User Manual
ALS Home Automation

Group UBTALKER Research and ALS Group

Date : May 2017
Prepared by : Group ALS – Home - Automation
Members : Bin Li, George Vanhoose Sr., Jordan Hoeber, Vincent Cheng

Table of Contents

A. Introduction

Document Identification

This document describes the setup and overview of the Senior Computer Engineering Design and Integration project. This document is prepared by Group ALS Home Automation for the UB Talker Research Group.

Project Overview

Our goal was to improve the life of an ALS Patient named Tim. We overtook an existing research project at the University at Buffalo, called **UBTalker** which was headed by Kris Schindler and Michael Buckley. The main goal of our project was to improve Tim's life. In order to do so we had to come up with solutions to automate his living conditions and make his life better and more practical. Some of his issues included a lack of an efficient method of communication to those around, his nurses, his family, or his friends. We have designed an emailing solution that allows him to communicate with them through wifi rather than them having to come in person or through word of mouth. There were other issues we wanted to solve as well. Such include automating his fan, his television, and his lights. By interfacing with the existing desktop application, we are able to automate these things for him. We can control the lamp, fan and television entirely through the application by use of microprocessors.

Document Overview

This document lists the specification requirements for this project. How to use the user interface of the desktop application, and how to setup the raspberry pi's for lamp, fan and television control for both software and hardware.

B. Product Description

This section is intended to give a general overview of the basis for the ALS Home Automation system design, of its division into hardware and software modules, and of its development and implementation.

Project Scenarios

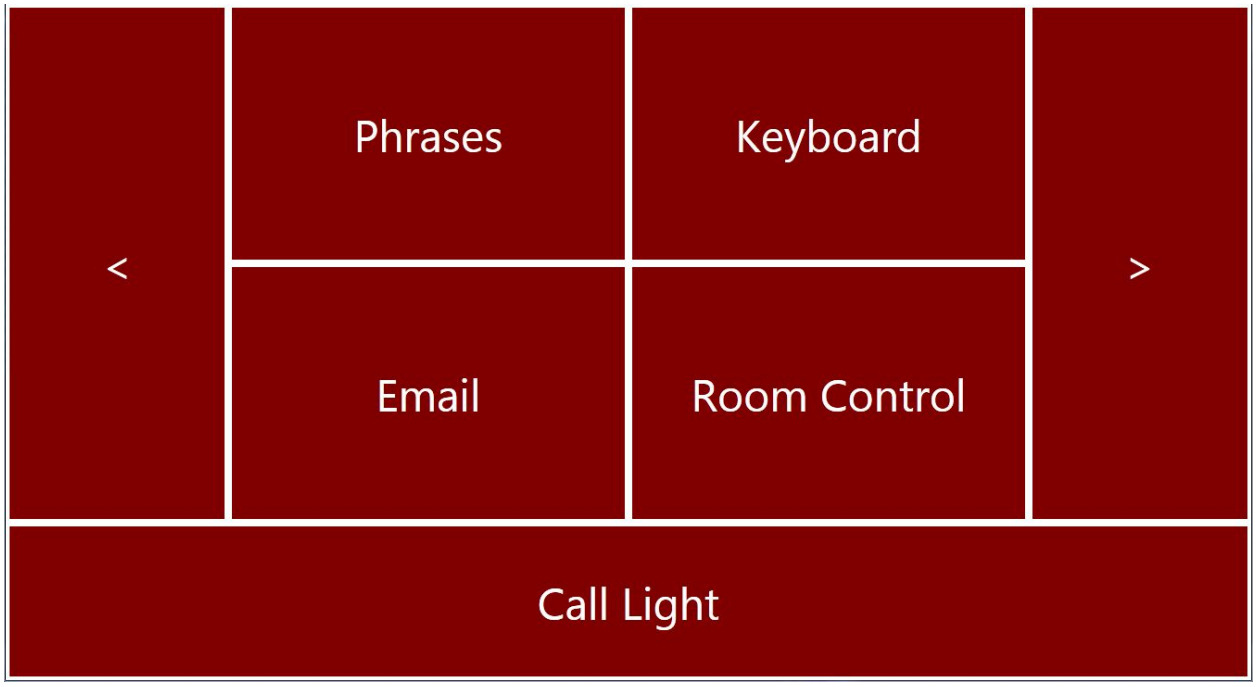
This project is intended for use by ALS patients and immobile patients but may be used by practically anyone.

System Requirements

Windows Operating System.
Laptop or desktop computer or a tablet.
Raspberry Pi (3 has on-board WiFi access, other versions must have a stable WiFi dongle or the ethernet port must be used)

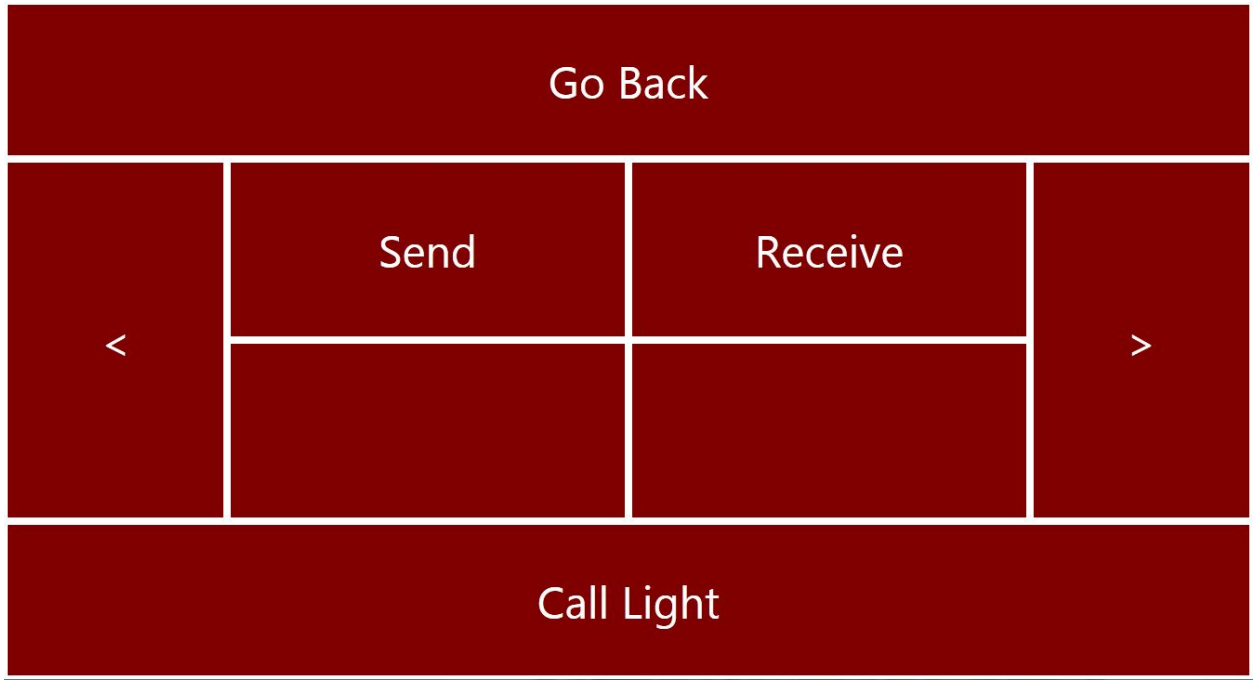
C. User Interface

The desktop application can only be run on computers with a Windows Operating System.
How to use:

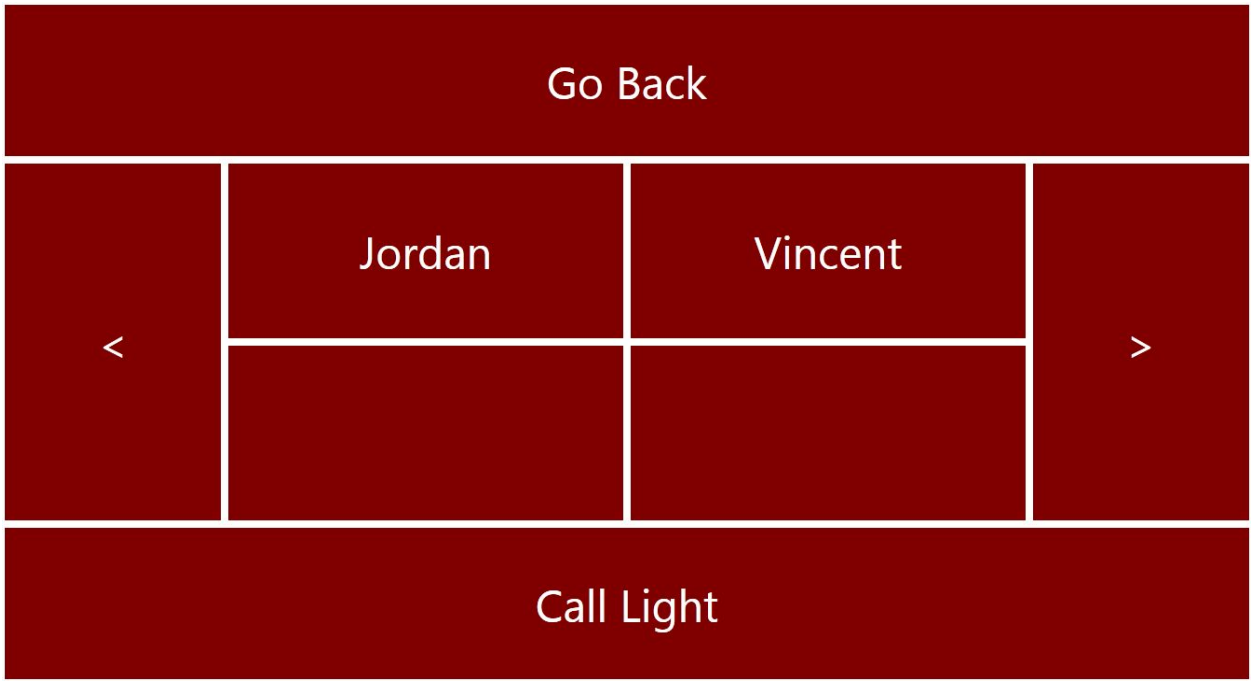


The main view is called the Hubview and specifies services and actions that can be run. To select a service, you can click, or use the eye tracker to select on any of the options, known as **Gaze Buttons**.

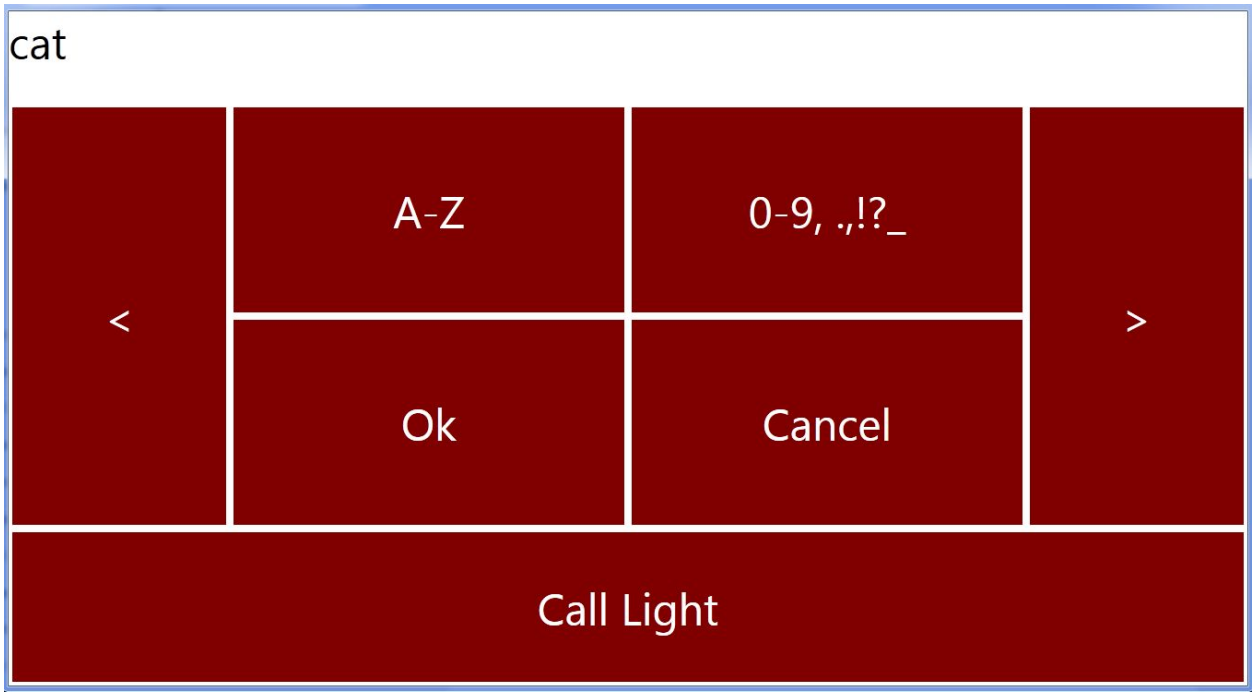
The phrases will take you to a panel of built in phrases that can be read by clicking on those already specified in the list of known phrases. Once you select a phrase from the list, you may click on the selected **phrase** gaze and the voice from your machine will read the phrase back to you. The **keyboard** button will take you to a keyboard that allows you to create messages of any any sort consisting of letters and numbers. Use this to form words or sentences that are not already listed in the phrase gaze.



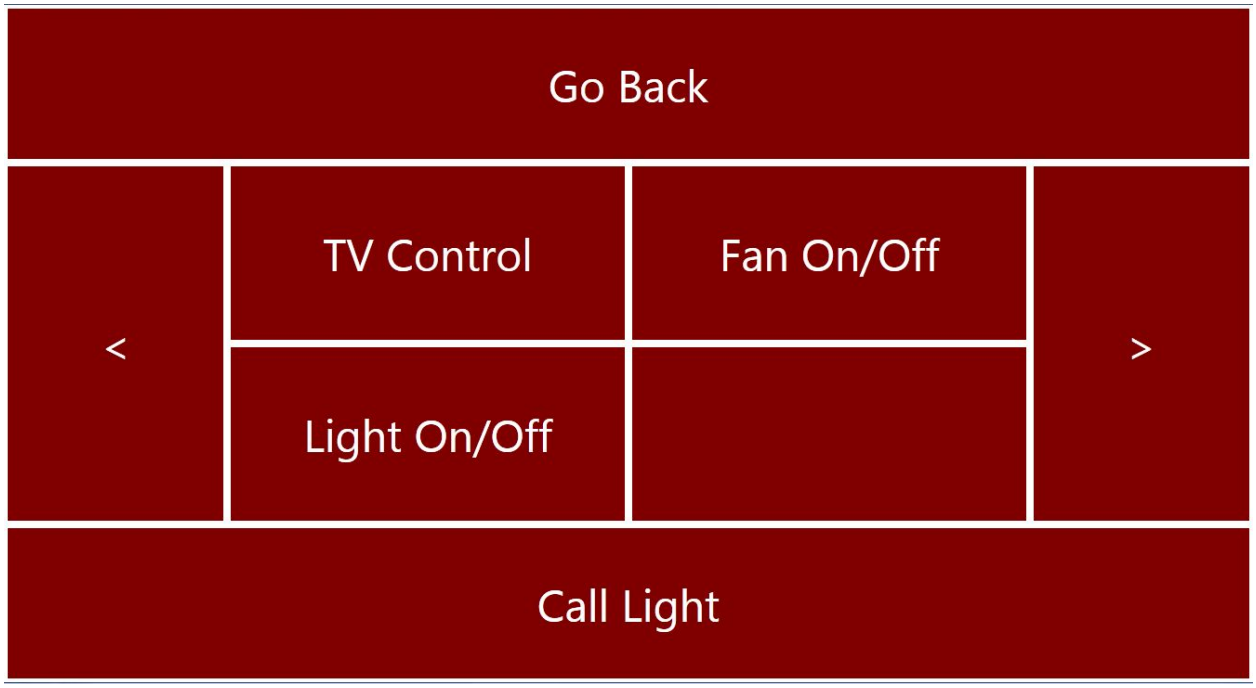
This is the **Email View**. It specifies two options for sending and receiving emails. To send an email, simply select the **Send** gaze button from the Email View.



This is the **Contacts Menu**. These contact names and their email addresses are already saved in the system and can be selected as a regular gaze and not a service. Once you select a contact in the list, it will take you to the keyboard view, where you can type your message and once you press ok, the email will send automatically.



This can be shown above. The message you wish to send will be displayed above the keyboard in a section of the view called the **Modal**. Once you press **Ok**,the message will send automatically. In this case, the message of ‘Cat’ will send to the person that you have selected. You may cancel sending an email/message at any time by pressing the cancel button.



The next feature is the **Room Control**. This feature allows us to control things such as the television, the fan, and the lights. The setup instructions are listed below.

D. Hardware Setup

Setup Raspberry Pi for Lamp and Fan control

These instructions are for connecting the Raspberry Pi to the relay, and for connecting that relay to an AC power source.

1. Ensure Pi is powered off and unplugged
2. Connect female end of color-coded wire to GPIO pin 4.
3. Fasten male end of color-coded wire to positive terminal, on the DC / control side, of the relay.
4. Connect female end of black wire to Ground pin on Pi.
5. Fasten male end of black wire to negative terminal, on the DC / control side, of the relay.
6. **!!! ENSURE EXTENSION CORD IS <u>NOT PLUGGED IN</u> !!!**
7. Remove approximately six inches of the outer casing of a grounded extension cord approximately midway through the length of the cord.
8. Strip approximately two inches off one wire -- not the ground, which should be bare copper anyway!
9. Cut the stripped wire and the ground halfway through the stripped wire.
10. Curl both ends of the stripped wire such that they form an arc with a diameter of approximately ¼ inch
11. Place the curled ends of the stripped wire under the floating plates around the terminal screws on the AC end, such that the curl points around the screw in a clockwise fashion.
12. Press each curl against the threads of the terminal screw as much as they will go.
13. Proceed to screw in the terminal screws in a clockwise fashion until the floating plates have clamped down on the curled ends of the wires.
14. Take a spare length of bare copper wire and tie it securely through the hole in the ground plate on the AC side of the relay.
15. Splice the other end of the bare copper wire to the severed ends of the ground wire.
16. Wrap the splice in electrical tape.
17. Plug the male end of the extension cord into a single-socket surge protector
18. Plug the lamp, fan, or other fixture into the female socket of the extension cord
19. Ensure the lamp, fan, or other fixture is turned on
20. Plug the surge protector into the wall outlet

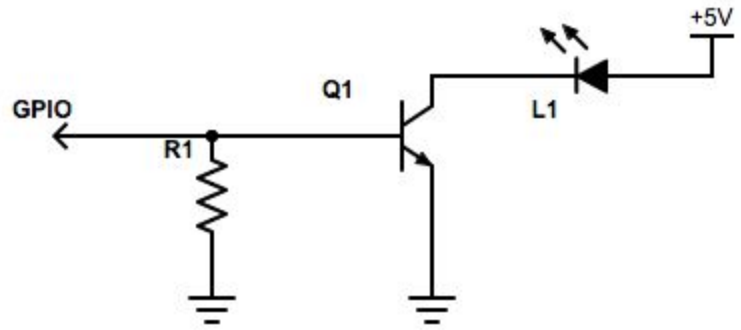
Alternatively, both ends of the severed ground wire can be tied to the hole in the ground plate.

Alternatively, if the lamp or fan have no ground connection, only the ground wire which leads to the wall outlet needs to be fastened to the relay's ground plate.

Set up for TV Controller

Full Guide can be found at the following URL
http://alexba.in/blog/2013/01/06/setting-up-lirc-on-the-raspberrypi/

The circuit for the TV Controller is as follows:



Parts needed:

- 10k-Ohm resistor
- IR Emitter LED
- PN2222 NPN Transistor
- IR Receiver Sensor - TSOP38238

GPIO22 is the pin currently configured to output the signals. Any 5V pin on the Pi can be used for the 5V supply.

E. Software Setup

TV Controller

Full Guide can be found at the following URL
http://alexba.in/blog/2013/01/06/setting-up-lirc-on-the-raspberrypi/
Use this guide to start from scratch if anything goes severely wrong.

How to copy a remote to the Raspberry Pi

- Connect an IR sensor (preferably the TSOP38238) to the Raspberry Pi using the 3.3v power, GPIO23 for input, and the Raspberry Pi's ground
- Turn off LIRC "sudo /etc/init.d/lirc stop"

- Record the data from the remote by using the following commind: “sudo irrecord -d /dev/lirc0 ~/lircd.conf”
    - Follow the prompts given by LIRC exactly
- Make a backup of the current config file “/etc/lirc/lircd.conf”
- Move the file you recorded “~/lircd.conf” into the directory “/etc/lirc/” replacing the file you just backed up
- Start LIRC back up: “sudo /etc/init.d/lirc start”

If something goes wrong with current setup check for the following issues

- Is anything obstructing the line-of-site of the IR LED
    - There must be a clear LOS
- Is the Raspberry Pi server working?
    - The Pi must be receiving commands from the client for the TV
    - Check Raspberry Pi setup details for how to fix this issue
- Is the IR LED still working?
    - The first thing to go on the circuit will most likely be the IR LED. Check if current is flowing through the LED properly

Messaging

Setting up Message Sending

- Go to the “Views/EmailView.xaml.cs” file
- Go to the “OnKeyboardInput” function
- Replace the Client Credentials with the credentials of the address Tim will be sending emails from

```
client.Host = "smtp.gmail.com";
client.Credentials = new System.Net.NetworkCredential("cse453alsgroup@gmail.com", "alshomeautomation");
client.EnableSsl = true;
```

    - Above is the code that should be updated, line that starts with “client.Credentials”
    - “cse453als…” should be replaced with the correct email address
    - “alshome…..” should be replaced with the correct password for that email address
- This will allow Tim to send emails from his email address

Setting up Contact List

- There must be a Google Drive that the person managing Tim’s contacts will have access to
- In the Google Drive’s root directory, there must be a file named “contacts.csv”
- In this file, there should be a single contact on each line in the following format

    - <Contact Name>, <Contact Number or Email>
    - Phone numbers must 10 digit and have their carrier's domain appended
    - Here's a list of carrier domains:

Carrier	Domain
AT&T	@txt.att.net (SMS) @mms.att.net (MMS)
T-Mobile	@tmomail.net
Verizon	@vtext.com (SMS) @vzwpix.com (MMS)
Sprint	@messaging.sprintpcs.com (SMS) @pm.sprint.com (MMS)
US Cellular	@email.uscc.net (SMS) @mms.uscc.net (MMS)
Boost Mobile	@myboostmobile.com
Virgin USA	@vmobl.com

    - For an example contact:
        - John Smith's Phone, 7165551235@vtext.com
- Once the Google Drive is set up with the contacts file, the app must log into the Google Drive on first use
    - To do this, open the app on Tim's computer
    - Click Messaging then Send
    - This should take you to the internet to log into the Google Drive
        - Make sure you log into the correct Google Drive
- After the first login, the login should never need to be done again, a JSON file is saved that will login automatically

Raspberry Pi

The Raspberry Pi has been configured to automatically (on boot) initialize all required GPIO pins, (i.e.: Exporting them and setting their directions) and it launches a wrapper process which launches and monitors the server program. (to which the tablet or other client device can connect) The server program takes data from its client over LAN or WLAN, interprets such data, then executes the command specified by the data in the form of a child process.
All code that executes on the Raspberry Pi is written in C
The command line to reboot the Pi with immediate effect is "`sudo shutdown -r now`"
The command line to power down the Pi with immediate effect is "`sudo shutdown -h now`"

Automatically Initialize GPIO Pins

In file "`/etc/rc.local`":
All lines which begin with a pound sign '`#`' are commented-out and ignored by the OS.
This file is most-easily accessed through the command-line-interface, (CLI) and it is easier to test its functionality if the Pi boots directly to CLI
In the file "`/etc/rc.local`", before the line containing "`exit`" and after the termination of the loop (signified by "`fi`") add the lines: (NB: It may be necessary to edit this file in "`sudo`" mode)

```
echo [nn] > /sys/class/gpio/export
echo {out/in} > /sys/class/gpio/gpio[nn]/direction
```

For each pin thus required, specified by *nn*. For example, in order to initialize pin 4 for output, this must be added:

```
echo 4 > /sys/class/gpio/export
echo out > /sys/class/gpio/gpio4/direction
```

GPIO pin 4 can be used as output by any program once this is accomplished.
In order to reconfigure pin 4 for input, use this command: (either in CLI or add it to the "`/etc/rc.local`" file)

```
echo in > /sys/class/gpio/gpio4/direction
```

In or to remove it from the available GPIO pinouts, use this command: (again, either in CLI or add it to the "`/etc/rc.local`" file for automatic-on-boot functionality)

```
echo 4 > /sys/class/gpio/unexport
```

This will immediately set that pin to Active Low, and will remove that pin from any accessibility anywhere on the Pi. This is especially useful if the pin must be repurposed as (for example) an I2C serial bus.
NB: Pin configurations are not stable -- they are erased once the Pi powers down. This is because they are not files in the traditional sense; rather, they are created by the OS, they are protected and maintained by the OS as if they were files, but they have no allocation on long-term memory! Therefore, each pin must be exported every time the Pi boots!

Set the Wrapper to Launch on Boot

In the file "`/etc/rc.local`", in the same area in which the pins are initialized, (after "`fi`" and before "`exit`") enter a line containing the absolute filepath of a program and its name as a single command, plus any arguments, exactly as if these commands were being entered in bash. Observe:

```
/home/pi/Desktop/On_Startup/wrapper &
```

This line commands the OS to launch the wrapper program in the background. It works from any directory in bash, and will work when included in the "`/etc/rc.local`" file.

Behavior of the Wrapper Program

The wrapper launches the server as a foreground process. This is not a problem since the server program's process enters the "wait" queue after setting up the socket connection, while it is listening for a connection from a client. The command line

```
[ctrl]+c
```

Will return the OS to the bash prompt without terminating the server's process or otherwise altering its process state.
The server process is configured to exit in one of two ways, Safely and Unexpectedly.

- If the server process exits safely, it disposes of all resources and exits through returning from main -- which is considered a "safe exit" by the parent process. The wrapper then exits out of its main method and will not be able to establish any more connections with the client until it is relaunched -- though it is strongly recommended that the wrapper itself be relaunched and not solely the server program. This is useful if the Pi must be worked on, and incoming connections may interfere with such work.
- If the server process exits unexpectedly, the wrapper receives a different exit code. The wrapper then uses

  ```
  system("sudo shutdown -r now");
  ```

  to restart the Pi. This restart is for two reasons:
  1. Restarting the Pi was deemed the quickest way to restore functionality of the server program.
  2. Restarting the Pi also guarantees that resource leaks are not a problem, since an improperly-terminated server likely will not have released its socket.

To test the ability of the wrapper to reboot the Pi in the event of unexpected server termination, simply locate the Process ID (PID) number of the server process (by using "`top`" in bash) and then "`kill -9 [`*`PID of server`*`]`" in bash. This will cause the server to quit unexpectedly, and it will cause its wrapper to reboot the Pi. Killing off the wrapper process will not affect the server, and the server will continue to run, except there will be no wrapper to reboot the Pi if the server quits unexpectedly. The server, whose wrapper was killed off, will also become a zombie if it does quit for any reason.

Behavior of the Server Program

The Server program multiplexes control over in-room furnishings and devices. The scope of this project was a lamp, a fan, and the TV in the client's room.
The Server sets up a TCP socket (using IPv4) which listens for an incoming connection from the client. If the server is unable to successfully call "`open()`," "`bind()`," and "`listen()`," then any allocated resources are thus released and the server waits 20 seconds before retrying. This delay is to allow time for the OS to release any zombied resources -- which it normally does.
When the Server program receives a connection from the client, it reads the data sent by the client. Under normal operation this data is the absolute filepath and name of an executable, plus any arguments as may be necessary. Currently these transmissions are limited to 128 bytes of actual data -- not counting wrapping/start/stop bits. If the transmission contains a single byte of value `0x71` (which is read by the server as a "`q`") then it triggers an immediate and safe exit of the server. (i.e.: The Pi will not reboot if the safely-exiting server was launched from the wrapper program)
For all other inputs, the server program then parses through the transmitted data -- treating it as type `char*` -- and constructs a `char*[]` array. For example:

Transmitted data:

"`/home/pi/Desktop/On_Startup/foo arg0 arg1 arg2`"
Parses into:
Array[0] = "`/home/pi/Desktop/On_Startup/foo`"
Array[1] = "`arg0`"
Array[2] = "`arg1`"
Array[3] = "`arg2`"

The server itself launches a child process, then executes the specified executable and passes all given arguments through a call to "`execv()`".
If this child process fails for any reason (suppose either the call to `fork()` fails or the call to `execv()` fails) the server process remains unaffected, thereby guaranteeing total process safety during the server's lifetime. The failed execution is not re-attempted, as this may result in the server becoming so "indefinitely fixated" on executing an incorrect command that it misses a subsequent transmission containing a correct command. Ideally, the human user on the client end will notice his request was not executed and will reattempt.
The server also records every invocation of itself, including its Process ID (PID) number, connection from the client, and all data passed to in the "`/home/pi/Desktop/On_Startup/Log_File.txt`" file. It also records all errant behavior in the "`/home/pi/Desktop/On_Startup/Error_File.txt`" file. Entries in both of these files are timestamped to the second, so they may be cross-referenced with one another during debugs and testing.
NB: A successful launch of the server program will record a blank entry in the "`Error_File.txt`" file, this will aid in determining which launch of the server had errors if any develop during the server program's life cycle.
The server may be launched without the wrapper, but if it quits unexpectedly and without a wrapper then it will not reboot the Pi.

Notes on using the "`/etc/rc.local`" file

Anything set to execute within the "`/etc/rc.local`" file will do so in "`sudo`" mode. Care must be taken to ensure each command will not cause undesired activity or behavior.
For ease of use, all executables should be contained in the same directory, (to which end the "`/home/pi/Desktop/On_Startup`" directory is provided) so as to minimize confusion among many programmers working on the Pi. It also guarantees that no executables are inside git-managed directories which may not be visible or even existent on boot!

Building the executables for Lamp and Fan control

The git-managed directory "`/home/pi/Desktop/TCP_IP_code`" is a local workbench for the remote github repository "`https://github.com/gvanhoose/TCP_IP_code.git`", a private repository maintained by George H. VanHoose Sr. It contains two branches which are relevant for controlling Lamp and Fan furnishings: "`RPi_Server`" and "`RPi_GPIO`", which contain code for the server program and the gpio access.

RPi_Server branch:
To build and place the server and wrapper executables in the "`/home/pi/Desktop/On_Startup`" directory, use command line "`make live`" in bash, within the directory "`/home/pi/Desktop/TCP_IP_code/server_code`" directory.
To build the server and wrapper without placing them, simply use command line "`make`" in bash.
To perform a build test -- where an executable is not specifically desired but a compilation check thereof is desired -- use command line "`make build_test`", which builds the wrapper and server executables, then promptly erases them.

RPi_GPIO branch:
This branch contains many .c files which can be used to export, unexport, or provide read/write access from/to the GPIO pins. The final implementation uses the programs described by "`light_toggle.c`" and "`fan_toggle.c`".
To build and place the executables in the "`/home/pi/Desktop/On_Startup`" directory, use command line "`make toggles_and_place`" in bash.
To build without placing the executables, use command line "`make toggles`" in bash.
`fan_toggle.c` and `light_toggle.c` simply read the "`/sys/class/gpio/gpio[nn]/value`" file, then write the opposite of its contents back to that same file. Since these files are guaranteed to contain either '`0`' or '`1`', and no other values, this is accomplished by

1. Reading the single byte in the file into a buffer of type `char[]` within the toggling program
2. Assigning the zeroth index of that buffer (which contains the byte read from the GPIO pin) to an exclusive-or bitwise operation of itself with `0x1`

```
buffer[0] ^= 0x1;
```

3. Writing that buffer back to "`/sys/class/gpio/gpio[nn]/value`"

These toggle programs do not save the most-recent value of the pin to long-term memory. This is because the human client may not wish to have the lamp or fan (or other furnishing as may be desired in the future) come back on after a long power outage, as he or she may be asleep or otherwise engaged. It was assumed that having a light or fan turning on after a long power outage might be extremely annoying if the human client has since fallen asleep.

F. Future Improvements

Using the Raspberry Pi to multiplex the Lamp and Fan controls will also enable a large amount of additional controls, over hardwired connections currently in-use by Lamp and Fan, by IR blasting currently in-use by TV control, or by extending the functionality of the socket

programming to reach other devices via LAN or WLAN connections. By making some modifications to the server program (source code available at the private github repository "https://github.com/gvanhoose/TCP_IP_code.git", maintained by George H. VanHoose Sr.) other LAN or WLAN connections can be made to slave devices -- probably an Arduino or another Raspberry Pi -- to allow control of devices which are not directly adjacent to the human user. Using a WLAN-accessed Raspberry Pi will also ensure that no adverse effects which may damage the Raspberry Pi can feed back into and damage the human user's tablet or other device.