



UNIVERSITY OF MISSOURI – COLUMBIA
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
"A PROUD TRADITION, ENGINEERING THE FUTURE!"

PRIVATE TEXT MESSAGE CONTROLLED COFFEE MAKER FOR REMOTE BREWING

ECE 4220
REAL TIME EMBEDDED SYSTEMS PROJECT

May 12, 2015

— Engineering Team —

Mitchell Hoppenstedt, Electrical and Computer Engineering

— Course Instructor —

Mr. Luis Rivera

Department of Electrical and Computer Engineering, University of Missouri

Table of Contents

Chapter 1. Abstract	3
Chapter 2. Introduction	4
Description	4
Goals and Objectives	4
Motivation.....	5
Chapter 3. Background	6
Chapter 4. Implementation.....	7
brew.c.....	7
main()	8
check_fifo().....	8
receive.py	10
receive_sms()	11
coffee.sh	12
coffeechk.sh	13
coffeekill.sh.....	13
Hardware Component	14
Chapter 5. Experimentation and Results.....	15
Chapter 6. Discussion and Conclusion	17
Chapter 7. Appendix A	19
Chapter 8. Appendix B	25
Chapter 9. Appendix C	28
Chapter 10. Appendix D	29
Chapter 11. Appendix E.....	30
Bibliography	31

Table of Figures

Figure 1 - Flow diagram of the brew.c program.....	7
Figure 2 - Flow diagram of the receive.py program	10
Figure 3 - Hardware component of project.....	14
Figure 4 - Output of brew.log	15
Figure 5 - Output of coffee.sh, coffeechk.sh, and coffeekill.sh.....	16

Chapter 1. Abstract

The purpose of this project is to be able to start brewing a cup of coffee remotely via text message. This is assuming that the water and coffee grounds have been placed in the coffee maker before the user tells the coffee maker to start brewing coffee. This project incorporates many things we have learned in class including pthreads, semaphores, named pipes, and a form of client-server programming. The service that is used for receiving and sending text messages is Twilio [1]. When the user sends a text message to the coffee maker, which has its own phone number through the Twilio service, it checks to see if the word “coffee” is in the message. If the message contains “coffee”, then coffee is made and a message is sent back to the user saying the coffee is being brewed. If the message does not contain “coffee”, then no coffee is made and a message is sent back to user saying the word “coffee” was not detected. When a second user sends a message to the coffee maker to make coffee and the coffee maker is already busy, then the user will get a reply saying coffee is already being made and to try again later. Throughout the project BASH scripting, Python, and C programming were used to create the final project.

Chapter 2. Introduction

This section describes the project as well as points out the goals and objectives and focuses on the motivation behind the project.

Description

The project is designed to be a more convenient way to make coffee in the morning. When a person is running behind and does not want to wait for a cup of coffee to be made, they can text a number and that will tell the coffee maker to start making a cup of coffee. This will save time by not having to stand there and wait for coffee to brew. A user can wake up, turn to their phone, text the coffee makers number, get into the shower, and the coffee will be hot and ready to drink when they go to get their coffee. This is a more convenient way to save time and energy in the morning.

Goals and Objectives

There are a few goals that needed to be met when implementing this project. One goal was to create a way for the user to know if the coffee was being made or not made and if the coffee maker was busy making coffee already or not. The objective that was focused on to achieve this goal was to create a system that responded back to the user telling them if their coffee was being made or not.

Another goal was to create a way to keep track of when the coffee was made and to also the number which the user messaged from. The objective that was focused on to achieve this goal was to create a log file that kept a record of when a user messaged the coffee maker and to write to this file every time an event took place.

The final goal was to make starting all of the programs and services needed, which was quite a bit, as easy as possible. For this to happen, the main objective that was focused on was creating BASH scripting files with all the commands inside of them to start all the programs and services that are needed. Multiple BASH files were needed to make the system easy to use.

Motivation

The motivation behind the project is a simple one: to make brewing coffee in the morning as easy as possible to save time. Most people use their phone as an alarm to wake up. This makes it even more convenient to be able to text a number and coffee will start to brew, all before the user even gets out of bed. This way they can get dressed or go through their normal morning routine and will have a cup of coffee waiting and only had to send a message for that to happen. The user can get the cup of coffee and walk out the door, ready for the day.

Chapter 3. Background

For this system, there was no method or paper in which it was replicated from. However, similar systems such as these have been implemented using the Twilio service for the purpose of the Internet of Things [2]. The company Zipwhip [3] has created a similar system in which it uses its own cloud texting messaging services to be able to send text messages to make coffee. Zipwhip's system is much more advanced and has more features than this projects system, but the requirements were also different where the projects system must constrain to sticking to concepts learned in class and implementing those concepts.

Chapter 4. Implementation

In this section, the implementation of the system and project will be discussed in depth.

brew.c

In Figure 1, the flow of the brew.c program is shown.

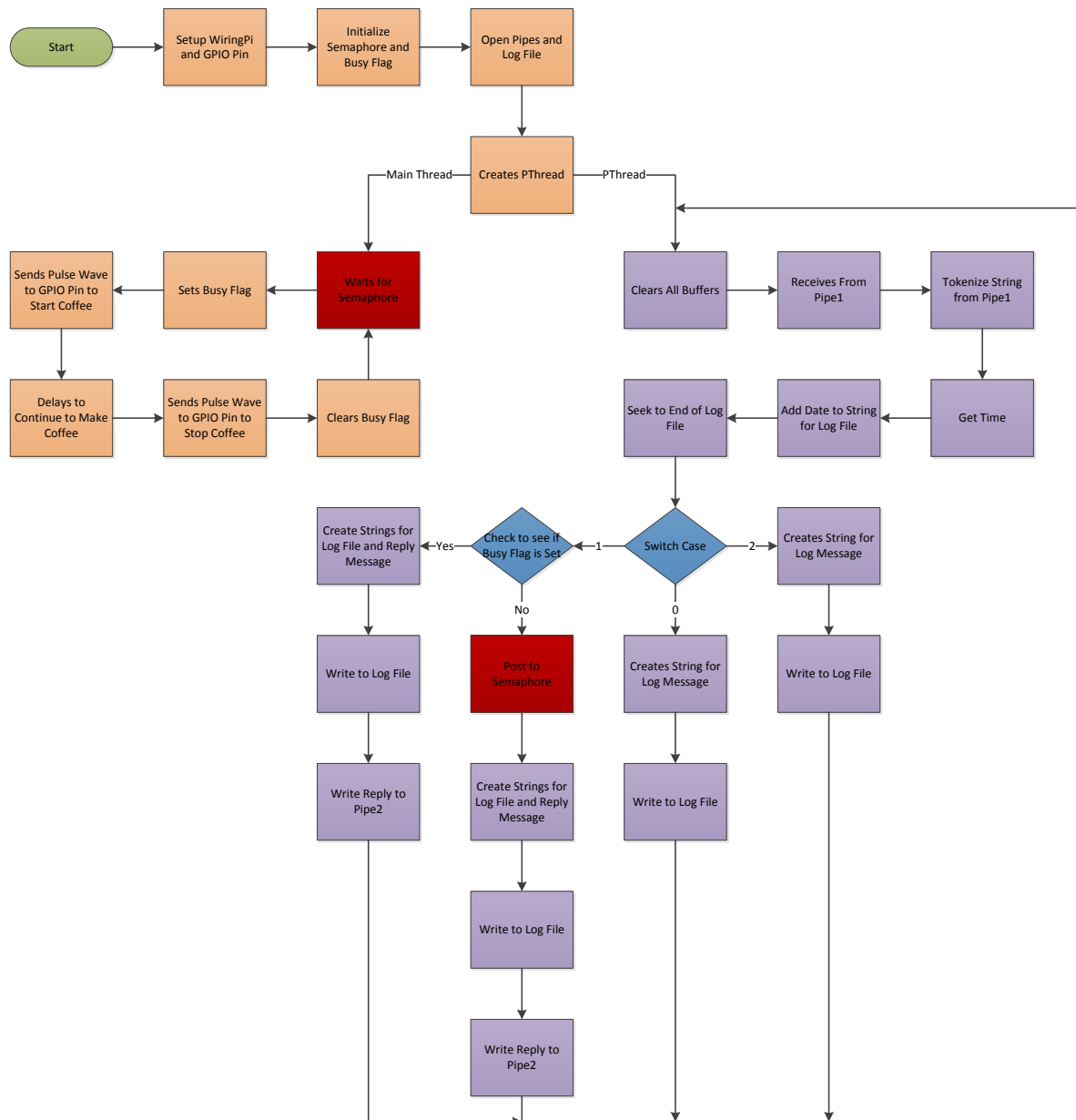


Figure 1 - Flow diagram of the brew.c program

main()

This program is the program that actually interfaces with the hardware and make the coffee maker start to brew. To start, the WiringPi library is setup and the GPIO pin 17 on the Raspberry Pi is to output. Next the semaphore and busy flag are set. After that pipe1 and pipe2 are opened for communication between the brew.c program and receive.py as well as opening the brew.log file. The pthread that is used is then created and assigned to run the *check_fifo()* function. The main program then continues to go and wait for a semaphore to be posted from the *check_fifo()* function. Once a semaphore has been posted, the program will set the busy flag to one, send a pulse wave to pin 17 to turn on the coffee maker, wait for a minute and thirty seconds (1:30), and then send another pulse wave to turn the coffee maker off. The busy flag will then be cleared to 0. The process will then wait for another semaphore post to happen to turn on the coffee maker again.

check_fifo()

This function is ran concurrently with the *main()* function through a pthread. Its purpose is to check to see if any new data has come through pipe1. At the beginning of the while loop, all the buffers are cleared and waits for data to be available to read from pipe1. Once there is data available, the string is then tokenized into two parts. The first part is the number in which texted the coffee maker. The second is an integer value that can be 0, 1 or 2. 0 means the coffee will not be made because the word “coffee” was not in the text message. 1 means that the word “coffee” was found, but it may or may not make coffee depending on if the coffee maker is busy or not (i.e. the busy flag is set). 2 means that an unknown number messaged the coffee maker. An unknown number is a number that is not in the “callers” list in the receive.py program. Next the time is captured and converted into a readable string for the user to easily read. The date and

time is then appended to the log message along with a tab and the number which messaged the coffee maker. The end of the brew.log file is the traveled to later append to it.

Now the switch case is done using the integer value that was the second token (0, 1, or 2) which tells what the program does next.

If the integer is 0, this means coffee will not be made because the word “coffee” was not in the text message. It will append to the log message a tab, the coffee was not made, and also write the log message to the brew.log file.

If the integer is 1, then it checks to see if the busy flag is set. If the busy flag is not set, meaning the coffee maker is not brewing coffee, then it posts a semaphore so that the *main()* function can make the coffee maker start. Next it appends to the log message that the coffee was made and also appends to the reply message, which is sent back to the sender, that the coffee was made. After that the log message is written to brew.log and the reply message is written to pipe2 to be read by receive.py. If the busy flag is set, then a string is appended to the log message saying the coffee was not made due to brewing in progress. Also a string is appended to the reply message saying that brewing is in progress and the user needs to try again later. The log message is then written to brew.log and the reply message is written to pipe2 to be read by receive.py.

If the integer is 2, this means that an unknown number that was not in the “callers” list in receive.py, message the coffee maker. The log message appends that an unknown number was detected and is written to the brew.log file.

receive.py

In Figure 2, the flow of the receive.py program is shown.

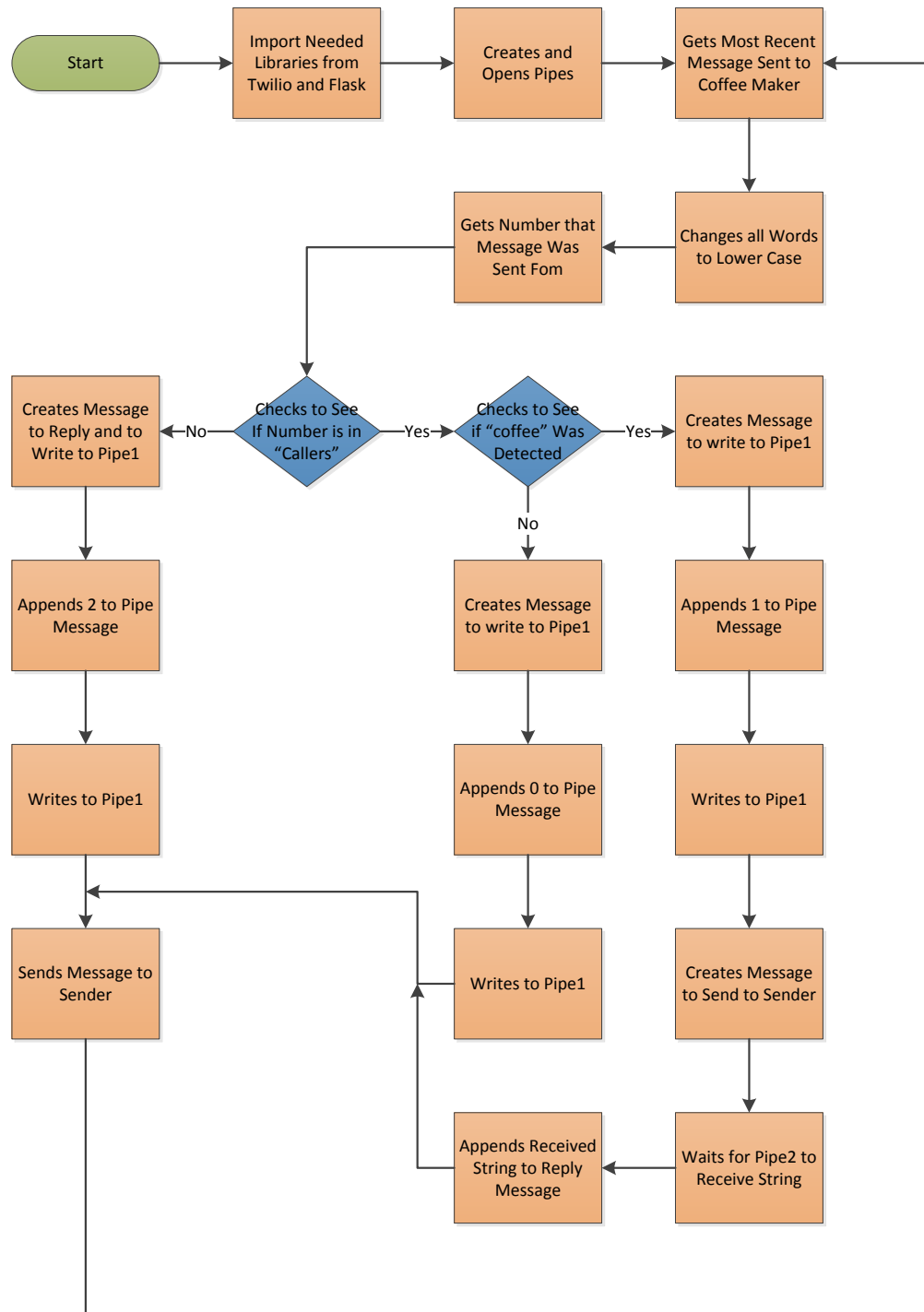


Figure 2 - Flow diagram of the receive.py program

For this part of the program, the Python language had to be learned due to the Twilio service using Python (also had other languages, but Python was the best way in my opinion) to interact with their services. To start off, the correct libraries needed to be imported (see code for the list of imports). After that the pipe names that were used needed to be set up as well as setting up the account information for Twilio and to use the Flask application. The Flask application is a way to set up a web server on your local machine and also the service that Twilio requires to use. Next is a set of trusted callers that are able to use the coffee maker. The following steps are to set up the GET and POST methods to use HTTP with Twilio, and then create pipe1 and pipe2.

receive_sms()

To start off in the *receive_sms()* function, pipe1 and pipe2 are opened to use. Following this, the last message that was sent to the coffee makers phone number was retrieved and set the body of the message to all lowercase letters for later use to see if the word “coffee” in all lowercase was detected. The phone number that the message was from in then saved. The program then checks to see if the phone number just saved was in the “callers” list.

If the phone number is not in the “callers” list, the reply message is made to say “How did you get this number?” and the pipe message, the message that will be sent to brew.c, will append the phone number the message was from as well as the number 2 for the integer code that is used in brew.c. The pipe message is then written to pipe1 and will be received by the *check_fifo()* function in brew.c.

If the phone number is in the “callers” list, then the message is checked to see if the word “coffee” is detected.

If “coffee” is detected, then the pipe message appends the phone number that sent the message as well as a 1. This is then written to pipe1 to be received by the *check_fifo()* function in brew.c. After that the reply message is set up by getting the name of person who sent the message from the “callers” list and then receives a string back from the *check_fifo()* function in brew.c, which is also appended to the reply string, which tells the user if the coffee is being made or if it is not being made due to the coffee maker already brewing coffee.

If “coffee” was not detected, the pipe message gets the phone number which sent the message, appends that as well as the integer 0 to it. This is written to pipe1 and will be received by the *check_fifo()* function in brew.c. The reply message is then set to say that no coffee was made because “coffee” was not detected.

The reply message is then finally sent to the number who sent a message to the coffee maker.

coffee.sh

This script file was more of a convenience utility and does not have much to do with class concepts; this is why I do not have a flow diagram for this file. However it is important to talk about.

This file sets up a lot of the tedious things that need to be done when setting up these programs. This first thing that is done is clearing the serv.log and ngrok.log files at the start of every new launch of the programs. This creates something to look at for debugging purposes. The next thing is starting the receive.py program in the background and starting ngrok. ngrok is a service which allows the user to open up their local network to the internet. The output of this program is redirected to the ngrok.log file. Following this, the file brew.log is created if it already has not been. The sleep command sleeps so the ngrok program can do what it needs to do

before parsing through the ngrok.log file to determine the website that the local network is exposed on. Next the brew.c program is started with sudo privileges because the WiringPi library needs it. The ngrok.log file is then parsed through using the grep command to find what the website is. The website is then put into the Twilio account page so the service knows where to go to use the programs that are being written. Information is then displayed on the screen telling the user that everything has been started.

[coffeechk.sh](#)

This script file was more of a convenience utility and does not have much to do with class concepts; this is why I do not have a flow diagram for this file. However it is important to talk about.

The purpose of this file is to check and see what the website that is needed for the Twilio service is and to see what the current processes are. This simple file just gets the site address in which the localhost is exposed on as well as getting all the process ID's that each process has. This is done so the user can manually kill all the processes if needed.

[coffeekill.sh](#)

This script file was more of a convenience utility and does not have much to do with class concepts; this is why I do not have a flow diagram for this file. However it is important to talk about.

This programs purpose is to kill all the processes that have been created. The first thing that is done is getting all the process ID's of the programs that are running. The next line that runs actually kills all the processes for the user using the kill command with SIGKILL. pipe1 and pipe2 are then removed and appends that it is the end of the server and ngrok run to the serv.log

and ngrok.log files. Information is then displayed to the screen letting the user know what has been done.

Hardware Component

In Figure 3, the hardware component is shown.

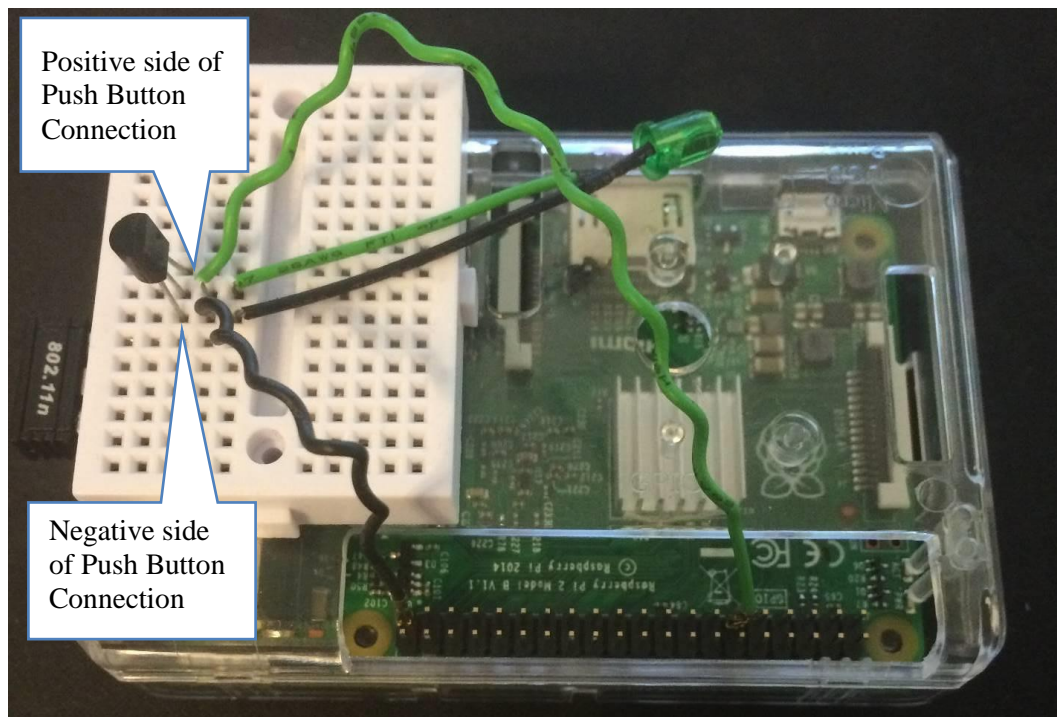


Figure 3 - Hardware component of project

This is a MOSFET transistor. The purpose of this is voltage controlled switch. When 3.3V is applied to the middle pin of the transistor (from the GPIO pin on the Raspberry Pi), it allows current to flow through from the positive side to the negative side of the push button. The push button is used to turn the coffee maker on and off. What this transistor switch does is basically bypass the push button on the coffee maker.

Chapter 5. Experimentation and Results

As shown in Figure 4, the brew.log file logs every time a message was received.

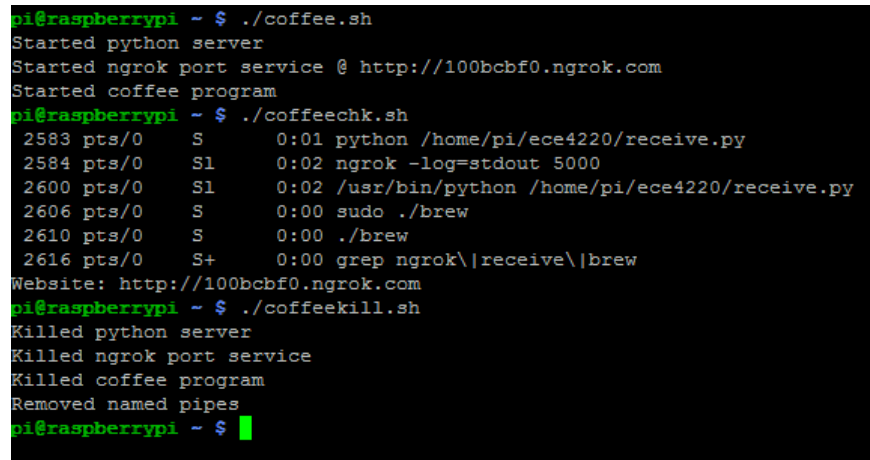
```
pi@raspberrypi ~ $ cat brew.log
Tue Apr 28 22:18:10 2015      +18165471918      Coffee NOT made
Tue Apr 28 22:18:38 2015      +18165471918      Coffee WAS made
Tue Apr 28 22:18:39 2015      +18165471918      Coffee NOT made; Brewing in progress
Tue Apr 28 22:40:21 2015      +18165471918      Coffee NOT made
Tue Apr 28 22:40:44 2015      +18165471918      Coffee WAS made
Tue Apr 28 22:42:22 2015      +18165471918      Coffee WAS made
Tue Apr 28 22:44:17 2015      +18165471918      Coffee WAS made
Tue Apr 28 22:44:24 2015      +18165471918      Coffee NOT made; Brewing in progress
Tue Apr 28 22:45:28 2015      +18165471918      Coffee NOT made; Brewing in progress
Tue Apr 28 23:00:07 2015      +18165471918      Coffee WAS made
Fri May 1 17:53:29 2015      +18165471918      Coffee NOT made
Fri May 1 17:55:57 2015      +18165471918      Coffee WAS made
Fri May 1 18:00:05 2015      +18165471918      Coffee NOT made
Fri May 1 18:00:28 2015      +18165471918      Coffee WAS made
Fri May 1 18:00:51 2015      +18165471918      Coffee NOT made; Brewing in progress
pi@raspberrypi ~ $
```

Figure 4 - Output of brew.log

For testing and experimenting with the project, the programs had to be tested in multiple ways. One way the programs had to be tested to see if the message contained the word “coffee”. As seen in Figure 4 on Tuesday, April 28th at 22:18:10 (the first entry), the message did not have the word “coffee” in it, therefore no coffee was made. This was confirmed by hooking up the coffee maker as well as an LED to designate if the coffee maker turned on. Now also in Figure 4 on Tuesday, April 28th at 22:18:38 (the second entry), the word “coffee” was detected, therefore coffee was made.

Now for testing when the coffee maker was already in use, a text message needed to be sent to the coffee maker when it was making coffee, so increasing the delay in the brew.c file in the *main()* function after the pulse wave to turn on the coffee maker will make it easier to test whether this part of the program is working. As seen in Figure 4 on Tuesday, April 28th at 22:18:39 (the third entry), the coffee maker was already making coffee, therefore could not make coffee.

In Figure 5, the output of the three commands `coffee.sh`, `coffeechk.sh`, and `coffeekill.sh` are shown in a screenshot.



```
pi@raspberrypi ~ $ ./coffee.sh
Started python server
Started ngrok port service @ http://100bcbf0.ngrok.com
Started coffee program
pi@raspberrypi ~ $ ./coffeechk.sh
2583 pts/0    S        0:01 python /home/pi/ece4220/receive.py
2584 pts/0    Sl       0:02 ngrok -log=stdout 5000
2600 pts/0    Sl       0:02 /usr/bin/python /home/pi/ece4220/receive.py
2606 pts/0    S        0:00 sudo ./brew
2610 pts/0    S        0:00 ./brew
2616 pts/0    S+       0:00 grep ngrok\|receive\|brew
Website: http://100bcbf0.ngrok.com
pi@raspberrypi ~ $ ./coffeekill.sh
Killed python server
Killed ngrok port service
Killed coffee program
Removed named pipes
pi@raspberrypi ~ $
```

Figure 5 - Output of `coffee.sh`, `coffeechk.sh`, and `coffeekill.sh`

When testing these files, the expected outcome was the actual outcome. The `coffee.sh` file started all the necessary programs in the background and told the user what programs were started. The `coffeechk.sh` file also worked as expected by displaying to the user all of the process ID's and the website that the python server is running on. The `coffeekill.sh` also works as expected. It killed all the processes and also removed the named pipes that were used.

Overall the main test was to see if sending a text message with the word "coffee" in it would turn on the coffee maker and make coffee and this succeeded.

Chapter 6. Discussion and Conclusion

For the discussion of the results, the results matched what was expected of the programs. All of the observations made sense and even when the program wasn't working as expected, following the code and understanding what it actually was doing, the outcomes of the programs were acting as expected.

From the outcome of this project the results and outcomes made sense and were what was expected. On the way there were several problems that had to be overcome to complete this project.

One of these problems included learning an entire new language, Python. Python was tricky to learn when no formal education of it was given. There were other languages that Twilio could be used with, however Python was the best documented and incorporate nicely with Linux. Java could have been used, which is well known to me, but due to its bulkier platform, Python was the better way to go.

Another problem that was encountered was figuring out how to bypass the push button that controlled the coffee to start and stop brewing. Through experimentation, the way the push button for the coffee maker works is when the button is pressed, it allows a pulsed square wave to be sent. The microcontroller inside the coffee maker must know when it sees a pulsed wave, toggle the coffee maker from either on to off or from off to on. This pulsed wave is then simulated in the program and the microcontroller in the coffee maker behaves the same way.

The last and major problem that was encountered was communication between the receiver.py program and the brew.c program. It was known that named pipes had to be used in

order to communicate between the two programs, but not knowing how to code in Python and having to interchange information between the two programs was difficult. Transferring information from receive.py to brew.c had to be done as well as going from brew.c to receive.py. It seemed the most efficient to create two pipes (pipe1 and pipe2) to exchange information back and forth to programs. pipe1 was used to send information from receive.py to brew.c and pipe2 was used to send information from brew.c to receive.py.

In conclusion there were many obstacles that needed to be worked out and overcome. In this project, semaphores, pipes, and threads were all class concepts that were implemented. In addition to those, Python language and BASH scripts were also used. This project reinforced the material that was taught throughout the semester and was a great learning experience for the student.

Chapter 7. Appendix A

Below is the code for the brew.c file.

```
/* Name: Mitchell Hoppenstedt
   Date: April 29, 2015
   Assignment: ECE 4220 - Real Time Embedded Systems
   Description: This program is used in combination with a python program that receives instructions
                via text message that will turn on and off a coffee maker. The coffee maker will only turn on
                if the word "coffee" is sent in the message. The programs will send back a message to the sender
                saying if the coffee will be made, if it will not be made, or if the coffee maker is busy and
                lets them know to try again later. The service that is used to implement text messages is Twilio.
*/

/* Libraries */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include <time.h>
#include <wiringPi.h>
#include <pthread.h>
#include <errno.h>

/* Defines the GPIO pin for WiringPi (GPIO 17) */
#define PIN 0

/* Variables used for flag, pipes, and file descriptors */
int busy_flag;
unsigned int pipe1;
unsigned int pipe2;
unsigned int brew_log;

/* Variables for messages, tokens, and time string */
char rec_pipe_message[100];
char log_message[100];
char reply_message[100];
```

```

char* tokens[2];
char* date_time;

/* Variables for thread, semaphore, and time */
pthread_t thread;
sem_t sem;
time_t rawtime;
struct tm* timeinfo;

/* This function is a pthread that runs concurrently with the main thread.
   It runs in a constant loop waiting to receive from a named pipe a phone number
   and an integer code:
   0 = No coffee
   1 = Coffee (if not in use already)
   2 = Unknown number, so no coffee
   pipe1 is used for reading and
   pipe2 is used for writing
*/
void check_fifo()
{
    /* Begins to loop immediately */
    while(1)
    {
        /* Clears all the char arrays back to zeros */
        bzero(rec_pipe_message, sizeof(rec_pipe_message));
        bzero(reply_message, sizeof(reply_message));
        bzero(log_message, sizeof(log_message));

        /* Reads from pipe1 */
        if(read(pipe1, &rec_pipe_message, sizeof(rec_pipe_message)) >= 0)
        {
            /* Tokenizes the incoming message and saves the phone number in tokens[0]
               and the integer code in tokens[1] */
            tokens[0] = strtok(rec_pipe_message, " ");
            tokens[1] = strtok(NULL, " ");

            /* Gets the date and time of day */
            time(&rawtime);
            timeinfo = localtime(&rawtime);

            /* Gets the date and time into a readable string and
               makes last character of string null terminator instead of newline */
            date_time = asctime(timeinfo);
            date_time[strlen(date_time) - 1] = '\0';
        }
    }
}

```

```

/* Starts to concatenate the log message string to log to file */
strcat(log_message, date_time);
strcat(log_message, "\t");
strcat(log_message, tokens[0]);

/* Seeks to end of brew_log to append to file */
lseek(brew_log, 0, SEEK_END);

/* Switch statement for integer case */
switch(atoi(tokens[1]))
{
    /* This case is when coffee will not be made. The word "coffee" was not found in the text message
    and the integer code is 0 */
    case 0:
        /* Appends to log_message that the coffee was not made and write to the log file */
        strcat(log_message, "\tCoffee NOT made\n");
        write(brew_log, log_message, sizeof(log_message));
        break;

    /* This case is when coffee will be made if the busy flag is not 1. The word "coffee"
    was found in the test message and the integer code is 1 */
    case 1:
        /* Checks to see if the busy flag is not 1 */
        if(!busy_flag)
        {
            /* Posts to semaphore letting code in main function to execute and coffee to brew */
            sem_post(&sem);

            /* Appends that coffee was made to log file and sets the reply message to
            send to the named pipe that will reply back to the sender in a text message */
            strcat(log_message, "\tCoffee WAS made\n");
            strcat(reply_message, ", thanks for your message. Your coffee is being brewed!\n");

            /* Write to log file that coffee was made*/
            write(brew_log, log_message, sizeof(log_message));

            /* Write the reply_message to the pipe2 with the reply text message */
            if(write(pipe2, &reply_message, sizeof(reply_message)) != sizeof(reply_message))
                printf("Error writing to NAMED PIPE\n");
        }
        /* When the busy flag is set to 1, means the coffee maker is already making coffee */
        else
        {
            /* Appends to log file that coffee was not made because it is already brewing and

```

```

        sets the reply message to send to the named pipe that will reply back to the sender in
        a text message*/
        strcat(log_message, "\tCoffee NOT made; Brewing in progress\n");
        strcat(reply_message, ", another cup of coffee is being brewed. Try again later!\n");

        /* Write to log file that coffee was not made*/
        write(brew_log, log_message, sizeof(log_message));

        /* Write the reply message to the pipe2 with the reply text message */
        if(write(pipe2, &reply_message, sizeof(reply_message)) != sizeof(reply_message))
            printf("Error writing to NAMED PIPE\n");
    }
    break;

    /* This case is when an unknown number sent a text message and the integer code is 2 */
    case 2:
        strcat(log_message, "\tUNKNOWN NUMBER No coffee\n");
        write(brew_log, log_message, sizeof(log_message));
        break;
    }
}

/* When pipe was not read correctly, print error and exit pthread */
else
{
    printf("Error reading NAMED PIPE\n");
    pthread_exit(0);
}
}

}

/* This is the main function that sets up the wiringPi library, initializes variables,
opens pipes, creates pthread, and write to the GPIO pin to allow the coffee maker to brew.
*/
int main (void)
{
    /* Sets up WiringPi library to use and sets pin mode */
    wiringPiSetup ();
    pinMode(PIN, OUTPUT);

    /* Initializes semaphore */
    sem_init(&sem, 0, 0);

    /* Sets busy flag to 0 */
    busy_flag = 0;

```

```

/* Checks to see if pipe1 opens properly */
if((pipe1 = open("pipe1", O_RDONLY)) < 0){
    printf("Error opening NAMED PIPE 1\n");
    return -1;
}

/* Checks to see if pipe2 opens properly */
if((pipe2 = open("pipe2", O_WRONLY)) < 0){
    printf("Error opening NAMED PIPE 1\n");
    return -1;
}

/* Checks to see if brew.log opens properly */
if((brew_log = open("brew.log", O_RDWR | O_CREAT)) < 0){
    printf("Error opening BREW LOG\n");
    return -1;
}

/* Creates pthread to run check_fifo() function */
if(pthread_create(&thread, NULL, (void*)&check_fifo, NULL) == 0)
{
    /* Loops, but will only perform when a sem_post was called */
    while(1)
    {
        /* Waits for sem_post to be called and sets the busy flag */
        sem_wait(&sem);
        busy_flag = 1;

        /* This provides a square pulse to turn on the coffee maker */
        digitalWrite (PIN, HIGH);
        delay(500);
        digitalWrite(PIN, LOW);
        delay(90000); // 5000    = 0:05
                     // 90000   = 1:30
                     // 180000  = 3:00
                     // 1200000 = 20:00

        /* This provides a square pulse to turn off the coffee maker */
        digitalWrite (PIN, HIGH);
        delay(500);
        digitalWrite(PIN, LOW);

        /* Sets the busy flag to 0 */
        busy_flag = 0;
    }
}

```

```
    }  
    /* If pthread was not created */  
    else  
        printf("Error - Failed to create Pthread\n");  
  
    /* Exits program */  
    return 0;  
}
```

Chapter 8. Appendix B

Below is the code for the receive.py file.

```
# Imports needed #
import os, sys, logging
import twilio.twiml
from datetime import date
from flask import Flask, request, redirect
from twilio.rest import TwilioRestClient

# Creates named pipe variables #
pipe1_name = 'pipe1'
pipe2_name = 'pipe2'

# Enables logging to a file instead of stdout #
# Uses the Flask app #
logging.basicConfig(filename='serv.log', level=logging.INFO)
app = Flask(__name__)

# Account information used with Twilio #
account_sid = "#####"
auth_token = "#####"
client = TwilioRestClient(account_sid, auth_token)

# List of numbers and people to reply back with name #
callers = {
    "+18165471918": "Mitchell",
}

# Used to use HTTP GET and POST #
@app.route("/", methods=['GET', 'POST'])

# Receives SMS from user and replies back if coffee is being made or not #
def receive_sms():

    # Opens the two pipes used for reading and writing #
    pipeout = os.open(pipe1_name, os.O_WRONLY)
    pipein = open(pipe2_name, 'r')
```

```

# Gets list of all messages that have been sent to +18164941011 #
# and gets the last messages that was sent to it #
messages = client.sms.messages.list(to="+18164941011")
sentence = messages[0].body.lower()

# Print for debugging purposes #
#print sentence

# Gets the number that the message was sent from #
from_number = request.values.get('From', None)

# Checks to see if number is in the callers list shown above #
if from_number in callers:
    # Checks to see if "coffee" is in the message #
    if ("coffee" in sentence):

        # Saves message saying coffee will be made #
        # Write to pipeout the from number and 1 #
        pipe_message = from_number + " 1"
        os.write(pipeout, pipe_message)

        # Saves the message to reply to sender #
        # Reads in message form pipein #
        message = callers[from_number]
        message += pipein.readline()[:-1]

    # When "coffee" is not in message #
    else:
        # Write to pipeout the from number and 0 #
        pipe_message = from_number + " 0"
        os.write(pipeout, pipe_message)

        # Saves the message to be replied #
        message = "The word \"coffee\" was not detected. No coffee made."

# Saves message asking how the person got the number #
else:
    # Saves message saying number is not recognized #
    # Write to pipeout the from number and 2 #
    message = "How did you get this number?"
    pipe_message = from_number + " 2"
    os.write(pipeout, pipe_message)

# Sets up a Twilio twiML response #
# and saves the message to the response message#

```

```
resp = twilio.twiml.Response()
resp.message(message)

# Returns the response as a string #
return str(resp)

# Creates a named pipe if there is not one #
if not os.path.exists(pipe1_name):
    os.mkfifo(pipe1_name)

# Creates a named pipe if there is not one #
if not os.path.exists(pipe2_name):
    os.mkfifo(pipe2_name)

# Used to initiate and run in debug mode #
if __name__ == "__main__":
    app.run(debug=True)
```

Chapter 9. Appendix C

Below is the code for the coffee.sh file.

```
#!/bin/bash

# Designates the start of a server run #
echo "----- START OF SERVER RUN -----" > serv.log
echo "----- START OF NGROK RUN -----" > ngrok.log

# Starts the python and ngrok programs in the background #
python ~/ece4220/receive.py &
ngrok -log=stdout 5000 > ngrok.log &

# Creates "brew.log" is not already created #
if [ ! -e "brew.log" ] ; then
    touch brew.log
fi

# Sleeps for 3 seconds to allow programs and connections to start and starts the "brew" program #
sleep 3
sudo ./brew &

# Saves the site that it is using #
site="$(grep -o 'http://[0-9a-z]\+\.\ngrok\.com' ngrok.log | uniq)"

# Displays on screen what programs were started #
echo "Started python server"
echo "Started ngrok port service @" $site
echo "Started coffee program"
```

Chapter 10. Appendix D

Below is the code for the coffeechk.sh file.

```
#!/bin/bash

# Saves the website to a variable #
website="$(grep -o 'http://[0-9a-z]\+\.\ngrok\.com' ngrok.log | uniq) "

# Lists all processes used and prints out the website that was used #
ps ax | grep 'ngrok\|receive\|brew'
echo "Website:" $website
```

Chapter 11. Appendix E

Below is the code for the coffeekill.sh file.

```
#!/bin/bash

# Saves the PID's to kill them #
var="$(ps ax | grep 'ngrok\|receive\|brew' | awk '{ print $1 }' | tr '\n' ' ')"
$(sudo kill -s SIGKILL $var)

# Removes the named pipes #
rm pipe1 pipe2

# Appends to files to designate the end of the log #
echo "----- END OF SERVER RUN -----" >> serv.log
echo "----- END OF NGROK RUN -----" >> ngrok.log

# Displays on screen what the script did #
echo "Killed python server"
echo "Killed ngrok port service"
echo "Killed coffee program"
echo "Removed named pipes"
```

Bibliography

- [1] "Twilio," 12 May 2015. [Online]. Available: <https://www.twilio.com/>. [Accessed 12 May 2015].
- [2] I. Wigmore, "Internet of Things (IoT)," June 2014. [Online]. Available: <http://whatis.techtarget.com/definition/Internet-of-Things>. [Accessed 12 May 2015].
- [3] Z. Lutz, "TextSpresso machine brews caffeinated goodness via text messaging (video)," 7 April 2012. [Online]. Available: <http://www.engadget.com/2012/04/07/textspresso-machine-brews-caffeinated-goodness-via-text-message/>. [Accessed 12 May 2015].