TECHNICAL GUIDE

Voice Assistant

FINISHED WORKING ON THIS DOCUMENT ON THE 10/05/19.

Author

ALEX RANDLES

15302766

Supervisor

CATHAL GURRIN

TABLE OF CONTENTS

1 ABSTRACT

1 MOTIVATION

2 RESEARCH

4 DESIGN

7 IMPLEMENTATION

10 SAMPLE CODE

17 PROBLEMS & RESOLUTION

20 RESULTS

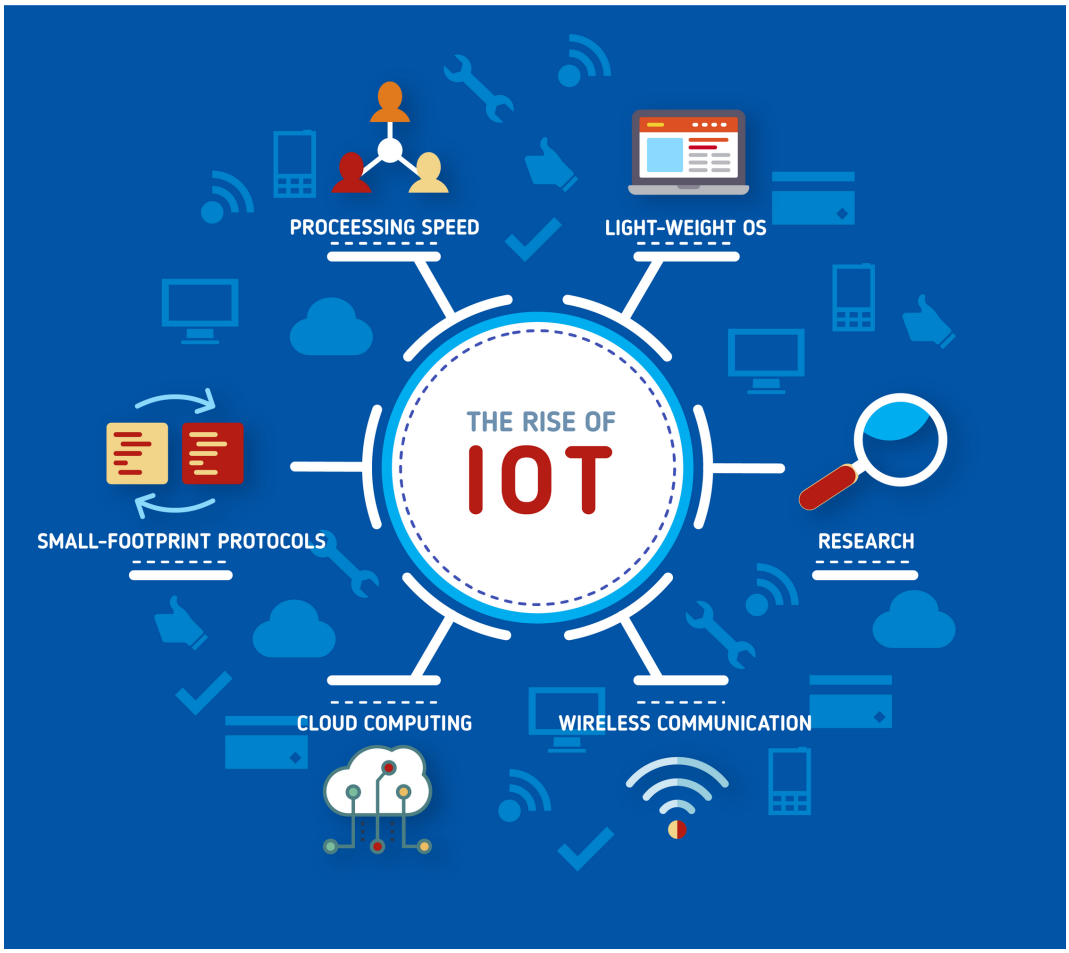
20 FUTURE WORK

ABSTRACT

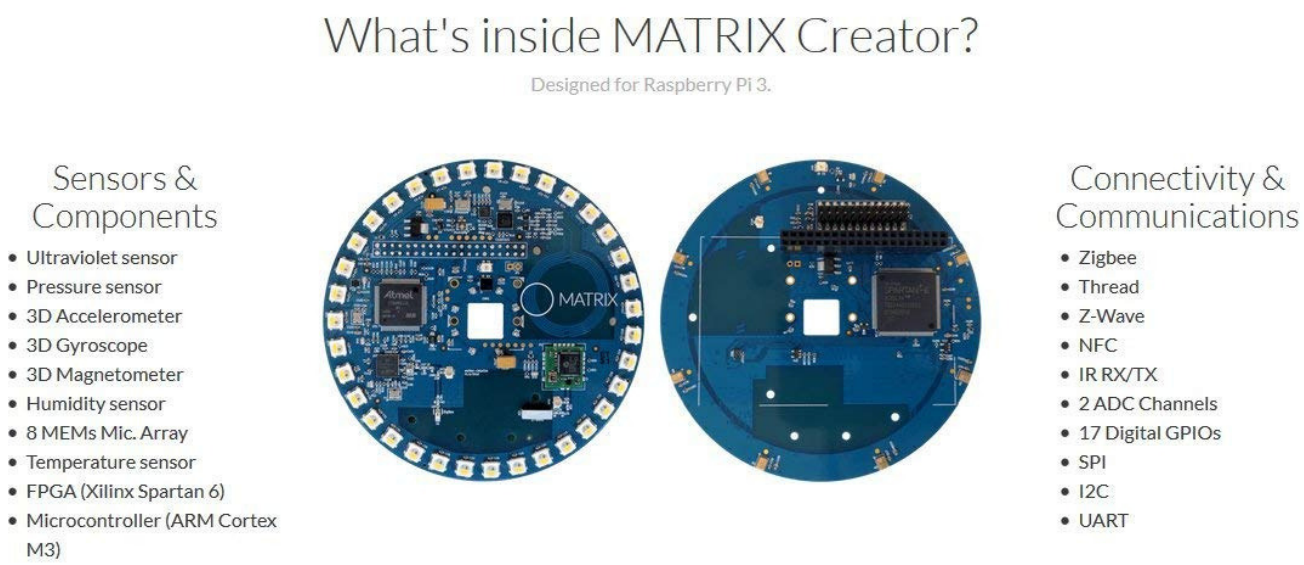
The Voice Assistant is a voice controlled device that can assist you with a wide variety of task's such as controlling a television or smart home device, reading or sending emails, playing music, receiving news and weather updates.

MOTIVATION

Whilst on work experience for INTRA, I was introduced to an area of computing which I was rather unfamiliar with called the Internet of things (IOT). After my work experience, I developed a keen interest in this area and was amazed at how software could control physical devices. Therefore, I thought it would be immensely interesting to design a project which could control a smart home device. Furthermore, my strong interest in Artificial Intelligence also led to me being incredibly interested in developing a voice assistant. The Amazon Alexa is a successful piece of software which has always amazed me and I thought it would be quite interesting to try and develop my own version of this. I believe it is extremely important that software is accessible and usable by all. Thus, I sought to develop a voice assistant as I feel it is a user friendly device which anyone can use regardless of their level of computing knowledge or inability to use computers.



https://taazaa.com/the-technologies-that-enable-the-internet-of-things/



https://taazaa.com/the-technologies-that-enable-the-internet-of-things/

RESEARCH

SIMILAR DEVICES

I began my research by looking into what similar devices are on the market. I was aware of the popularity of the Amazon Alexa and it was therefore no surprise to discover that it is the most popular similar device on the market. I researched how the device works and found that it uses a far-field microphone array to record the user.

HARDWARE REQUIREMENTS

I needed to find a device that was small and felt a device about the same size as the Amazon Alexa would be appropriate. This device would also need to support a programming language I know, be capable of recording sound, have a WIFI and Bluetooth connection.

HARDWARE RESEARCH FINDINGS

I found two main devices that fit my criteria, the Raspberry Pi and Arduino. Initially, The Arduino seemed like a good contender however I then found out it required a lot of electronics skill and was programmed using C++. The Raspberry Pi had a perfect set up already and I could program it using any language I knew. The sole problem the Raspberry Pi presented was that it lacked a built in microphone. After doing some research I found the Matrix Creator Board, pictured above. This was a Raspberry Pi development board with lots of built in functionality such as Microphone Array, IR transmitter & receiver, Zigbee, Z-Wave and an Led Array. Therefore, I selected the Raspberry Pi as I felt this device in conjunction with the Matrix Creator Board was exactly what I needed for my project.

SOFTWARE REQUIREMENTS

I required a device that was programmable with a language I had an abundance of previous experience with. Thus, the choice was either Java or Python as these were the two languages I had the most experience with and felt would offer a great deal of support with libraries.

SOFTWARE RESEARCH FINDINGS

After discovering the Matrix Creator board, I had to research the platforms they offered to developers. I found they offered two different programming layers which fitted exactly what I needed. The first layer was Matrix Core which was fully developed and provided lots of functionality that could be programmed with Python. The second layer was Matrix Lite which was still in production but was being developed to be programmable with Python. The Matrix Lite layer was a simplified version of Matrix Core however unfortunately I discovered it while it was still in development. At first, I felt the Matrix Core programming examples looked slightly complicated. Nonetheless, I knew I would be capable of using this software effectively if I conducted lots of research.

Since I knew I was using Python I had to ensure there was a library available that would translate speech-to-text as programming this myself would simply prove too difficult. I found a library called "SpeechRecognition" which offers a number of different services to translate such as Google, IBM, Bing and more.

INFORMATION SOURCES

Some really helpful information sources:

- https://matrix-io.github.io/matrix-documentation/#programming-layers
- https://pypi.org/project/SpeechRecognition/
- https://snowboy.kitt.ai/

RESEARCH CONCLUSION

After completing my initial research I felt I had enough confidence to go ahead and order my Raspberry Pi and Matrix Creator development board. Through talking with fellow students and researching the internet, I found an Irish company which offered the Raspberry Pi and all other accessories needed. This company is called "RS Radionics" and is conventionally located near the college in Dublin 12. As I had found a company who offered the Raspberry Pi and all other accessories needed, I then had to research where I could source the Matrix Creator development board from. This proved more difficult than I had anticipated as I discovered that a very small number of companies sell the development board. I found a UK company called "farnell" which was selling them but after receiving my order they emailed me stating that the device could not be shipped to Ireland. However, I found a loophole by using a UK parcel motel address who then delivered the development board to me.

DESIGN
DESIGN COMPONENT DIAGRAM

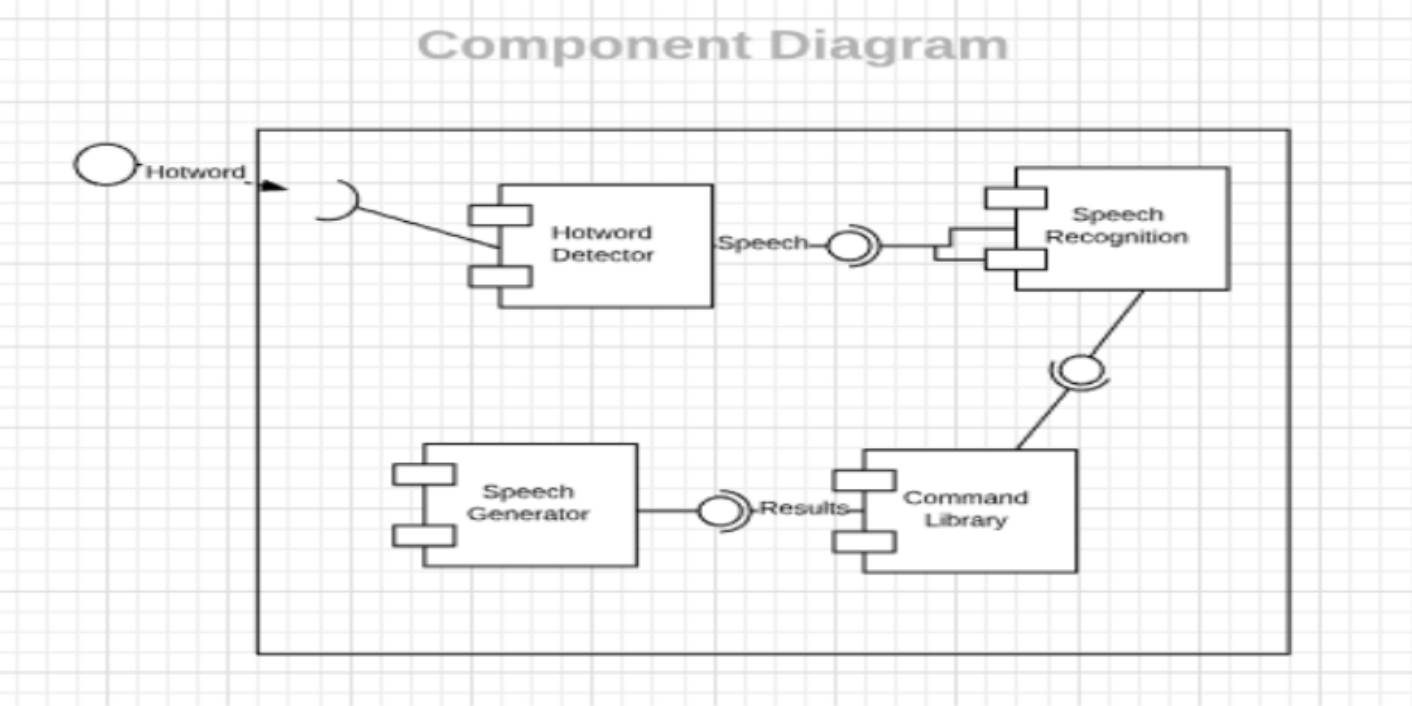


Figure 1.1 Design Component Diagram

The original design component diagram consist of most of the same components as the implementation component diagram. The main difference to be observed relates to the command library component. In the original component diagram the command library did not consist of all the commands I have implemented in my finished design. The primary reason for this difference is because I was somewhat unsure of how difficult it would be to implement the command despite having conducted thorough research.

MODULARIZATION

In my initial design I had planned to run each command as a separate script. This would mean my hotword script would run the command recogniser script which would then run the command script. Once I started implementing my design I quickly realized this would not be a viable option as each script would be run as a separate process which would each have there own resources and this caused a delay in execution speed.

Instead I decided to implement my command library as a package and each command would be a module within the command library package. This would allow me to re use code, import the functions I needed and most importantly increased execution speed thus reducing the time for the user.

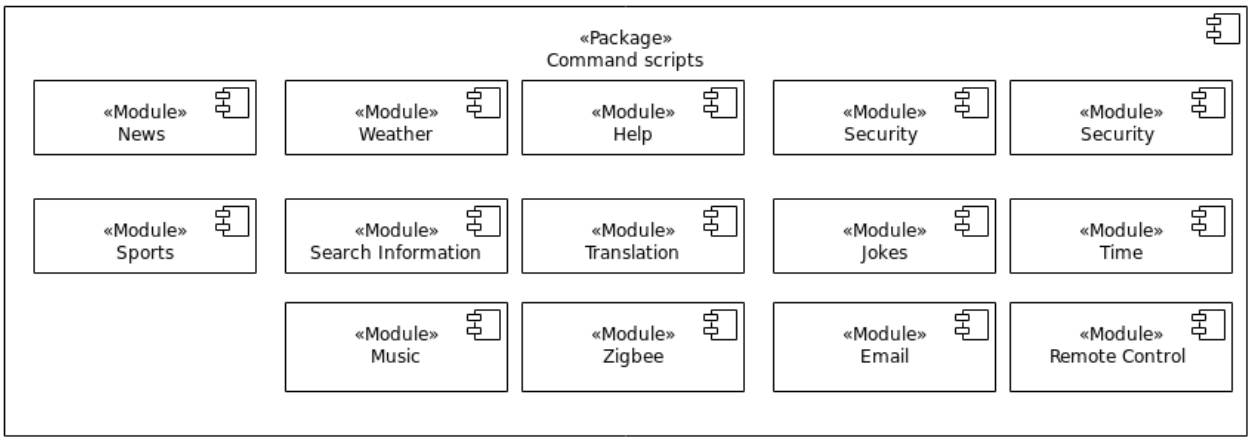


Figure 1.2 Command scripts package

DESIGN DATA FLOW DIAGRAM

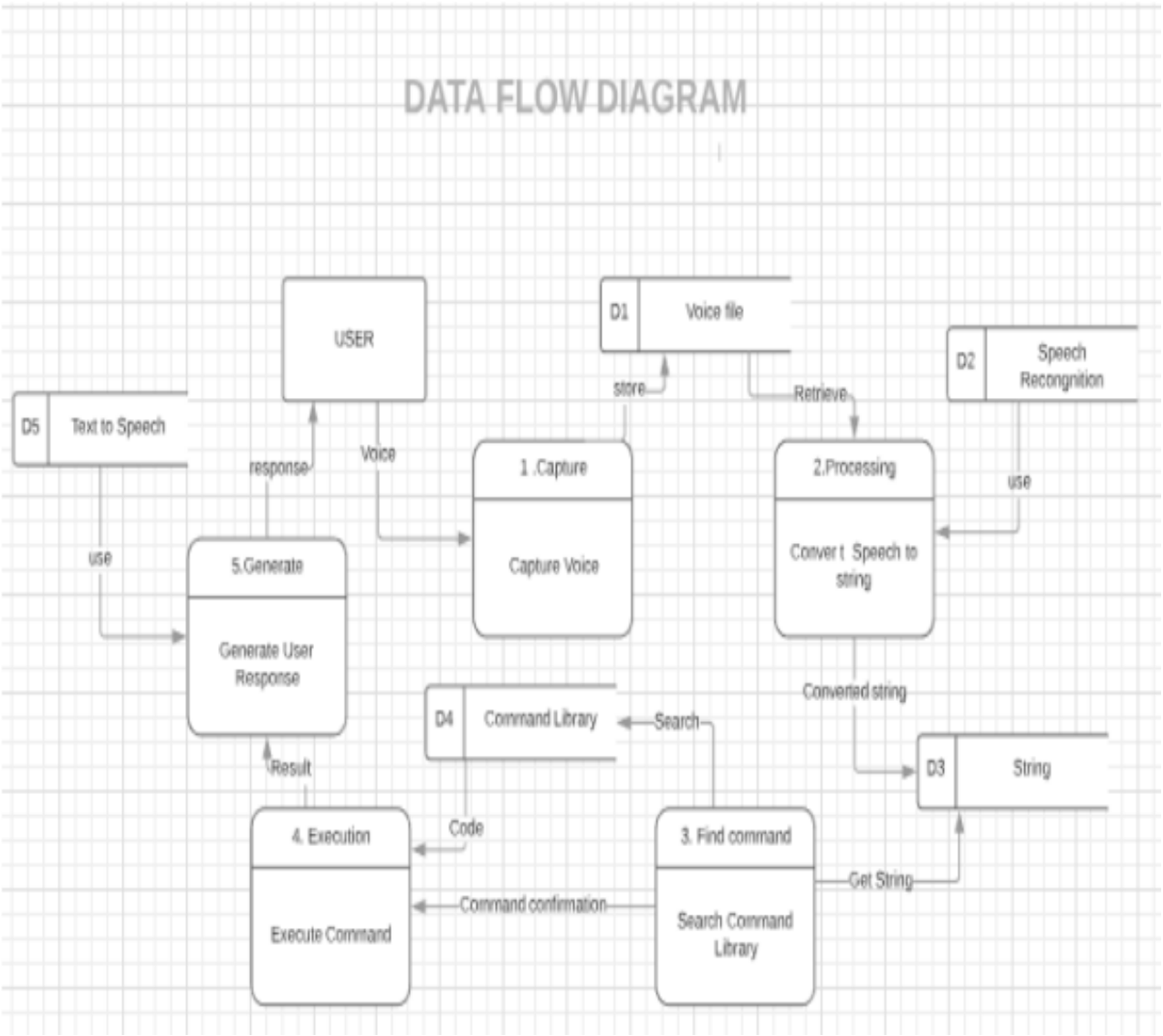


Figure 1.3 Design data flow diagram

The design & implementation data flow are very similar. The speech flows into the command recogniser script which then p;rocesses using speech recognition and then searches the command script library I have implemented, then it executes the command. In my design I planned to call the script as a seperate process.When implementation I created each command as a module which I could import then I would call the function within that module.

IMPLEMENTATION

The list of commands I have implemented in the system

- **News command** - This will return a list of the top five news headlines from the RTE RSS news feed.
- **Sports command** - This will return a list of the top five news headlines from the RTE RSS sports feed.
- **Weather command** - This will return you a brief weather forecast from OpenWeatherMap API.
- **Time command** - This returns the time using Pythons built in time module.
- **Email command** - This command will either return the last unread email or send an email depending on which script is called.
- **Remote control command** - This command either record an IR signal or play one back depending on which script is called. This command uses matrix creators IR transmitter and reciever. The communication is carried out using Pythons pigpio module.
- **Zigbee command** - This command either pair a Zigbee device or control a previously pair device. This uses matrix creators built in Zigbee module.
- **Security command** - This command will change the settings within the security database. The security database is called by the hotword detector to find out which hotword is currently set.
- **Music command** - This command has a number of different functions. It can play a song, pause a song, stop a song and shuffle the users playlist. The users previously searched songs are saved in the songs database in the music command library.
- **Search information command** - this command returns a brief summary for a search term from Wikipedia.
- **Translation language -** This command will return a translated sentence. This command processes the users input to find the language and sentece the user wants to translate. It uses a languages JSON file to match the target language and then uses Pythons speech translator module.

COMPONENT DIAGRAM

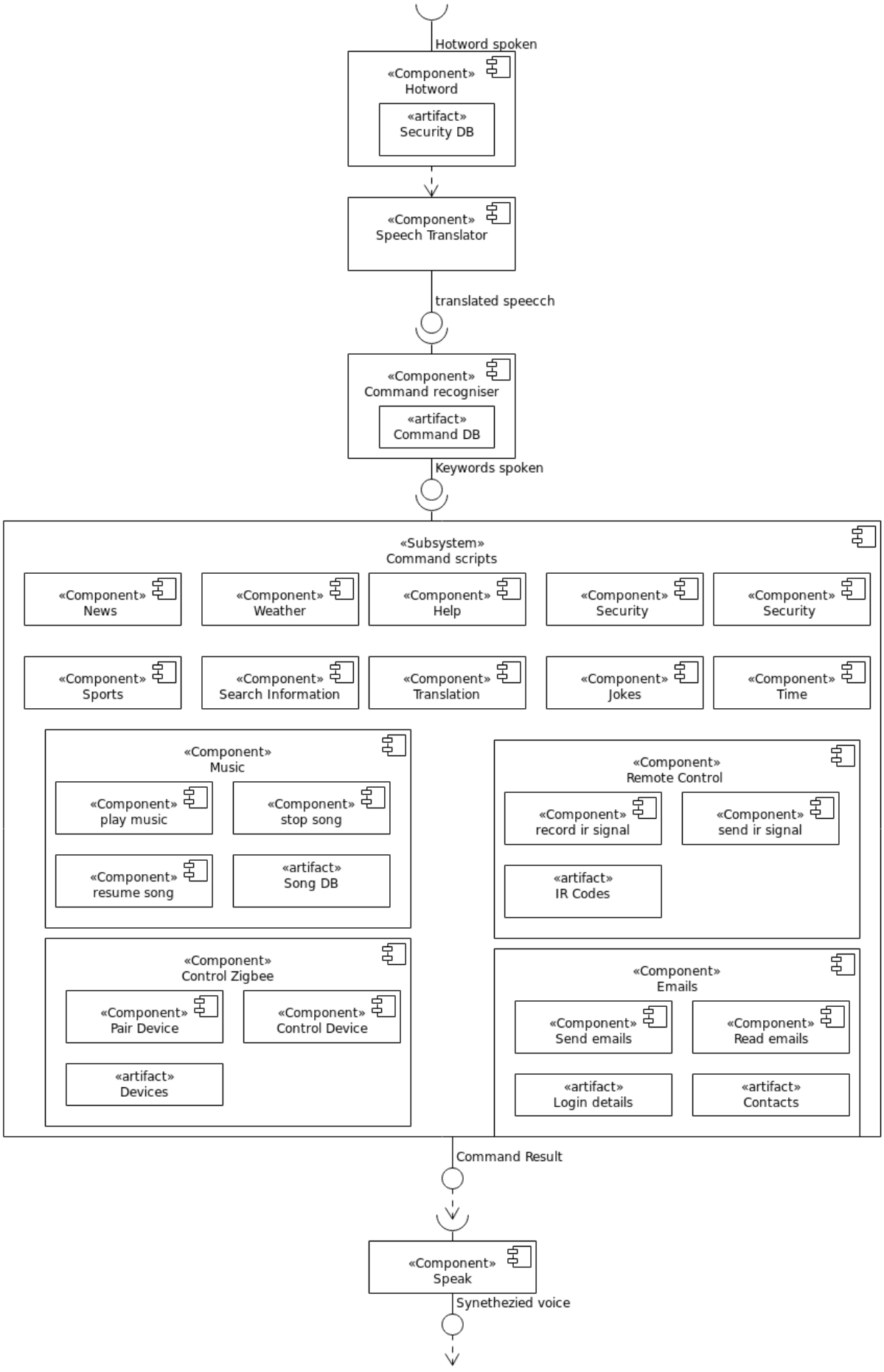


Figure 2.1 Implementation component diageam

WEATHER COMMAND SEQUENCE DIAGRAM

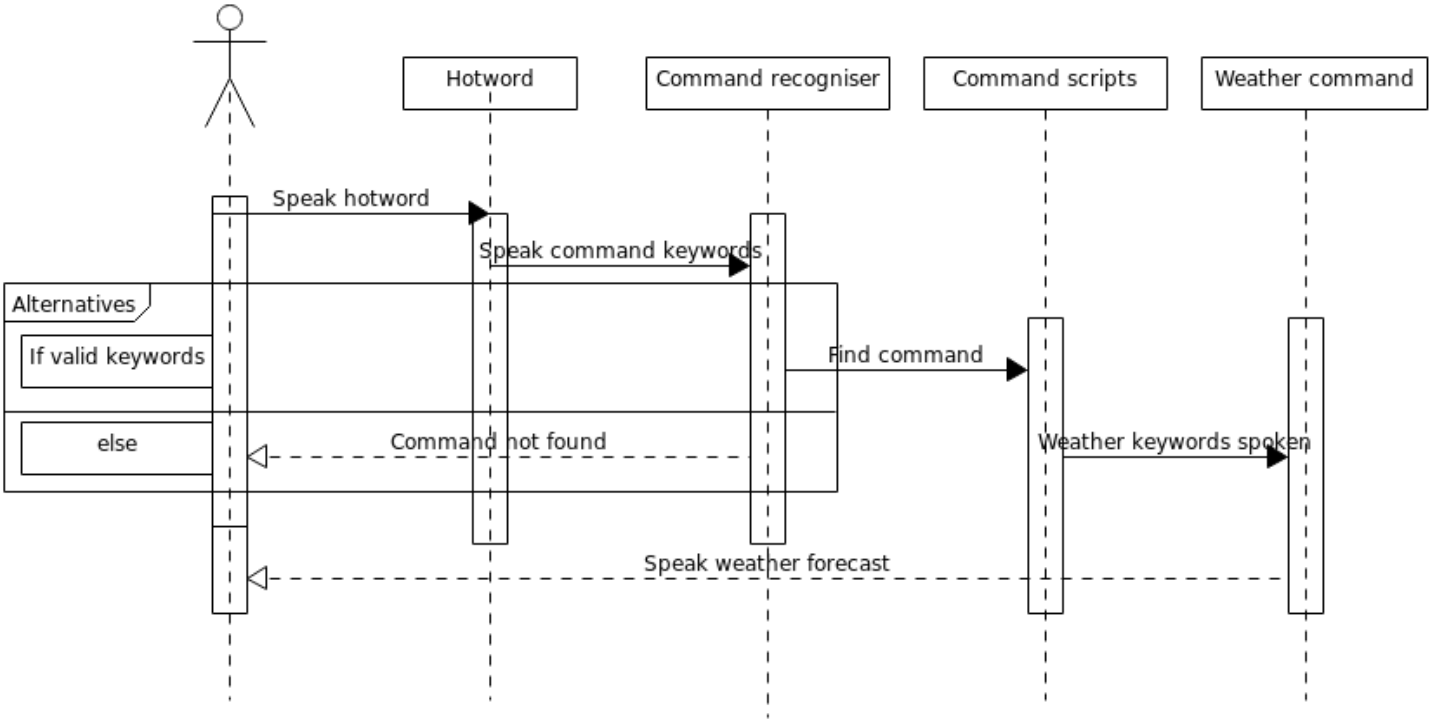


Figure 2.2 Weather command sequence diagram

EMAIL COMMAND SEQUENCE DIAGRAM

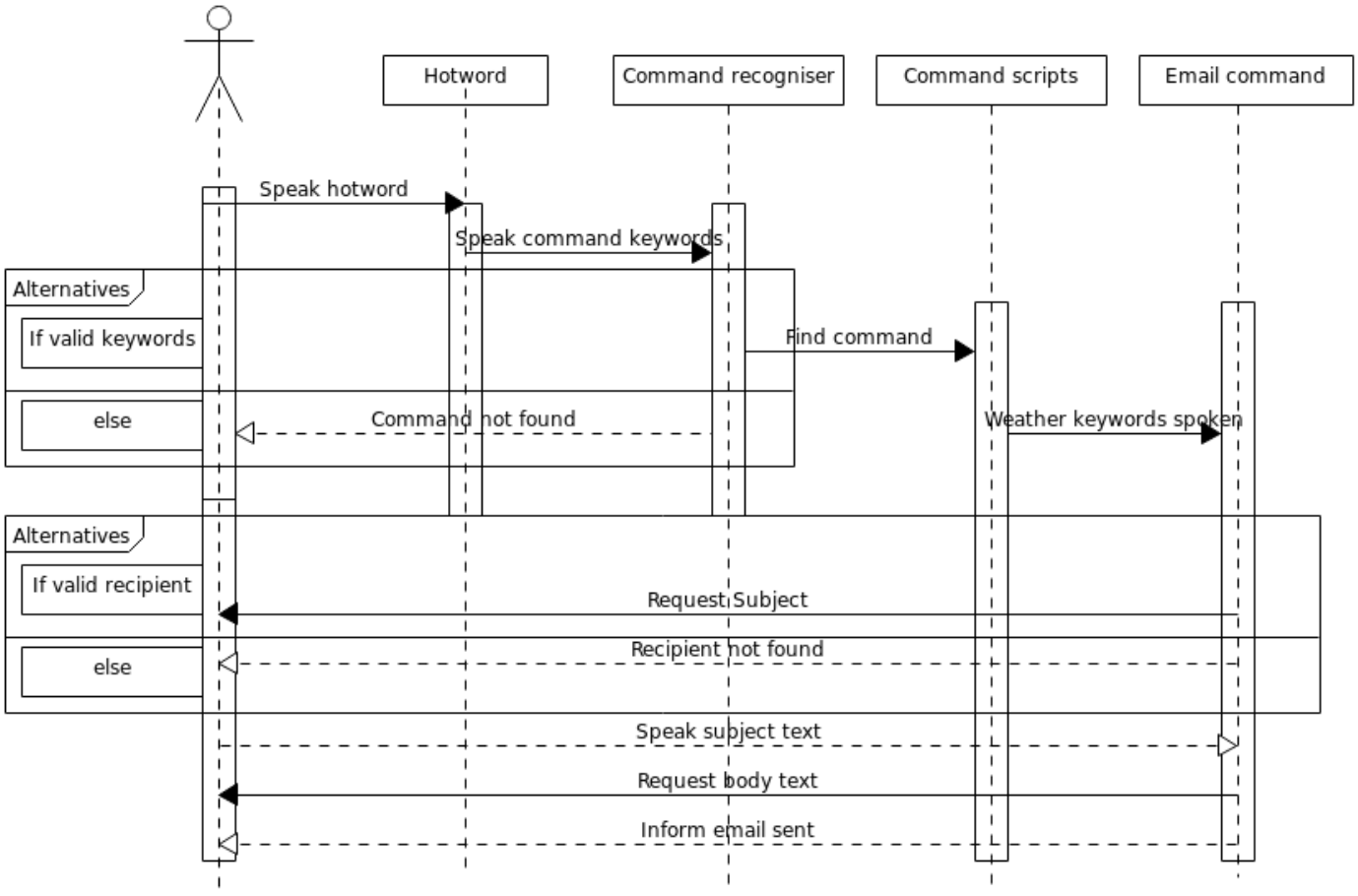


Figure 2.2 Email command sequence diagram

SAMPLE CODE

HOTWORD DETECTION

```
28  # return the hotword for current security setting
29  def get_hotword_set():
30      # connection to database which stores all security information
31      with sqlite3.connect(security_db) as connection:
32          cursor = connection.cursor()
33          cursor.execute("SELECT * FROM security")
34          # return the universal or personal model depending on settings
35          hotword_file = cursor.fetchall()[0][1]
36          print(hotword_file)
37          return "/home/pi/2019-ca400-randlea2/src/" + hotword_file
38
39
40  # get current hotword set from security table
41  detector = snowboydecoder_arecord.HotwordDetector(get_hotword_set(), sensitivity=0.5)
42
43  def detected():
44      #  hotword has been detected, start command recogniser script
45      detector.terminate()
46      subprocess.call(["/home/pi/2019-ca400-randlea2/src/suspend_sound_services.sh"])
47      snowboydecoder_arecord.play_audio_file()
48      command_recogniser.query_user()
49      print("hotword detected")
50
51
52
53
54  def detect_hotword():
55      # need to stop arecord to prevent chunk overflow when recording
56      subprocess.call(["/home/pi/2019-ca400-randlea2/src/stop_arecord.sh"])
57      global detector
58      signal.signal(signal.SIGINT, signal_handler)
59      # kill all recording before hotword detection starts
60      print('Listening... Press Ctrl+C to exit')
61      listening_leds.listening_leds()
62      detector.start(detected_callback=detected,
63                     interrupt_check=interrupt_callback,
64                     sleep_time=0.03)
65
```

Figure 3.1 Extract from hotword.py

This piece of code is an extract from hotword.py. The function of this code is to listen for and to detecthe hotword. It uses snowboy hotword detection module to carry out this function. Firstly, it searches the security database to check which hotword is currently set. This will be either a personal model or universal depending on the current settings. After this it will change the LEDs to the listening configuration to make the user aware that the device is now listening for the hotword. Hereafter, the device will be continuously reading the microphones audio stream and once it detects the hotword, it will call the detected function. This function will firstly suspend all current audio being played and then it will play a "ding" sound. After this it will call the query_user function in the command recogniser.

QUERYING USER

```
21  def query_user():
22      try:
23          speaking_leds.speaking_leds()
24          spoken_words = translate_speech.start_translator()
25          create_command_table()
26          add_all_commands()
27          command = find_matched_keywords(spoken_words)
28          # find command if words spoken otherwise do nothing
29          if spoken_words  == "":
30              hotword.detect_hotword()
31          elif command is not None:
32              command_script = find_module_location(command)
33              print(command_script)
34              run_script(command_script, command, spoken_words)
35          else:
36              speak.speak_to_user ("command not found")
37              hotword.detect_hotword()
38      except:
39          speak.speak_to_user("problem with command, try again")
40          hotword.detect_hotword()
```

Figure 3.2 Extract from command_recogniser.py

This function is the main point of interaction for the user when they want to activate a command. This is called after the hotword has been spoken. It will try to find a command for the words spoken and call that command module.

COMMAND DATABASE

```
58  def add_all_commands():
59      # command keywords must be stored as string with whitespace then split as sql cannot store list
60      #                           (command_name,command_key_words,module location,require_command_line?,activation_speech)
61      command_scripts = [("joke", "joke", "joke.jokes", "false", ""),
62                         ("send_email", "send email", "emails.send_email", "true", ""),
63                         ("read_email", "read email", "emails.read_unread_email", "false", ""),
64                         ("time", "time", "time.get_time", "false", ""),
65                         ("help", "help", "help.get_help", "false", ""),
66                         ("news", "news", "news.get_news", "false", ""),
67                         ("sports", "sports", "sports.get_sports", "false", ""),
68                         ("weather", "weather", "weather.get_weather", "false", ""),
69                         ("security_on", "security on", "security.change_security", "true", ""),
70                         ("security_on", "security off", "security.change_security", "true", ""),
71                         ("search_info", "what is", "search_information.search_wiki", "true",""),
72                         ("search_person", "who", "search_information.search_wiki", "true",""),
73                         ("control_candle", "candle", "tv.send_ir_signal", "true", ""),
74                         ("record_signal", "record", "tv.record_ir_signal", "false", ""),
75                         ("pair_device", "pair", "smart_plug.pair_device", "false", ""),
76                         ("control_plug", "plug", "smart_plug.control_plug", "true", ""),
77                         ("translator", "translate", "translation.translate_language", "true", ""),
78                         ("play_music", "play", "music.play_music", "true", "searching for song"),
79                         ("pause_song","pause song", "hotword", "false", ""),
80                         ("play_song", "play song", "music.resume_song", "false" , ""),
81                         ("stop_song", "stop song", "music.stop_song","false", ""),
82                         ("shuffle_playlist", "shuffle playlist", "music.play_music", "true", "")
83                         ]
84
```

Figure 3.3 Extract from command_recogniser.py

This is the entries within the command database. Each entry consists of a tuple. This tuple contains the following (command name, command keyword, command module, requires user arguments and an activation speech). This design allows me to add a new function to the system simply by creating the module and then adding the entry into the command database as shown above.

FINDING COMMAND

```
100 def find_matched_keywords(spoken_words):
101     # find the correct script user wants to activate
102     with sqlite3.connect(command_db) as connection:
103         cursor = connection.cursor()
104         # add if not in database
105         cursor.execute("SELECT * from commands")
106         results = cursor.fetchall()
107         all_command_key_words = []
108         matches = []
109         for command in results:
110             command_key_words = command[1].split()
111             spoken_key_words = spoken_words.split()
112             # string for closest match
113             all_command_key_words.append(" ".join(command_key_words))
114             if set(command_key_words).issubset(set(spoken_key_words)):
115                     matches.append((command, command_key_words))
116         print(matches)
117         print("matches")
118         # if commands with similar keywords
119         if matches:
120             return find_closest_match(matches)
121         return None
```

Figure 3.4 Extract from command_recogniser.py

This is the function to match the users spoken words with a command. If it finds more then one potiental match it will call another function with the matches it has found that will find the closest match out of all the potiental matches.

CALLING COMMAND MODULE

```
181 def run_script(module_name, command, spoken_words):
182     # check if  arguments
183     check = requires_arguments(command)
184     speak_activation_sentence(command)
185     print(command)
186     if "." not in module_name :
187         command_module = importlib.import_module(module_name)
188     else:
189         command_module = importlib.import_module("command_scripts." + module_name)
190     if check == "true":
191         spoken_words = spoken_words.strip().split()
192         command_module.main(spoken_words)
193     else:
194         command_module.main()
195     return None
```

This function calls the main method method within the module passed into as a parameter. This is where control is transferred from command recognition to command execution.

Figure 3.5 Extract from command_recogniser.py

RECORDING USER

This is piece of code is an extract from the translate_speech.py file. The function of this piece of code is to record the users speech after they have activated the hotword. It uses PyAudio to read the matrix creators audio stream

```
def record_audio():
    audio = pyaudio.PyAudio()
    audio_stream = audio.open(format=pyaudio.paInt16, channels=2, rate=8000, input=True, frames_per_buffer=1024)
    return audio, audio_stream


def start_recording(audio_stream):
    # minimum length of recording
    min_time = 30
    time_counter = 0
    # level it must reach to start recording
    volume_threshold = 10
    data_container = []
    while time_counter < min_time:
        ## start taking in data
        data = audio_stream.read(1024, exception_on_overflow=False)
        data_chunk = array.array('h', data)
        sound_level = max(data_chunk)
        # threshold for sound
        print(sound_level)
        if sound_level > volume_threshold:
            data_container.append(data)
        time_counter += 1
    return audio_stream, data_container
```

Figure 3.6 Extract from translate_speech.py

TRANSLATING USERS SPEECH TO TEXT

This is piece of code is another extract from the translate_speech.py file. The function of this piece of code is to convert the previously recorded audio into text. It uses Pythons speech recognition module and I have set it to use googles translation service. There were a number of different services available but my testing concluded google was the most appropriate for my implementation.

```
def start_translator():
    # stop any other recording
    subprocess.call(["/home/pi/2019-ca400-randlea2/src/stop_arecord.sh"])
    audio_file = "speech.wav"
    audio, audio_stream = record_audio()
    # get the audio
    new_audio_stream, data_container = start_recording(audio_stream)
    #  wite the audio to a file
    write_audio(audio_file, new_audio_stream, audio, data_container)
    # translate the audio file to  text
    recognizer = speech_recognition.Recognizer()
    with speech_recognition.AudioFile(audio_file) as source:
        try:
            audio = recognizer.record(source)
            text = recognizer.recognize_google(audio)
            print(text.lower())
            return text.lower()
        except:
            return ""
```

Figure 3.7 Extract from translate_speech.py

SPEAKING TO USER

This piece of codes function is to speak to the users. I had to implement this in a different format than usual. All of the voice synthesizers I tested that were available from python were too robontic sound or took too long to synthesize. I found a fast and human like sounding voice synthesizer called pico2wave but unfortunately it was only available for unix systems and didn't offer a python wrapper. I carried out research to find a third party wrapper for python but I could only find ones available for python 2. I then decided to design my own module to use pico2wave.

```
#!/bin/bash
# bash script to change leds color and run pico2wave voice synthesizer

pico2wave -w=speakText.wav "$*"
aplay speakText.wav
rm speakText.wav
```

Figure 3.8 Extract from speakText.sh

This is the bash script which generates a .wav file with the command line arguments passed to it, these would be the words wanting to be spoken. This .wav file is then played through aplay which is the built in sound player for the raspberry pi.

```
# module for speaking to user with speakText.sh
import subprocess
import sys
import speaking_leds


def speak_to_user(sentence, background=None):
    speaking_leds.speaking_leds()
    if background is True:
        subprocess.Popen(["/home/pi/2019-ca400-randlea2/src/speakText.sh", sentence])
    else:
        subprocess.call(["/home/pi/2019-ca400-randlea2/src/speakText.sh", sentence])
```

This is the python script to run the above bash script. It can be either run as a background process or normal process. I used the subprocess module to run it.

Figure 3.9 Extract from speak.py

```
import zmq
from matrix_io.proto.malos.v1 import driver_pb2,io_pb2

# connect to matrix socket
socket = zmq.Context().socket(zmq.PUSH)
socket.connect("tcp://127.0.0.1:20021")

def listening_leds():
    driver_config_proto = driver_pb2.DriverConfig()
    everloop_image = []
    for i in range(0, 35):
        led_value = io_pb2.LedValue()
        # set red brightness to 100
        led_value.blue = 100
        # append that led configuration to led container
        everloop_image.append(led_value)
    # put the image into driver
    driver_config_proto.image.led.extend(everloop_image)
    # send the configuration to the driver
    socket.send(driver_config_proto.SerializeToString())
    return None


if __name__ == "__main__":
    listening_leds()
```

Figure 3.9.1 Extract from listening_leds.py

This piece of codes function is to change the LED configuration on the matrix creator. It creates ZMQ messaging connection with the matrix creator LED port 20021 over the localhost. After this the listening_leds function is called. This function creates a drive and a list to keep all the LED values. After this the function loops from 0 to 35 as this is the number of LED's on the matrix creator. These are all set toblue and with a brightness of 100. Then they are appended to to the LED values list. After the function has looped for 35 times, the full LED configuration is created. Finally the LED values list is put in a driver and sent to the matrix creator using the ZMQ socket set up at the beginning of the code.

```
19  def generate_wave(code):
20      pigpio_connection.set_mode(ir_emitter_gpio, pigpio.OUTPUT)
21      pigpio_connection.wave_add_new()
22      start_time = time.time()
23      marks = {}
24      wave = [0 for i in range(0,len(code))]
25      for i in range(0, len(code)):
26          current_code = code[i]
27          # check if a space, led off
28          if i % 2 != 0:
29              # delay for current code time
30              pigpio_connection.wave_add_generic([pigpio.pulse(0, 0, current_code)])
31              wave[i] = pigpio_connection.wave_create()
32          # else marks, led on
33          else:
34              mark_signal = generate_mark(current_code)
35              pigpio_connection.wave_add_generic(mark_signal)
36              marks[current_code] = pigpio_connection.wave_create()
37      # stop signal
38      wave[i] = marks[current_code]
39      time_difference = start_time - time.time()
40      if time_difference > 0.0:
41          time.sleep(time_difference)
42      return wave
```

Figure 3.9.2 Extract from send_ir_signal.py

This piece of codes function is to generate a chain of wave forms from a given code. The code firstly starts off by connecting to GPIO pin 13 as this is the matrix creators infrared transmitter pin. The for loop iterates through each element of the code. Within in the for loop, it checks whether the code is a space or mark. A mark means the Infrared LED is on while a space means the LED is off. If the the code is a space a wave will be created using pigpio's wave_add_generic function to create a wave with the led off for the same amount of time as the space. Another wave is then created afterthis and stored in the wave dictionary. This dictionary is used to store all the waves we've created. If the code is a mark i.e infrared LED is on, it will call the generate_mark function which will create a wave form with the infrared LED on for the specifced time. This is alalso saved in the wave dictionary. After this a check is made to calculate how much time has elapsed between the start and end of the function, if it's greater than 0 seconds the function will sleep for the time difference. This is needed to ensure all wave forms have been fully created. The dictionary containing all the waves is then returned.

PROBLEMS & RESOLUTION

HOTWORD DETECTOR

I encountered numerous problems when implementing my hotword/ wake word detector. The first of these problems was choosing the best quality hotword detector. After I finished researching each option, I decided that the only reliable one that would fulfill my requirements was Snowboy hotword detector as it allowed non user specific hotword detection and was also lightweight which was important with the small memory on the Raspberry Pi. When I starterd implementing it I ran into some real issues. Firstly it wouldn't detect my microphone, this was fixed by configuring the ".asoundrc" file within the Raspberry Pi as shown in figure 4.1.

```
pcm.!default {
  type asym
   playback.pcm {
     type plug
     slave.pcm "hw:2,0"
   }
   capture.pcm {
     type plug
     slave.pcm "hw:3,0"
```

Figure 4.1 Extract from .asoundrc

This was a temporary fix as it wouldn't stay configured correctly after rebooting. After I carried out more research I found that this was a problem with the matrix creator device. Instead of using PyAudio to record as the standard Snowboy script. I would instead need to use a script which was created to record using the linux command line. This script is called snowboydecorder_arecord.py and can be found in the src directory. This is a script freely available online from the Snowboy repository which I take no credit for. This script worked great but I was getting a recording error after multiple detection's. The error generated is shown in figure 4.2.

```
Exception in thread Thread-5:
Traceback (most recent call last):
  File "/usr/lib/python3.5/threading.py", line 914, in _bootstrap_inner
    self.run()
  File "/usr/lib/python3.5/threading.py", line 862, in run
    self._target(*self._args, **self._kwargs)
  File "/home/pi/2019-ca400-randlea2/src/snowboydecoder_arecord.py", line 100, in record_proc
    wav = wave.open(process.stdout, 'rb')
  File "/usr/lib/python3.5/wave.py", line 499, in open
    return Wave_read(f)
  File "/usr/lib/python3.5/wave.py", line 163, in __init__
    self.initfp(f)
  File "/usr/lib/python3.5/wave.py", line 128, in initfp
    self._file = Chunk(file, bigendian = 0)
  File "/usr/lib/python3.5/chunk.py", line 63, in __init__
    raise EOFError
EOFError
```

Figure 4.2 Error code

I couldn't understand for a while why this error was being generated. I researched for days and while I saw that it was a common problem, I still found no crediable aswers. It finally occurred that since it was a chunk error it could be related to the recording stream taking in chunks from a previous recording stream that was not of the same size. I fixed this by creating a bash script called which is shown in figure 4.3. The function of this script is to kill any recording processes which are currently runing. This script worked great and prevented recording errors after this.

```
#!/bin/bash


# if arecord is running stop it
# prevents chunk error from previous arecord input
if pgrep -x "arecord" > /dev/null
then
    sudo killall arecord
    echo "arecord stopped"
fi
```

Figure 4.3 Extract from stop_arecord.sh

SPEECH RECORDING

Writing a script which to record the users speech correctly was quite difficult. I had to ensure the script only recorded the users speech and not any unwanted noises. A snippet from my first attempt is shown in figure 4.4. The script records the users speech once the sound level reaches a certain threshold. The problem with this was the environment in which the device was in. If the device the was in a noisy environment it would start recoding when the sound level reaches that threshold despite a person not speaking to the device.

```
    volume_threshold = 700
    ## has a word been deteced
    words_spoken = False
    data_container = []
    ## while silence is not greater than 1 second
    while silent_counter < max_silence:
        ## start taking in data
        data = audio_stream.read(1024, exception_on_overflow=False)
        data_chunk = array.array('h', data)
        sound_level = max(data_chunk)
        ## threshold for silence
        print(sound_level)
        if sound_level > volume_threshold:
            words_spoken = True
            data_container.append(data)
        elif words_spoken is True:
            if len(data_container) > 10:
                silent_counter += 1
```

Figure 4.4 Extract from translate_speech.py

Fixing this problem required a lot of trial and error. If I solely relyed on a timer and recorded everything the translation quality was decreased drastically because of any background noise. I fixed this by combing both a timer and a sound threshold. The device will record for a set time but will allow keep pieces of the recording which were over the threshold. This would eliminate any background noise which would not be loud enough to be considered speech. This improved the translation quality greatly. A snippet from this implementation is shown in figure 4.5.

```
22      time_counter = 0
23      ## level it must reach to start recording
24      volume_threshold = 10
25      data_container = []
26      # while silence is not greater than 1 second
27      while time_counter < min_time:
28          ## start taking in data
29          data = audio_stream.read(1024, exception_on_overflow=False)
30          data_chunk = array.array('h', data)
31          sound_level = max(data_chunk)
32          # threshold for sound
33          print(sound_level)
34          if sound_level > volume_threshold:
35              data_container.append(data)
36          time_counter += 1
```

Figure 4.5 Extract from translate_speech.py

SPEAKER SOUND

An unsual problem I had was with the speaker sound. When using python's pigpiod service to send infrared signal this would cause the speakers to stop playing any sound. I researched and found that the infrared wave sent disrupts the auxiliary port on the Raspberry Pi. I couldn't find any software possible solutions to this despite trying exhaustively. I fixed this by buying a usb sound card which is shown in figure 4.6.

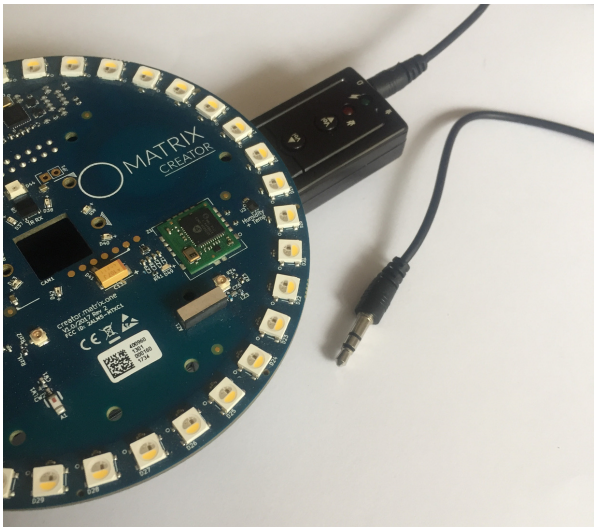


Figure 4.6 USB sound card

STOP PLAYING SOUND

Once the user says the hotword/ wake word I needed the device to stop playing any sound it currently was. This was harder than I thought and required some trial and error. I needed to find out if any sound was playing and stop playing it. The code for this is shown in figure 4.7.

```
#!/bin/bash


# stop speaking
if pgrep -x "aplay" > /dev/null
then
    killall aplay
fi

# pause playing music
if pgrep -x "mplayer" > /dev/null
then
   kill -STOP $(pgrep mplayer)
fi
```

Figure 4.7 Extract from suspend_sound_services.sh

I used the if statement shown above to check if mplayer or aplay were currently running, then kill them but I didn't want to completely kill mplayer as this is what I used to play music. I wanted to allow users to pause the music and resume the song when they wish. This was accomplished by finding the mplayer process id and instead of completely killing it, I used the kill -STOP command. I could then resume the song using a kill -CONT command. This would in turn make it possible to resume the song from the last position.

NUMEROUS MATRIX CREATORS ERRORS

The matrix creator is a device that can offfer a lot of functionality to a standard Raspberry Pi. Learning how to install matrix dependencies and use the resources available is more difficult than would be first anticipated.

Installing packages and configuring a device should be made as trouble free as possible but since matrix is a small componany they lack the ability to correctly maintain and develop code in my opinion. In my time developing with the matrix creator I encountered numerous problems. One of the most notable was the inability to install the matrix kernel modules after Raspbian had been updated to the latest version .

Solving this problem took many hours and a lot of work. I had to scroll through numerous error logs to find out what was causing the error. Solving this problem involved factory resetting the Raspberry Pi multiple times and reinstalling the kernel modules without success. The final solution involved formatting the SD card, writing an older version of Raspbian then installing this version without being updated. After this I then had to install the kernel modules from source and compile them instead of using a package download method.

RESULTS

The testing I carried out on the system can be found in my System Test Document which can be found in the documentation directory. This document highlights the test cases I carried out and their results.

FUTURE WORK

The current system can handle certain interactions however it is unable to generalise new interactions. If I was to improve the system, I would use a device that has more processing power then the Raspberry Pi as it struggles when a high load is placed on it.

I would also implement a Neural Network within the system and incorporate deep learning to train the system to think more like a person. This could enable the system to generalise new tasks without being specifically told what to do and when to do it.

In my initial design I wanted to make the system fully wireless however I was unable to accomplish this due to a number of constraints. Although I knew it would be a difficult task, I didn't realise to what extent and was unfortunately being slightly too optimistic. I found that the parts were difficult to source and also required some soldering which I had no experience with. These parts also lacked documentation and I felt the risk outweighed the reward however I feel this would most certainly be an interesting feature to incorporate in the future.