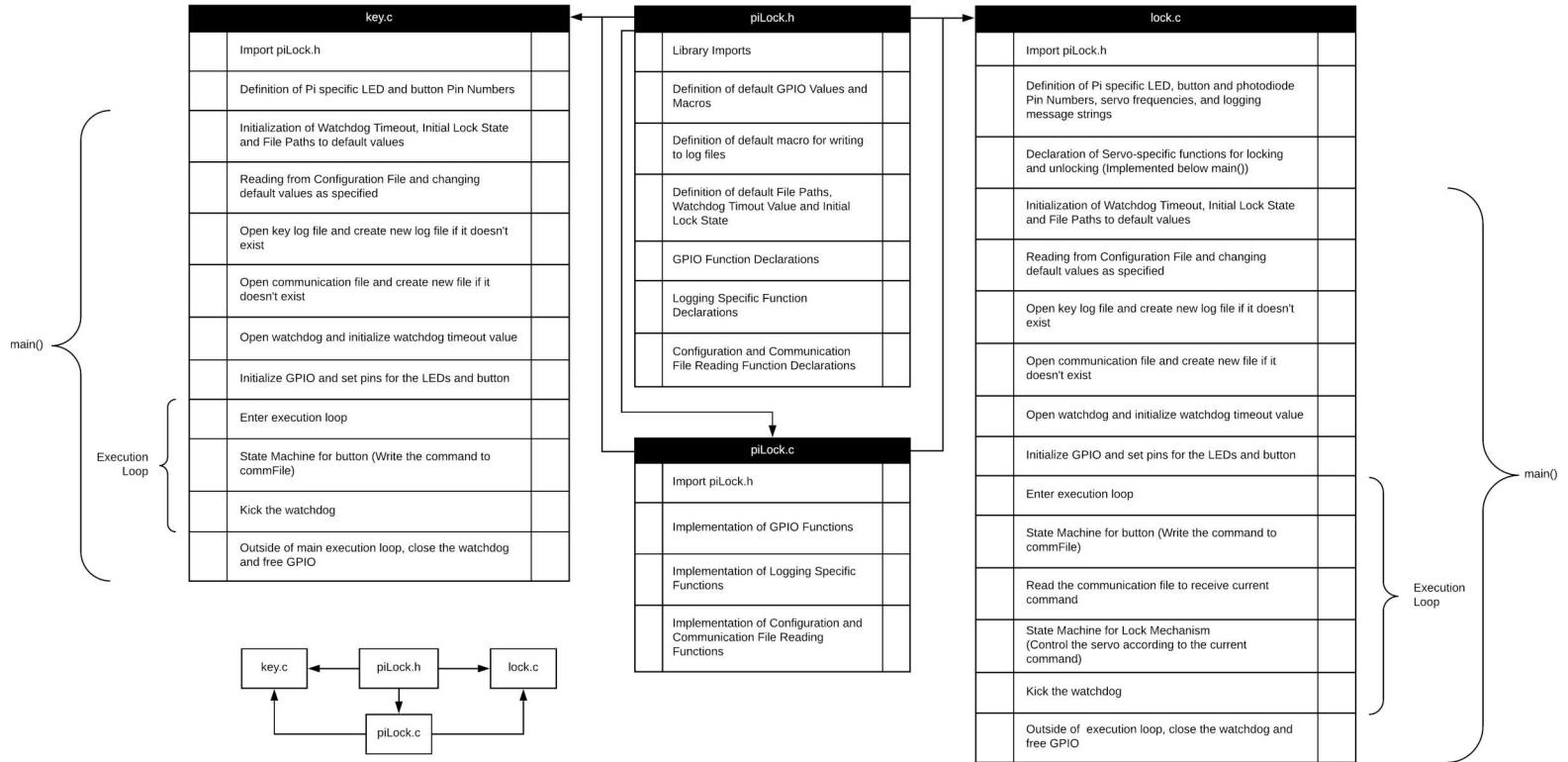


Figure 4: Diagram of overall file and code structure. Simplified file structure on bottom left of the diagram, and internal code structure shown overall in each file. Logging was done at pertinent points throughout `key.c` and `lock.c`. Both files were compiled separately on their respective Pi's dedicated to either the lock mechanism or the remote access key using the following commands respectively:

```
gcc -o key key.c piLock.c
gcc -Wall -pthread -o lock lock.c piLock.c -lpigpio -lrt
```

Note that the lock file needed to be compiled with additional compiler flags in order to enable the PIGPIO library that was used to control the servo. The lock executable also needed to be run using the `sudo` command in order for PIGPIO to function properly. The `sudo` command is also necessary for the watchdog implementation.

Header Files and Function Implementation (piLock.c and piLock.h)

The PiLock header file contains all the required libraries and functions that are used in both the key and lock software. This allows for only one header file to be included in the build during compile time. The implementations of the functions defined in the header file are found in piLock.c. In addition, these files contain definitions of constant values required during runtime, such as the paths to the external data files, timeout constants, servo frequencies, etc.

Remote Access Key Software

State Processing:

The execution loop in main() for key.c includes one state machine that processes input from the push button connected to one of the GPIO Pins. The state machine determines its state by comparing the previous state of the button stored at the end of the previous loop with the current state of the button. After processing the input the state machine writes the applicable command to the communication file if the button has been pressed. The state processing for the remote access key is described below.

- **START:** The initial state in which the button pin is read to determine if the button is initially pressed or not pressed. A transition is made to **PRESSED** if the button is pressed and **UNPRESSED** if the button is not pressed.
- **UNPRESSED:** The button pin is read continuously to determine if the button has been pressed. If not then the program remains in the unpresed state. If the readPin() function determines that the button has been pressed it will transition to the **PRESSED** state.
- **PRESSED:** In the pressed state nothing is done unless the button becomes unpresed. This is to prevent the remote access key from sending a command to the lock mechanism if the remote has just been dropped or trapped in a tight space such that the button remains pressed. Once the button is released, the previous command in the communication file is read. If the previous command was a '1' (to lock) then a '0' (to unlock) will be written to the communication file. If the previous command was a '0' (to unlock) then a '1' will be written to the communication file. Then a transition is made back to the **UNPRESSED** state.

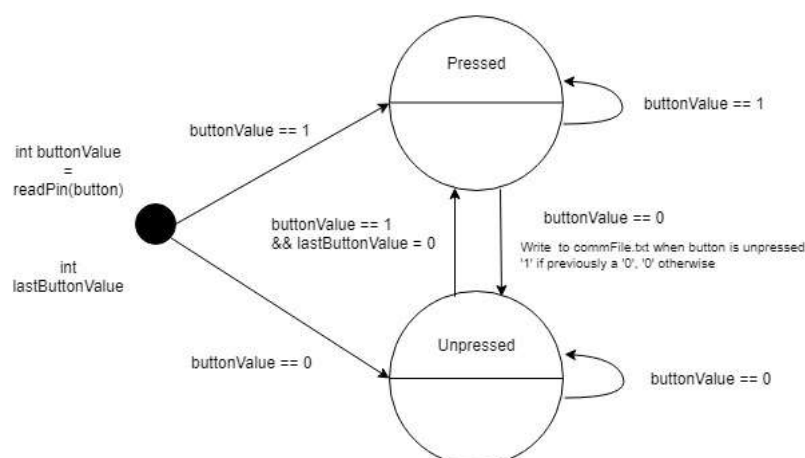


Figure 5: State machine diagram for the state machine implemented in key.c and lock.c in order to read the button pin and write the applicable command to the communication file based on the previous command stored in the communication file.

Lock Mechanism Software

State Processing:

The execution loop in `main()` for `lock.c` includes two state machines. One for the button, which has the exact same implementation as the state machine for the button in `key.c`. It is done above the lock mechanism state machine so the command can be written to the communication file before it is process with the `readCommFile()` function used in the lock mechanism state machine. The state processing for the lock mechanism is described below.

- **START:** The initial state in which the first command in the communication file is read and executed.
- **UNLOCKED:** A state in which the door is unlocked (green LED is on) and the program is reading the communication file until it receives a command to lock the door ('1') at which point it will transition to the state **WAITING_TO_LOCK**.
- **WAITING_TO_LOCK:** In this state the program checks if the door is closed by checking if the laser is hitting the photodiode and the door is closed. Once it reads that the door is closed through the photodiode, it will wait 1 second before fully locking the door, to prevent unwanted early locking before the door is fully closed. After locking the door it will then transition to the **LOCKED** state.
- **LOCKED:** A state in which the door is locked (red LED is on) and the program is reading the communication file until it receives a command to unlock the door ('0') at which point it will unlock the door and transition to the **UNLOCKED** state.

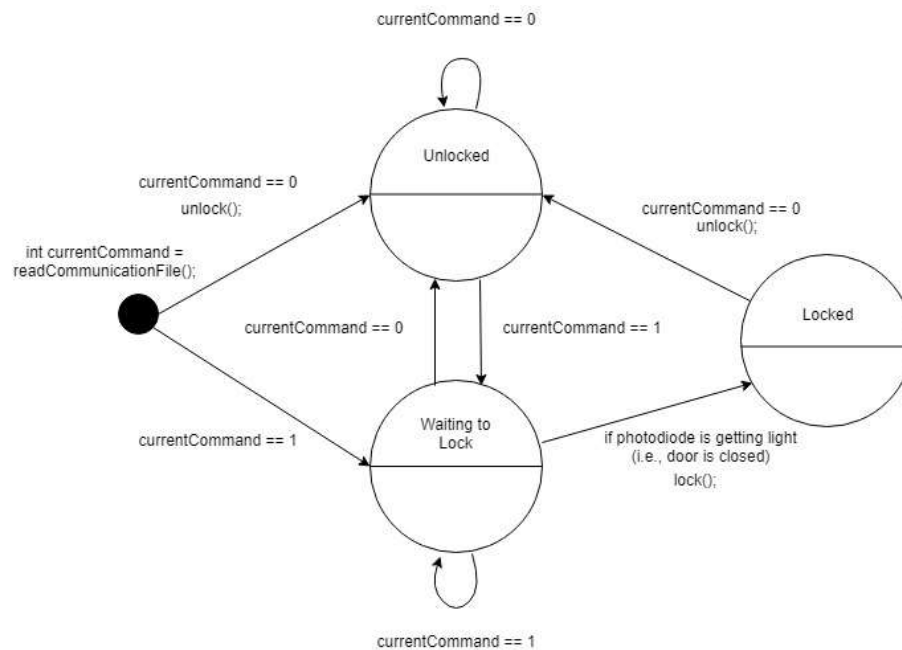
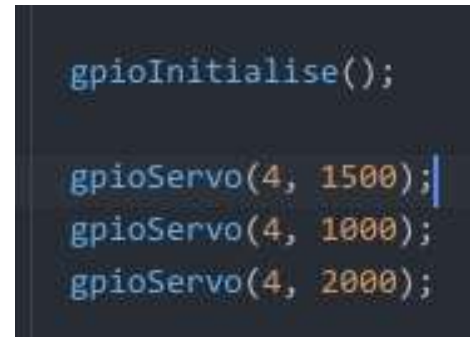


Figure 6: State machine diagram for the state machine implemented in `lock.c` to read the current command in the communication file and execute that command through the servo.

Servo Control:

The servo for the lock was controlled using a third-party library named PIGPIO. This library allowed for proper initialization of the GPIO pins for use with a servo motor. Since a servo motor requires an uninterrupted signal of a particular frequency in order to operate, it can often be difficult to schedule the servo updates within the code. This library was effective in that it managed the scheduling of the servo pulses without the need for additional program cycle management. Pictured is a screenshot of the functions used from the PIGPIO library.

The `gpioInitialize()` function initializes the gpio pins and enables control of the servo. The `gpioServo()` function commands the servo to move to a certain position. The first argument in the function is the GPIO pin number that the servo is connected to. The second argument is the frequency of the servo PWM signal in microseconds.

A screenshot of a code editor showing three lines of C code. The first line is `gpioInitialise();`. The second line is `gpioServo(4, 1500);` with a blue cursor at the end. The third line is `gpioServo(4, 1000);`. The fourth line is `gpioServo(4, 2000);`. The code is syntax-highlighted with blue for keywords and orange for numbers.

```
gpioInitialise();  
  
gpioServo(4, 1500);  
gpioServo(4, 1000);  
gpioServo(4, 2000);
```

System Independence and Dependence

System-Independent Software:

Several elements of the software are independent of the system hardware and are generalizable to any computer system. These modules of code consist mainly of the functions that read and write to the various types of files on the system. Read functions were implemented through the use of state machines and are able to tokenize the data of config files and text files. This allows for existing preferences and system data to be read during runtime. Additionally, functions were also developed to write data to system files, such as the log file, the communication file, and the stats file. This allows for program data and history to be archived on the machine even after the runtime has terminated. Because the file read and write software is not at all dependent on the hardware of the Raspberry Pi, this code can be considered hardware-independent, as file manipulation can be tested and implemented into any computer system that contains a file system.

System-Dependent Software:

The hardware-dependent software consists mainly of the functions that directly interface with the GPIO pins of the Raspberry Pi. This includes the select, read, write, set, and clear functions that are used to manipulate the state of the GPIO pins. The PIGPIO servo functions are also system-dependent as a GPIO interface is required to output a PWM signal to the servo motor. Finally, the watchdog code can also be considered system-dependent. While watchdogs are implemented in many computer systems, the implementation is slightly different among different machines. Thus, the code that operates the watchdog is specific to the Raspberry Pi system and will not operate on other computer systems. Additionally the `getTime(char* time)` function implemented in this software is system dependent as it relies on the internal clock of the Raspberry Pi in both `lock.c` and `key.c`.

Function Descriptions

piLock.c and piLock.h

System Dependent GPIO Functions

The following functions are used in both lock.c and key.c to control the LEDs and read inputs from the photodiode and buttons connected to the GPIO pins on each Pi.

```
GPIO_Handle gpiolib_init_gpio(void);  
void        gpiolib_free_gpio(GPIO_Handle handle);  
void        gpiolib_write_reg(GPIO_Handle handle, uint32_t offst, uint32_t data);  
uint32_t    gpiolib_read_reg (GPIO_Handle handle, uint32_t offst);
```

Description: These are the standard GPIO functions that the higher level functions depend on to interface with the GPIO hardware on the raspberry pi.

```
int selectPin(GPIO_Handle gpio, int pinNumber, int pinType);
```

Description: This function sets the I/O state of a GPIO pin to an input or an output. The function accepts the number of a GPIO pin on the Raspberry Pi and an integer number specifying the I/O state of the pin. The specified pin must exist for the function to be successful. The pinType parameter must either be a 1 or a 0, where 1 indicates that the pin should be configured to an output and 0 indicates an input configuration. The function returns -1 if an error occurs.

```
int setPin(GPIO_Handle gpio, int pinNumber);
```

Description: This function sets the state of a GPIO output pin to HIGH. The function accepts the GPIO pin number as an argument, which must be a valid pin number. The function returns -1 if an error occurs.

```
int clearPin(GPIO_Handle gpio, int pinNumber);
```

Description: This function clears the state of a GPIO output pin to LOW. The function accepts the GPIO pin number as an argument, which must be a valid pin number. The function returns -1 if an error occurs.

```
int writePin(GPIO_Handle gpio, int pinNumber, int state);
```

Description: This function changes the state of a GPIO output pin. The function accepts the GPIO pin number as an argument, which must be a valid pin number. Additionally, it accepts an integer parameter specifying the state of the pin, where a 1 indicates that the pin should be set to HIGH and a 0 indicates that a LOW state is to be set. The function returns -1 if an error occurs.

```
int readPin(GPIO_Handle gpio, int pinNumber);
```

Description: This function reads the state of a GPIO input pin and returns the result. The function accepts the GPIO pin number as an argument, which must be a valid pin number. It will then read the value of that pin and return the digital state as a int. A return value of 0 indicates that the pin is LOW and a return of 1 indicates that the pin is HIGH. This allows the function to be used in a boolean context. The function returns -1 if an error occurs.

```
int freeGPIO(GPIO_Handle gpio);
```

Description: This function simply shuts down the GPIO by invoking `gpioLib_init_gpio()` to deallocate the memory associated with the GPIO pins. The pins will be left in whatever state they were last in.

System Dependent `getTime()` Function

```
void getTime(char* buffer);
```

Description: This function takes in a buffer character array and returns the current date in a yyyy-mm-dd format as well as the time of day 24 hour notation. It makes use of `<time.h>` and `<sys/time.h>` for `time_t` and `time()` as well as the `getTimeOfDay()` functions. The purpose of this function is mainly for logging in the current implementation of the software, as it allows log messages to be printed to the log files with the time at which certain parts of the code are executed properly or if an error occurs.

System Independent File Writing and File Reading Functions

```
int findLength(const char* fileName);
```

Description: This function takes a character array (string) as an input and returns the length of the string (number of characters in the string). It is used to determine the length of strings that need to be copied to new strings as the declaration of a new string requires a parameter specifying the length of the string. In the current implementation it is only used to find the length of `argv[0]` or the program name in order to copy the program name to a new string without the `“./”` in front through the `copyProgramName()` function.

```
void copyProgramName(char* programName, const char* fileName);
```

Description: This function takes in two character arrays and copies the second character array `fileName` to the first, `programName`, with the first two characters of the argument removed. This is done through a simple for loop. This function is used to strip `argv[0]` or the filename of the `“./”` that exists at the beginning character array, for the string `programName` to be used later for logging purposes. The function has no return value as it modifies a given string.

```
int strCompare(const char* compare, const char* source);
```

Description: This function takes in two character arrays and checks if they are the same. It returns a 1 if they are the same, 0 otherwise. This function is used when reading the config file to check if the parameter name written in the file is one of the parameters that we want to check for, so that the value declared after the `‘=’` can be assigned to the correct parameter by transitioning to the correct state within the `readConfig` state machine. For example, we would check if the config file says `“logfile:”`, before we take in the value after it to get the logfile path.

```
void strCopy(char* dest, const char* source);
```

Description: This function takes two character arrays as inputs and copies the second argument, `source`, to the first argument, `dest`, by means of a simple for loop. The function has no return as it modifies a given string. The function is used in `lock.c` and `key.c` when initializing the file path character arrays declared in `main()` to their default values by copying the default file paths (as character arrays) declared in the header file rather than simply pointing to them.

```
void readConfig(FILE* config, int* timeout, int* lockState,  
                char* commFilePath, char* lockLogFilePath,  
                char* keyLogFilePath);
```

Description: This function takes in the pointer for the config file and uses it to find the values to define parameters that are given to it. Based on the information in the config file, it defines the number of seconds before the watchdog should time out, the default initial lock state, the file paths to the key specific log file, the lock specific log file as well as the log file. This function is used to make our system configurable such that the user can specify the default lockState the lock should be in upon startup. It also allows the user to specify the location of the communication file, which may not be the same depending on the shared network folder settings for each user. Although all of the values are initialized to their default values declared in the header file piLock.h, this function is then called to make any changes as specified in the configuration file. The function has no return as it only modifies the given parameters.

```
int readCommunicationFile(char *commFilePath);
```

Description: This function takes a character array or string specifying the file path to the communication file as an input. It opens the communication file (default or configuration-defined) and returns the first character it reads. This should be a 0 or a 1 depending on the command written to the communication file by the remote access key or manual button located within the lock mechanism. This function is used to determine the current command for the state of the door (locked or unlocked) based on the input it reads.

lock.c

System Dependent GPIO Servo Functions

```
void lock(GPIO_Handle gpio);
```

Description: This function uses a function from the PIGPIO library to command the servo to turn the deadbolt clockwise to lock the door based on the frequency defined above `main()` in `lock.c` file. It also updates the LED indicators to reflect the locked state, i.e., red LED is turned on while the green LED is turned off.

```
void unlock(GPIO_Handle gpio);
```

Description: This function uses a function from the PIGPIO library to command the servo to turn the deadbolt counter-clockwise to unlock the door based on the frequency defined above `main()` in the `lock.c` file. It updates the LED indicators to reflect the unlocked state, i.e., green LED is turned on while the red LED is turned off.

```
void cleanup(GPIO_Handle gpio);
```

Description: The `freeGPIO()` function unmaps the memory without regard for the state of the GPIO pins. This function resets all the GPIO pins to a safe state by setting all the outputs to LOW. This function should be called right before `freeGPIO()` or whenever the program is restarting.

Separation of System Dependent and Independent Code

Although complete separation of hardware dependent and independent code was not achieved in the file structure of the software, the functions themselves are very well separated. The system independent file reading and file writing functions do not depend on any of the hardware specific functions. An attempt was not made to create an overarching “writeToLogFile” function as this would require the use of the `getTime()` function, which in our implementation is dependent on the Raspberry Pi, a linux based system. Similarly other attempts were not made to create functions in which hardware dependent and hardware independent would need to be used simultaneously as this would complicate testing.

Within `key.c` and `lock.c`, only the code required for logging and the code used to initialize the pins and read from or write to them within the state machines are system dependent. Otherwise reading from the config file and communication file or writing to the communication file could all be done on any system. In fact, all of the software independent functions were also tested locally on a windows machine.

Function Call Tree

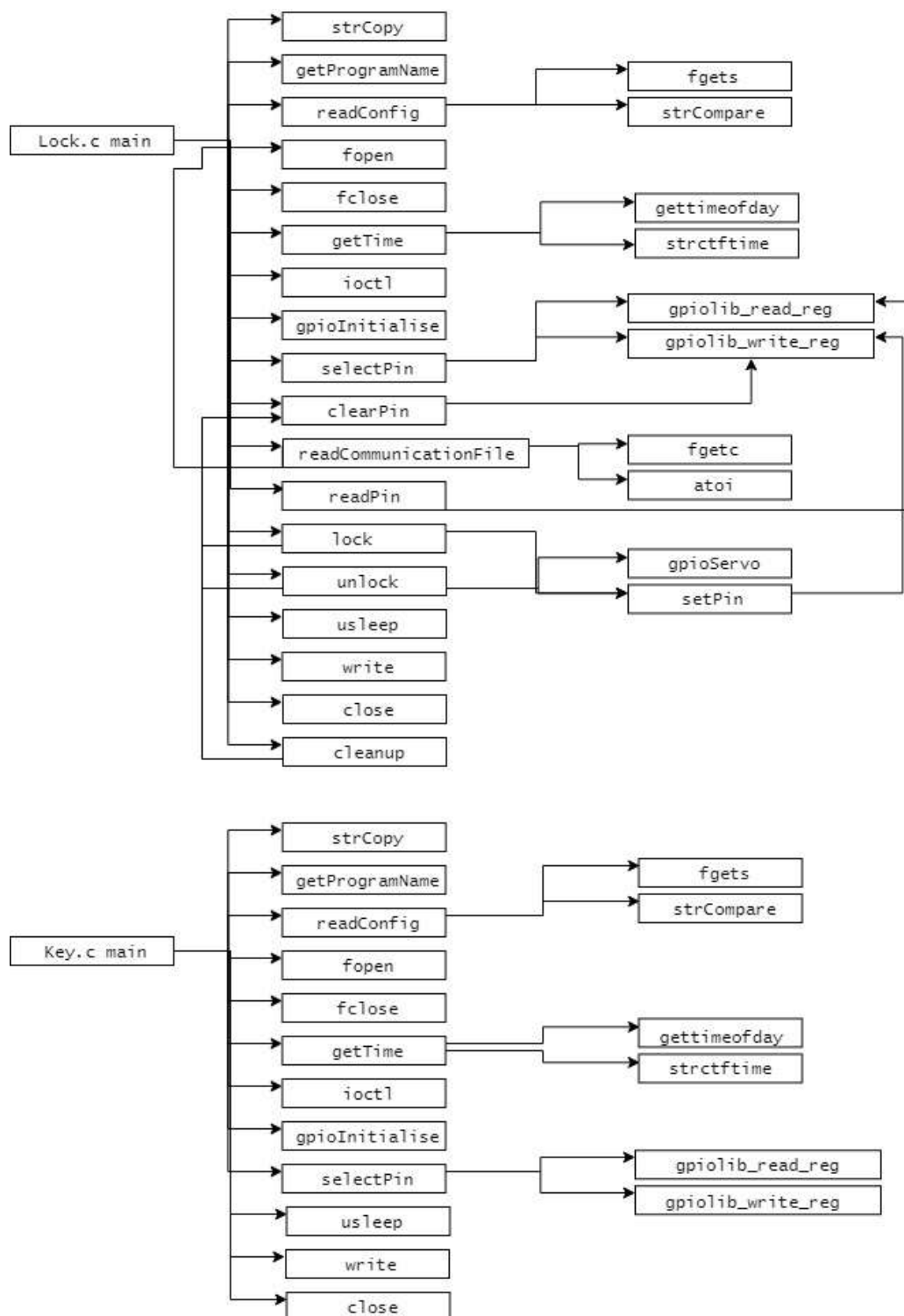


Figure 7: Function call tree diagram showing interdependencies between functions.

Logging Infrastructure, Unit File and Watchdog Implementation

Both pieces of the system have similar logging infrastructures, but it would be inappropriate to combine them since they run independently of each other. Additionally, by separating the log files for both the remote access key and lock mechanism, this allows a more detailed analysis of errors through the log files. When commands are written to the communication file they are recorded in the node-specific log file. Then it can easily be seen whether the lock mechanism receives and properly reads those commands by analyzing its own log file. This creates a useful separation between the nodes to more easily determine which one is faulting and at what point.

Logging Points within lock.c:

- System startup
- Opening the configuration file
- Opening the lock log file
 - Creating the lock log file if it doesn't exist
- Opening the communication file
 - Creating the communication file if it doesn't exist
- Opening, kicking and closing the watchdog
 - Setting the timeout value for the watchdog
 - Failure report if it cannot open
- Initialization of the GPIO Pins
 - Report if failure to initialize
- Setting of the GPIO Pins to input or output
- Writing to the communication file when the push button is released after being pressed
- Receiving a new command from the communication file
- Waiting to lock if the door is open
- Executed given command from communication file

Logging Points within key.c:

- System startup
- Opening the configuration file
- Opening the lock log file
 - Creating the lock log file if it doesn't exist
- Opening the communication file
 - Creating the communication file if it doesn't exist
- Opening, kicking and closing the watchdog
 - Setting the timeout value for the watchdog
 - Failure report if it cannot open
- Initialization of the GPIO Pins
 - Report if failure to initialize
- Setting of the GPIO Pins to input or output
- Writing to the communication file when the push button is released after being pressed

Logging Method

The logging for both is done through a macro `PRINT_MSG(file, time, programName, outputStr)`. This macro takes the file to be written to, the current time (found using `getTime()`), the program name, and a string that will be output. This macro allows for all log messages to be consistent in their style, with a modified output string based on what needs to be logged by each program.

The logging points for both files are necessary and justifiable. For both the lock and key, there are log points for when external files are opened, created, or closed. This is important in order to know when the lock and key programs have access to other files and when they don't.

Furthermore, it is important for the key to log when it writes to the communication file, so that there is record that the information has changed. Similarly, the lock should log when there is a change in the information it is reading. It should not always log what it is reading, because the information is not always changing (there would be too many messages in the log file). The lock also logs when it tells the servo motor to move, to record that the call has been made to execute the command to lock or unlock the door.

Unit File Implementation

The programs run on the startup of each pi through the implementation of the unit file. The lock mechanism Pi will run the "lock" executable compiled with `lock.c` and `piLock.c` and the remote access key Pi will run the "key" executable compiled with `key.c` and `piLock.c`. The unit file is named `sample.service` and is stored in the directory `/lib/systemd/system/` on the Pi. The unit file was disabled for testing purposes but was re-enabled for the final demo. Once the unit file executes the program, the Pi command line is still useable. Thus, any console output from the program is not printed to the Pi terminal as it normally would when the program is executed manually. The unit file was beneficial to the concept of the system as a whole as the lock and the key should automatically commence their processes once receiving power. This also allows for auto recovery if the Pi loses power. Thus, if the system shuts down unexpectedly, it will restart without the need for any manual human input. Such a feature is vital for a lock and key system, which are required to be stable and robust to increase security.

Watchdog Implementation

The watchdog was not implemented during the demonstration, however it is implemented in the code in the same way that is shown in the manual for Embedded Lab 4.

The watchdog is kicked during our loop that runs as long as the program is running (i.e., `while(1)`). So, the watchdog makes sure the program is running as intended, and is kicked for as long as the state machines run. This is true for both `key.c` and `lock.c`. In the current implementation, upon reboot, the program will reset to the default lockstate specified by the user in the configuration file. The key goes back into its loop to wait for button presses to change the information in the communication file.

Configuration

Current Configurability

Although not shown during the demonstration, the system has been tested and is known to be configurable through the specified communication file. The following values are initialized to a default value declared in the header file, `piLock.h`, but are also configurable through the `readConfig()` function that reads the config file at the specified path in the header file.

- `WATCHDOG_TIMEOUT`
- `LOCKSTATE`
- `COMM_FILE_PATH`
- `LOCK_LOG_FILE_PATH`
- `KEY_LOG_FILE_PATH`

If the configuration file does not exist at the specified location, the program will continue to execute using the default values for each of the above found in the header file.

Configuration File Format

The configuration file is expected to have the following format:

- The file may have blank enters, or lines of only whitespace between parameter specifications
- Any line containing a parameter can begin with whitespace but the parameter must begin with an Alphabetic character
- The parameter name is then read and can contain any alphanumeric character including the underscore character '_'
- Whitespace after the parameter name is accepted, but the next character after the whitespace must be the equals character '='
- If the parameter name read in the configuration file does not match with any of the parameters being looked for, the rest of the line will be ignored
- Whitespace after the '=' character is accepted, but depending on the parameter now being assigned, the first character after the whitespace must be an integer for an integer value or a '/' when specifying a file path
- Whitespace after the parameter value is also accepted, but the next character after this whitespace must be '\n' or 0

Example configuration file contents.

```
WATCHDOG_TIMEOUT = 15
DEFAULT_LOCK_STATE = 0
COMMUNICATION_FILE_PATH = /home/pi/raspShare/commFile.txt
LOCK_LOG_FILE_PATH = /home/pi/raspShare/locklogFile.log
KEY_LOG_FILE_PATH = /home/pi/raspShare/keyLogFile.log
```

State Processing

The configuration file is read by means of a state machine. The state machine is embedded within a while loop that continues to run until the input character is '\n' and this loop is embedded within another while loop that runs until the fgets() function used to allocate the next line in the configuration file to a buffer (character) array does not equal 0 or the NULL character. Essentially the state machine is run anew for each new line until the line is read to be NULL, meaning the file has reached the end. Additionally, the state machine is implemented in a manner that allows a full reading of the configuration file even if there is an error in a single line. In other words, if an invalid character (different from the expected format described on the previous page) is detected, the machine simply ignores that line and does not modify the parameter value. The state machine is implemented as follows:

- **START:** The initial state the state machine enters upon reading a new line. Transitions to
 - **PARAMETER:** if the line begins with an alphabetic character
 - **WHITESPACE:** if the line begins with a ' '
 - **COMMENT:** if the line begins with a '#' or does not begin with an alphabetic character or a ' ' (meaning an invalid character)
- **COMMENT:** In this state the rest of a line is ignored. In other words, the program loops through the buffer until it receives the character '\n'. Transitions to:
 - **DONE:** if the character being read = 0 or NULL, meaning the end of the file has been reached.
 - Otherwise since the received character is '\n' it will break out of the while(input != '\n') loop and loop once more through the while(fgets() != 0) loop to get the next line.
- **WHITESPACE:** In this state, the program loops through the buffer array until the current character is no longer ' '. Transitions to:
 - **DONE:** if the character being read = 0 or NULL, meaning the end of the file has been reached.
 - **PARAMETER:** if the next character is an alphabetic character
 - **COMMENT:** if otherwise
- **PARAMETER:** Two while loops are implemented in this state. Since the only way to enter this state is if the character being read is alphabetic, it must mean that the parameter name is currently being read. As such the first while loop loops through the buffer array, assigning buffer[i] to char parameterName[n] until the current character is a ' ' or '='. The second while loop simply loops through the buffer until it finds the '=' character specifying that the next few characters will specify the value of the parameter given in that line. ++i is done upon exit to read the next character after the '='. Transitions to:
 - **DONE:** if the character being read = 0 or NULL, meaning the end of the file has been reached.
 - **COMMENT:** if an invalid character is detected in the parameter name (non alphanumeric and not '_') or if between the parameter name and the '=' character

there is a non ' ' character or if the string parameterName does not match any of the parameters being searched for in the configuration file.

- TIMEOUT: if strCompare("WATCHDOG_TIMEOUT", parameterName) returns true
- LOCKSTATE: if strCompare("LOCKSTATE", parameterName) returns true
- FILEPATH: if strCompare("COMM_FILE_PATH", parameterName) or strCompare("LOCK_LOG_FILE_PATH", parameterName) or strCompare("KEY_LOG_FILE_PATH", parameterName) returns true.
- TIMEOUT: Two while loops are also implemented in this state. Since it is known that the current character being read should be the one right after the '=' character in the line, the first while loop loops through the buffer array until the character received is an integer. Then while the characters received are integers, it will loop through the buffer array and assign the specified value to the integer "timeout" specifying the watchdog timeout value in seconds. Transitions to:
 - DONE: if the character being read = 0 or NULL, meaning the end of the file has been reached.
 - COMMENT: if before an integer character is received, there is a non ' ' character in the line. Also transitions to comment once the specified value is read to ignore the rest of the line (loop until character received is '\n')
- LOCKSTATE: Two while loops are also implemented in this state. Since it is known that the current character being read should be the one right after the '=' character in the line, the first while loop loops through the buffer array until the character received is a '1' or a '0'. Then if the character received is a '1' it will assign a value of 1 to the integer lockState, otherwise it will assign a 0. Transitions to:
 - DONE: if the character being read = 0 or NULL, meaning the end of the file has been reached.
 - COMMENT: if before an '0' or '1' character is received, there is a non ' ' character in the line. Also transitions to comment once the specified value is read to ignore the rest of the line (loop until character received is '\n')
- FILEPATH: Upon entering this state, the char* copyPath is declared and the strCompare() function is used on the string parameterName to determine which parameter to assign the next file path to by getting copyPath to point to that parameter. Then two while loops are implemented. Since it is known that the current character being read should be the one right after the '=' character in the line, the first while loop loops through the buffer array until the character received is a '/' specifying a file path. Then the second while loop assigns buffer[i] to copyPath[i] until a ' ' or '\n' character is received.
 - DONE: if the character being read = 0 or NULL, meaning the end of the file has been reached.
 - COMMENT: if before a '/' character is received, there is a non ' ' character in the line. Also transitions to comment once the file path is read and assigned to the correct parameter to ignore the rest of the line (loop until character received is '\n')



Figure 8: State machine diagram for the state machine in the `readConfig()` function.

Testing

Every function described in the Function Description section was tested independently and is known to be working properly on any system, regardless of the pins at which the LEDs, photodiodes or buttons were connected. Different Pi's and different pin configurations were also tested with these functions. Simple LED control, button reading and photodiode reading were all tested and are known to be working.

The demo system implemented everything specified in the software design except for the `readConfig()` function, watchdog and photodiode reading component. Rather than waiting for the photodiode to read true, the state machine for the lock mechanism was made to transition straight into the lock state for the demo. It was known from separate testing that the software works correctly, but due to the photodiode circuit not working as expected on the day of the demo, this component was omitted. We could not get the watchdog code to work as we hoped, as it seemed the watchdog would not open as implemented, so it was omitted as well.

The `readConfig()` function has been tested extensively locally on a windows machine, and has shown to correctly re-assign the values for the watchdog timeout, default lock state and file paths for the log files and communication files.

The state machine reading the button on either node and controlling the servo using the servo-specific were also tested without any of the communication framework on a single Pi when the prototype lock mechanism was made. This lock mechanism just took input from the button processed it through the button state machine and locked or unlocked the door. The early prototype confirmed that the button state machine and servo controlling functions were functional.

The implementation of all of the code excluding the `readConfig()` and watchdog components as shown in **Figure 4** has been tested comprehensively on the lock mechanism Pi and breadboard set up for `lock.c` and on the remote access key Pi and breadboard set up for `key.c`. The button state machine correctly writes to the communication file and the lock mechanism state machine was also shown to correctly read from the communication file and properly execute the command given if possible.

This was all possible through the use of the logging infrastructure. Once the `key.c`, `lock.c`, `piLock.c` and `piLock.h` code had been set up and the logging infrastructure was in place, it was easy to see that the code in `key.c` and `lock.c` was properly opening files, writing to them and reading from them in order to execute the given command. The precise logging messages showed that the software was functioning exactly as was expected and carrying out the correct actions based on the input commands. We were also able to read the communication file live as it was edited by the button state machine on both the lock mechanism and remote access key, and live time logging messages were also visible when new commands were written to the communication file and executed.

Extras, Limitations, and Reflection

Limitations

A limitation of our current readConfig() function is that it does not yet have the state processing to handle auto lock or auto unlock times as specified in the project overview and objectives. We had hoped that the user would be able to specify a list of times in the configuration file when the lock mechanism should automatically write a command to the communication file to lock or unlock the door depending on whether the time was in the char* autoLock[] array or char* autoUnlock[] array. We were expecting to implement the following the in the configuration file.

```
AUTOLOCK = 7:45, 10:30, 13:00, 22:00  
AUTOUNLOCK = 7:00, 12:00, 4:00
```

These values would then be read also through the readConfig function and stored somewhere to continuously compare to the current time and execute if they became equal. This proved to be challenging as it required the use of a delimiter of sorts to read the first time in the list and assign it to an integer to then be checked throughout the execution loop in lock.c until the current time reached the specified time.

One of the already mentioned limitations is the inability of the current implementation of the code to get the watchdog to work properly. If commented out, the code will work as implemented in main() for both key.c and lock.c. Otherwise, when we attempt to open the watchdog it will throw an error saying that the watchdog could not be opened.

Reflection

The most difficult aspect of this project was the need for real-time communication between the two Pi's. Trying to find a method to reliably achieve this communication between Pi's took the most time, but we feel that our solution is most likely not the best one considering our application. Firstly, as a shared network folder it is accessible by anyone on the network. Secondly, an entire file doesn't seem necessary to communicate a simple '0' or '1' integer/character. A better custom solution most likely exists for secure direct data transfer from Pi to Pi that would be better suited to our application. If we were starting over, we would most likely try to find a simpler communication solution for both Pi's that does not involve a shared network folder. A possible alternative would be to use transmitter and receiver modules that are able to interface with the Pi. If the Pi's could be connected remotely and directly through a service that runs on startup for both Pi's that would be even better.

Additionally, we would also try running both Pi's on a portable power source such as a battery rather than connecting the Pi's to a computer. This would make the system even more portable and realistically useable, as the unit files would immediately run both the key and lock service upon startup, allowing the user to begin using both nodes separately immediately.

Appendix

Source Code

The following code was all stored on the shared network folder that was mounted directly onto the home directory for both the key and lock Pi. This allowed the necessary files to be copied into the Pi's local memory for compilation, however the communication file, log file and configuration file were always accessed through the shared network folder so both Pi's could see real time updates made to these as was necessary for communication.

- commFile.txt
 - lockLogFile.log
 - keyLogFile.log
 - lockConfig.cfg
 - lock.c
 - key.c
 - piLock.c
 - piLock.h
- *The following were later moved into piLock.h and piLock.c*
- gpioFunctions.c
 - gpioFunctions.h
 - gpiolib_addr.h
 - gpiolib_reg.c
 - gpiolib_reg.h

Once the necessary code files were copied into the local memory for each Pi they were compiled for the key and lock respectively as follows:

```
gcc -o key key.c piLock.c
gcc -Wall -pthread -o lock lock.c pilock.c -lpigpio -lrt
```

As mentioned earlier the additional compiler flags on the command for compiling the lock file are necessary in order to enable the PIGPIO library that was used to control the servo.

The code for the entire project can be accessed through the public library on git uwaterloo:

<https://git.uwaterloo.ca/krpinto/Smart-Lock/tree/master>

Peer Contribution

For the most part, one person did not tackle any part of the project by themselves. One person would begin working on the part, but eventually everyone would take a look at it and edit/debug it to ensure everything worked correctly. The following specify who did a majority of the work on each specific hardware or software component.

Hemit:

- String Compare and Copy Functions in piLock.c
- File and Code Structuring and Organization
- Config File Reading
- State machine for Lock
- Logging Infrastructure

Efaz:

- Watchdog Timer Implementation
- Communication File Reading
- State Machine for Lock
- Logging Infrastructure

Kyle:

- Circuitry
- Servo motor functions and physical locking apparatus
- GPIO Functions in piLock.c
- State machine for Key
- Unit file

Everyone:

- Pi to Pi Communication using shared network folder
- Debugging
- Testing
- Writing and editing report