

E5IoT – radioT – an IoT Alarm Clock Radio

TEAM

#1 Stud. Nr: 201507096

Navn: Jacob Pedersen



AARHUS UNIVERSITET

Jacob Bechmann Pedersen

Contents

Introduction	3
Project Description	3
Requirements Analysis	3
Ubiquitous	4
Event-driven	4
State-driven	4
Option	4
Unwanted Behavior	4
System Design.....	5
Behavioral Design	5
Interface Design.....	6
Hardware Considerations	7
Software Considerations	9
Music Streaming	9
Local Audio	10
Display	10
Buttons	11
Calendar API	11
Weather API	11
Voice API.....	11
Finite State Machine.....	12
Implementation	13
Finite State Machine.....	13
Streaming Audio	14
Local Audio	14
Google Calendar API.....	16
Test and Verification.....	19
Internet Connection	19
Time Keeping	19
Finite State Machine.....	20
Google Calendar API.....	21
Audio Output	22
Audio Input	22

Verified Requirements..... 23

Discussion 24

Conclusion 24

References 25

Introduction

There are few things in life worse than waking up to an annoying alarm clock, with the same repeating pattern every morning. It gets mundane, frustrating, and most likely unbearable after being startled several mornings consecutively.

Project Description

How about waking up in a more elegant manner? With a smile on your lips in the morning? How is this achieved? This project will attempt to create an IoT product that solves the aforementioned issues. The project is done as part of the Internet of Things course at Aarhus University Herning.

The following challenges present themselves:

- How will the user be awakened in a pleasant manner?
- How will the user be awakened in time?
- What other ways can the morning routine be helped?

To make the solution a proper IoT product, and in line with the requirements of the IoT course at Aarhus University, the product will need to contain extra functionality that a non-IoT device would not. In addition to this, the original use cases must be kept intact, and a “graceful degradation” must be implemented, so that the system still offers value to the user. Even without an internet connection.

The proposed solution to these questions will come in the form of a little “radio alarm clock”-like device, that will play some music from the user’s library as the alarm (Either through streaming, or local library), and will fade in the tracks, to wake the user pleasantly. This could perhaps even be coupled with therapy lighting!

The Device has a cloud functionality, that pairs up with a cloud hosted Calendar, that has the user’s day-to-day alarms gathered, and uses these to wake the user. In the case these services are not available, a preprogrammed alarm time can be relied upon.

The Device can help the morning routine of the user, by providing weather information, as well as news or other upcoming events from their calendar. If these services are not available, it will still be functional for every other purpose. It could even feature some Voice recognition software, but should be operable from buttons as well.

A suiting name for the device could be radIoT (a mixture of radio and IoT). Simple, but pretty funky nonetheless. This will be the working name going forward.

This project development will be loosely based around some of the activities in EUDP, which is the Project tool of choice at Aarhus University Herning [1].

Requirements Analysis

In order to capture the most important requirements for the system-to-be, the EARS-requirement capture method, as suggested in EUDP, is used [2].

EARS includes the following classes of requirements: Ubiquitous, Event-Driven, State-driven, Option and Unwanted Behavior.

In addition to this, the requirements have been given a weighting; They either **shall** be fulfilled (Required for the system to be accepted), **will** be fulfilled (A future implementation is accepted), **must** be fulfilled (A strong wish, but no requirement), or be considered an improvement **“to be”** (A considered capability) [3].

For the system to be to be successful in fulfilling the criteria of the project description, the following requirements have been specified:

Ubiquitous

- The system **shall** be capable of telling time accurately, and with minimal drift.
- The system **shall** be capable of connecting to the internet.
- The system **shall** wake up the user, at the specified times.
- The alarm **will** be pleasant to awaken to.
- Voice commands using online voice recognition is **to be** implemented.

Event-driven

- When a specified time arrives, the system **shall** start the alarm.
- When the alarm starts playing, the system **will** slowly ramp up the sound.
- When the user has awoken, a weather message **will** be displayed.
 - When unavailable, the system **will** display a “weather unavailable” message.

State-driven

- While connected to an online calendar, the system **shall** use the “wake-up” times specified therein.
- While the calendar is unavailable, the system **shall** use hardcoded “wake-up” times.
 - If possible, the system **must** store the latest calendar updates to use instead.
- While connected to an online playlist service, the system **must** stream songs from there as alarms.
- While the playlists are unavailable, on-device songs **shall** be played.

Option

- Where possible, tough calculations **will** be performed off-device.
- Where user input is available, the input **must** be possible through button interface at least.
 - A microphone as interface is **to be** implemented if possible.

Unwanted Behavior

- If system cannot access information while online, a debug message is **to be** implemented.

-

Furthermore, some of the requirements from the formal project description available from Aarhus University carry over as well, these consist of:

Functional requirements

- 1 The device must be able to connect to the internet
 - 1.1 Internet connection shall be via WIFI
 - 1.2. The device should preferably be able to connect to AU's "AU Gadget network"
- 2 Your device must be able to read data from a connected sensor, local to the device
 - 2.1 a sensor can be anything that quantifies a physical measure, into an electrical signal, such as temperature, light, humidity, presence, movement, magnetism, pollution, etc.
- 3 Your device must be able to control an actuator
 - 3.1 An actuator can be anything that translates an electrical signal into a physical quantity, such as, motors, servos, valves, heaters, displays, lamps, etc.
- 4 Your device must be capable of using data from a web service, to augment "what it does", this could be weather data, traffic data, stock prices, twitter feeds, emails, rss-feeds or something different.
- 5 Your software and hardware design must be shared
 - 5.1 You must create a public **github** account, and add relevant project files here
 - 5.2 Hardware documentation, schematics, datasheets and pcb layouts are to be uploaded in pdf format
 - 5.3 Software files are to be uploaded in raw source code format, e.g. **.C, CPP, .h, .py**, etc.

Technical requirements

- 1 The technical platform can be a suited embedded platform of your choice, e.g. the Particle Photon, an ESP8266, a raspberry pi, beagle bone black or similar.
 - 1.1. The platform shall have Wifi connectivity
 - 1.2. The platform shall have available digital or analog I/O con connect sensors and actuators

Table 1 - Project requirements from Aarhus University Herning.

The project specific requirements, as well as the broader course-defined project requirements will both be considered and implemented. As the course-defined requirements naturally arise from the projects requirements anyhow, the implementation of some will likely entail others as well.

System Design

Behavioral Design

From the requirements set in the requirements analysis, some considerations towards the design emerge. For example; the System should have a default state that does not perform any tasks, unless the user asks it to. In addition to this the system should also switch into another state which plays songs when the alarm is triggered.

Based on these requirements, a simple Finite State Machine has been designed to help in implementing these states in the system. A triggering action has been defined for each transition between states. With some actions changing from several states into the next. The FSM diagram can be seen in Figure 1.

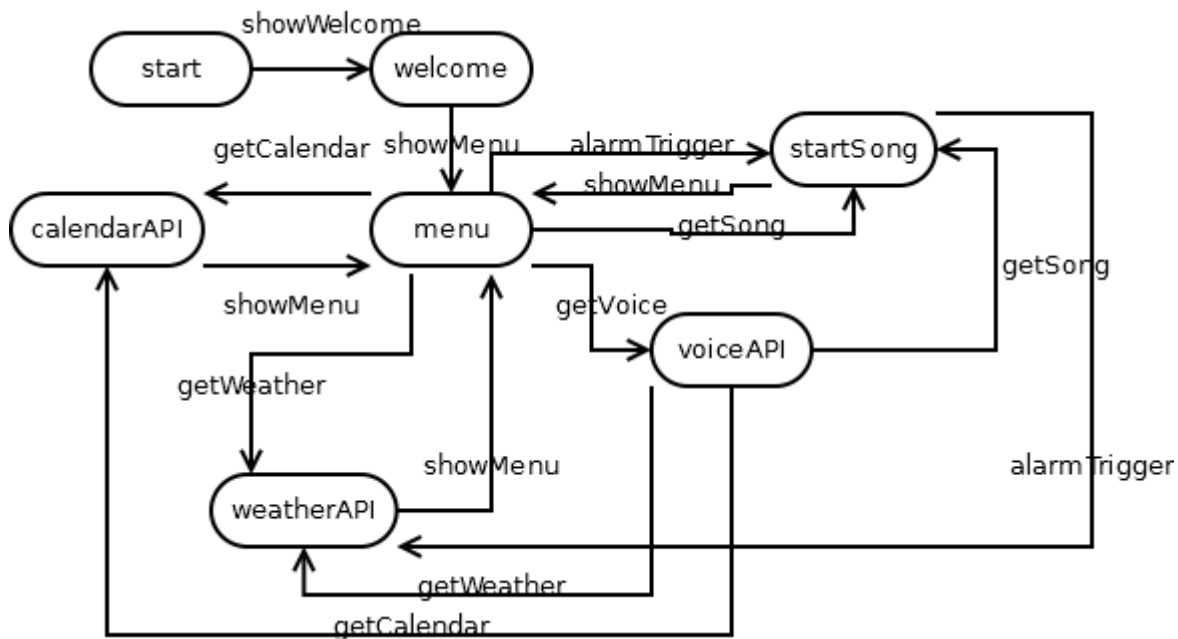


Figure 1 - FSM diagram of the system to be.

In the FSM diagram, every considered API is represented by a state that symbolizes the system making the necessary calls to that specific API, and then returning to the main idle state “menu”. Although only a “to be” implemented feature, the “voiceAPI” is included as a state to visualize its special state transitions. As it can transition between states like “menu”, due to its ability to make calls to the other states from the parsed voice input.

Interface Design

In broad strokes, the system will need to consist of the following hardware/software and services:

- A central processor, that does the simple logic – time etc.
- A media player, that will play the alarm media/songs when the time comes.
 - An amplifier to drive a speaker.
 - And the speaker itself.
- A calendar API to hold the “wake-up” times, and serve these to the logic.
- A weather API, to serve weather info to the logic.
- A display to display info, like the time, weather, etc.
- Some inputs to control the system, and request services.
 - Buttons to access manual controls in a menu.
 - Potentially a microphone to ask for services.
- Potentially a Voice recognition API to parse spoken controls.

The interfaces between these system components can be seen from Figure 2, where the connections are illustrated as simple inputs and outputs. The interactions between these interfaces will be explored more thoroughly, when specific possible solutions are found.

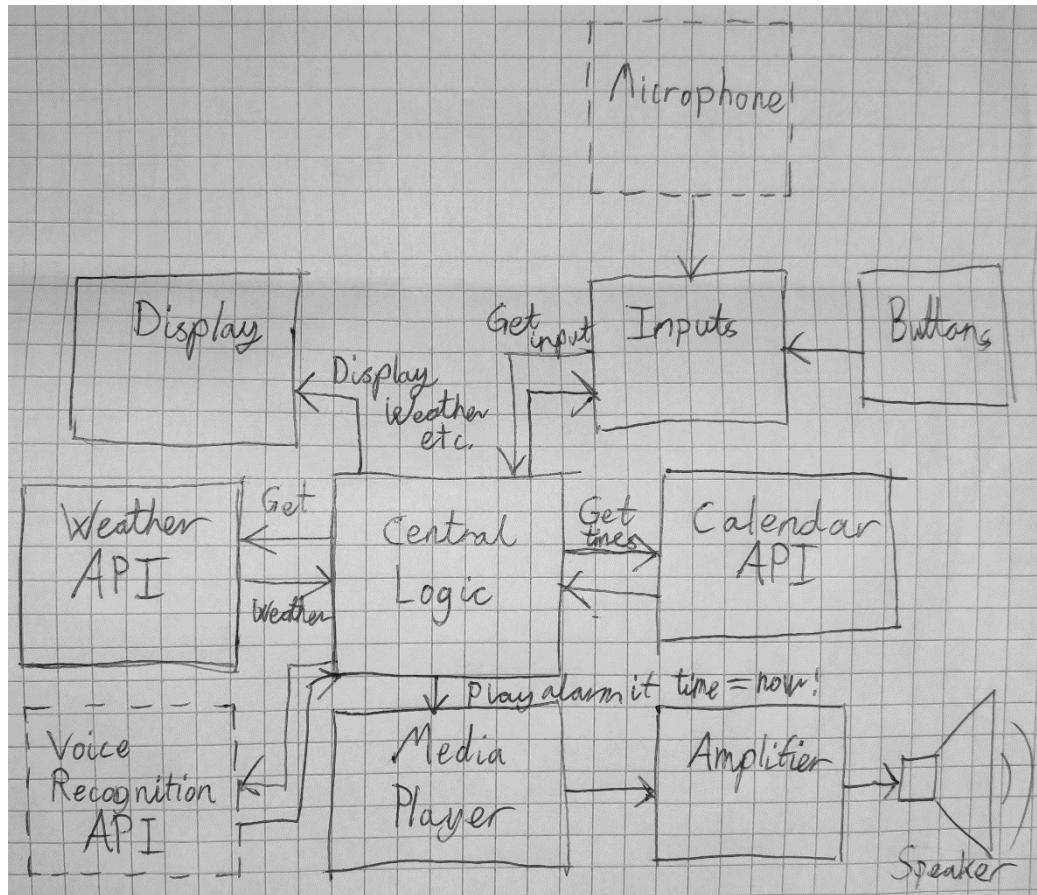


Figure 2 - A simple block diagram illustrating the construction of the system-to-be.

Among all these separate blocks, a lot of options are available to fulfill their individual requirements. For example, the central logic could be done in several ways depending on the platform:

- The logic could be written in C, and be a process running in a loop. Checking on webhooks online
 - This is a difficult approach, as C is a relatively low level language for this task.
- The logic could be run in a scripting language:
 - Bash scripts can handle logic, and is higher level than C.
 - Python Scripts run at a very high level, and have many external modules for several functions.
 - Node.js can run javascript on an embedded device.

The specific options for the implementation will be explored in the following chapters. Hardware Considerations and Software Considerations.

Hardware Considerations

One thing is abundantly clear. Since the device needs to access both WiFi, web API's and play music as shown in the Requirements Analysis, it would make the most sense to have an operating system on it with web stacks and audio drivers. There are only a few platfoms available for this purpose, among others are:

- BeagleBone Black
 - Embedded Linux device, with WiFi, TCP/IP-stack, scripting and file system.
 - No directly accessible Audio, difficult to set up for this purpose.

- Raspberry Pi 3
 - Embedded Linux Device with WiFi, TCP/IP-stack, scripting and file system.
 - Includes multimedia drivers, and has several community support for audio, along with a TRS output.
- Raspberry Pi Zero W
 - Embedded Linux device with WiFi, TCP/IP-Stack, scripting and file system.
 - Includes multimedia drivers, and has several community supplied overlays for audio, which support external DACs
 - Cheap, has small form factor.

Few other candidates apply for a project of this shape. As such, it seems most logical to pick the cheapest and most capable of the devices. From these three listed platforms, Raspberry Pi Zero will be the platform of choice.

Since the system to be is an IoT device that must be on for an extended amount of time, power is also a significant factor. Many of the embedded Linux boards tend to use quite a bit of power, due to their expensive processors and peripherals. A table of these consumptions per platform can be seen on Table 2 and Table 3.

MODE	USB	DC	DC+USB
Reset	TBD	TBD	TBD
Idling @ UBoot	210	210	210
Kernel Booting (Peak)	460	460	460
Kernel Idling	350	350	350
Kernel Idling Display Blank	280	280	280
Loading a Webpage	430	430	430

Table 2 - BeagleBone Black power consumption, taken from the BeagleBoard Wiki [4].

		Pi1 (B+)	Pi2 B	Pi3 B (amps)	Zero (amps)
Boot	Max	0.26	0.40	0.75	0.20
	Avg	0.22	0.22	0.35	0.15
Idle	Avg	0.20	0.22	0.30	0.10
Video playback (H.264)	Max	0.30	0.36	0.55	0.23
	Avg	0.22	0.28	0.33	0.16
Stress	Max	0.35	0.82	1.34	0.35
	Avg	0.32	0.75	0.85	0.23

Table 3 - Raspberry Pi platform power consumption. Taken from the Raspberry Pi FAQs [5]. Note that Zero's draw is without the WiFi and bluetooth module in Zero W.

These conditions considered, the Raspberry Pi Zero W still seems like a prime candidate given the info on Table 4 as well.

Product	Recommended PSU current capacity	Maximum total USB peripheral current draw	Typical bare-board active current consumption
Raspberry Pi Zero W	1.2A	Limited by PSU, board, and connector ratings only.	150mA

Table 4 - Raspberry Pi Zero W bareboard active consumption.

An unconsidered scenario from these tables is the power consumption of the Raspberry Pi Zero W while using a WiFi connection. It could be argued, that this scenario only arises whenever actually handling data, and optimizations to the software routine, could help alleviate this problem.

Software Considerations

To create the different blocks in the software, a lot of predefined, or robust software modules or libraries are available.

To determine which combinations of programming languages are suited for work on this project, a list of possible languages, and their respective qualifications for the task at hand is in order.

Since the project depends mainly upon services that are available online from suppliers of different kinds, there are a few candidates for each block:

Music Streaming

Based upon an extensive list by Jeff Dunn on Business Insider [6], the most widely used music platform for streaming is Pandora, which carries around 32% of all listeners in America as shown in Figure 3. While this is an impressive statistic, it carries the unfortunate caveat that this only applies to the United States.

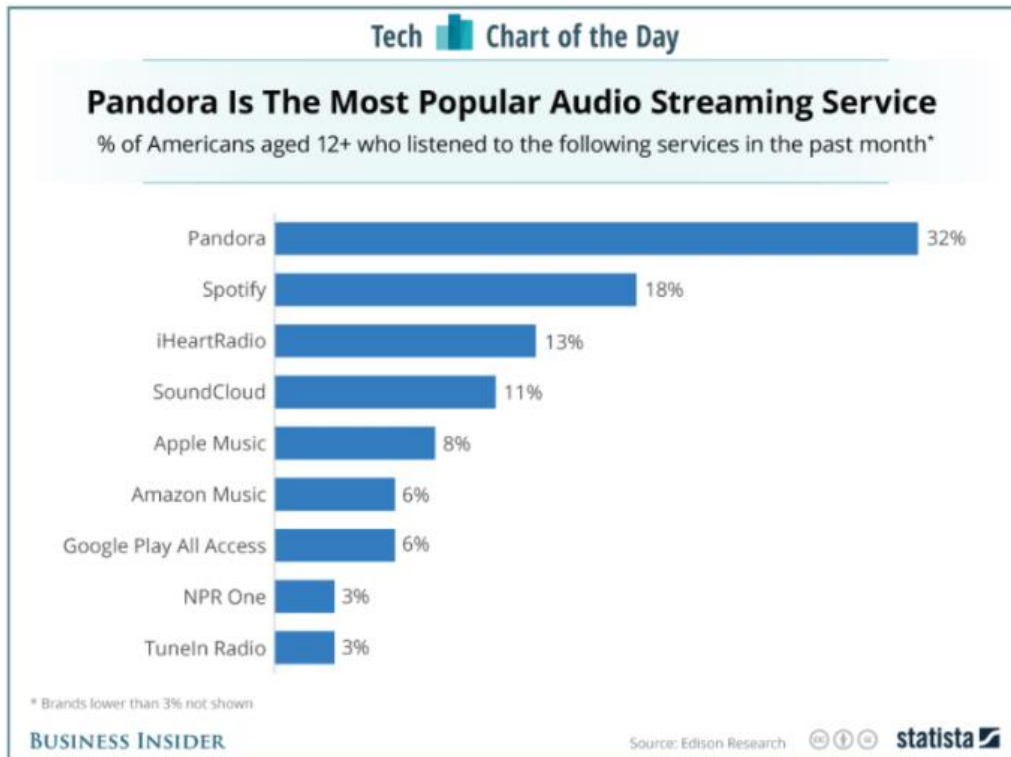


Figure 3 - Market shares divided between the music streaming services in America.

A more recent article by Reuters published in Fortune points this out as well, and also lists Spotify as the biggest player on the worldwide market, with over 140 million active users [7].

It just so happens, that Spotify does not have a Node.js implemented API. It does, however, have an extensive software library for playing tracks and fetching data. This libspotify is supported by a python wrapper, called pyspotify [8]. Since this is the only possible implementation available, it seems that Python scripts for the system is the way to go.

Local Audio

The local audio playback can be handled in several different ways. For example, the Raspberry Pi's Raspbian distribution (And many other Linux distros as well) comes with the Advanced Linux Sound Architecture (ALSA), that contains several built-in functions, including some for playing and recording audio. These can be called from any scripting language that allows passing messages to the operating system.

Another option is the pyAudio [9], that wraps the pulseAudio libraries. In keeping with the rest of the Python API's, this seems like a decent option so that everything can be worked into a collection of Python Scripts.

Display

The display block can be made from the software modules provided by Adafruit, since they supply examples in Python [10], that even includes a wiring diagram for their cobbler GPIO breakout board:

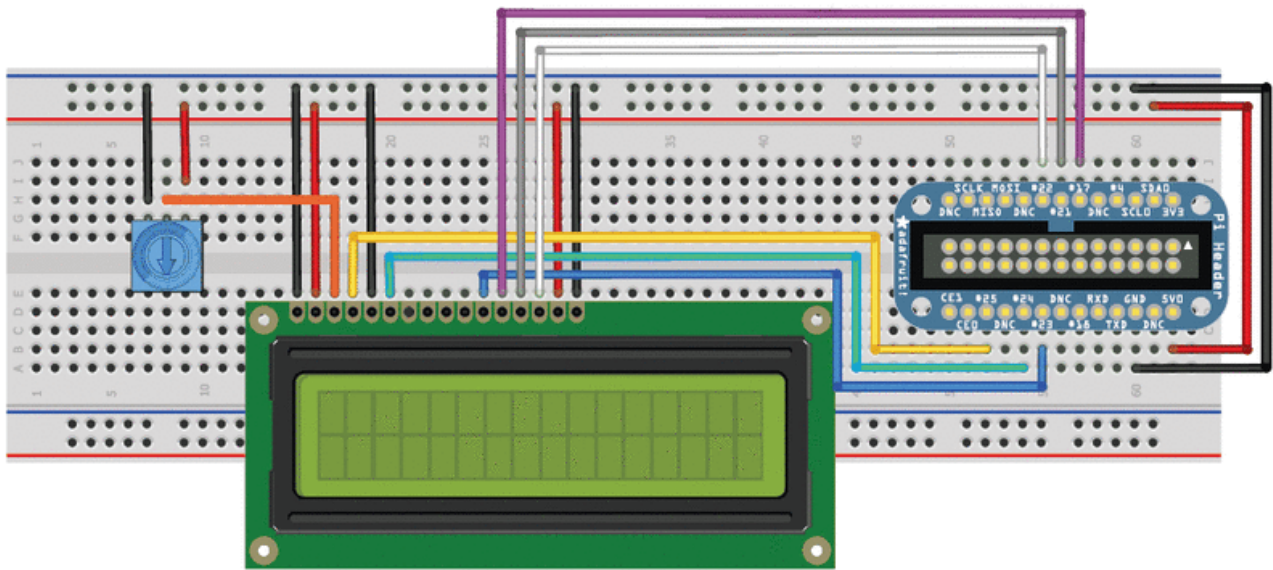


Figure 4 - LCD connected to an Adafruit Cobbler Board.

The wiring itself is perfectly fine, and can interact with several different libraries. Other implementations exist, either through pre-compiled C-based ones [11], or Node.js implemented ones [12].

Buttons

There are several candidates for implementing button I/O on a Raspberry Pi. The first approach is to read the button GPIOs as inputs on their files, in the Raspbian file system.

However, some more elegant solutions have been developed. For example, the wiringPi library [13] for C/C++ by Gordon Henderson and his contributors. There is also the RPi.GPIO module [14] for Python by Ben Cronston available to keep going out the Python route of libraries.

Calendar API

As for picking a calendar API, it did not seem like there was much of a choice. Seeing as a majority of Smartphones support Google Calendar, and Google have plenty of API support for scripting, and non-scripting languages, this was a no-brainer. As with any other API, one must sign up and get a Verification Key, but the more features can be provided by Google for this project, the easier development gets.

There are extensive tutorials and documentation available as well [15], which helps out a lot.

Weather API

OpenWeatherMap and Weather Underground seem to be two big players on the field of weather APIs. OpenWeatherMap however, wins on value for the free pricing range [16] [17]. This will be the API of choice. They both communicate out using JSON, XML and HTML, so regarding programming language, it does not seem to matter much.

Voice API

Since there is a significant advantage to using several of Google's API's for the same project, the obvious choice for a Voice processing API would be Google Assistants API [18]. Another possible candidate for this product could have been Amazon's Alexa [19]. Although that might probably be better suited for working on a product that implements more of Amazons other services. It is however worth mentioning that AWS

provides several of the needed services for voice recognition and big data analysis, if one wanted to implement these in their projects.

Finite State Machine

Finally, in order to bind it all together as shown in Figure 1, since so many of these API's are available for Python it has been decided to use a Python module, Fysom [20], to handle the Finite State Machine, that will control the flow of the system. This doesn't necessarily entail that everything else has to be written in Python, however for consistency's sake, the decision has been made to write the rest in Python as well.

Implementation

To verify that all of the functionality is even possible within the system, a good place to start is to see whether the specific software libraries can be implemented in a working manner, before further implementing them in the system.

Finite State Machine

As the specific API's are implemented, they will be worked into the FSM in a procedural manner, since the Finite State Machine easily implemented into Python as shown in Figure 1. The code resulting from using the Fysom module to generate a Finite State Machine can be seen in Figure 5. Here you specify the 'initial' state, and then the 'events' that transition the FSM from one state to the next. Each event has a 'name', 'src' and 'dst'. These are the names of the event itself, and the two states it transitions between.

```

1  #!/usr/bin/python
2
3  # For time delays and sleeping
4  import time
5
6  # For FSM
7  from fysom import Fysom
8
9  global fsm
10 global alarmFlag
11
12 fsm = Fysom({
13     'initial': 'start',
14     'events': [
15         {'name': 'showWelcome', 'src': 'start', 'dst': 'welcome'},
16         {'name': 'showMenu', 'src': ['welcome','startSong','weatherAPI','calendarAPI'], 'dst': 'menu'},
17         {'name': 'alarmTrigger', 'src': 'menu', 'dst': 'startSong'},
18         {'name': 'alarmTrigger', 'src': 'startSong', 'dst': 'weatherAPI'},
19         {'name': 'getCalendar', 'src': 'menu', 'dst': 'calendarAPI'},
20         {'name': 'getWeather', 'src': 'menu', 'dst': 'weatherAPI'},
21         {'name': 'getSong', 'src': 'menu', 'dst': 'startSong'},
22         {'name': 'getCalendar', 'src': 'voiceAPI', 'dst': 'calendarAPI'},
23         {'name': 'getWeather', 'src': 'voiceAPI', 'dst': 'weatherAPI'},
24         {'name': 'getSong', 'src': 'voiceAPI', 'dst': 'startSong'}
25     ]
26 })

```

Figure 5 - An FSM implemented in Python using Fysom. This state machine covers all of the possible states and transitions in a simple and understandable manner.

On Figure 6, the FSM is enveloped in a while True loop that goes on forever. If the fsm.isstate() evaluation is true, the code within is executed. This applies for every state in the machine. When the time comes to switch states, every event has a function associated with it, to make the proper transition between states, as mentioned in the model.

```

while True:
    if fsm.isstate('start'):
        print(fsm.current)
        time.sleep(1)
        fsm.showWelcome()

```

Figure 6 - How FSM's are worked in practice.

Streaming Audio

The main functions in the pySpotify library have been implemented in a module, that contains functions for logging in, finding a playlist, and playing a random song from there. It can be seen in Figure 7.

```

1  #!/usr/bin/python
2
3  # Import Spotify API
4  import spotify # Imports Spotify methods.
5  import threading # Imports Threading methods.
6
7  # Initialize Spotify session, and sinks for audio.
8  logged_in_event = threading.Event() # Tracks when the user is logged in.
9  session = spotify.Session() # Spotify Session object.
10 audio = spotify.AlsaSink(session) # Audio object for output - sinks to alsa.
11 loop = spotify.EventLoop(session) # Event loop for the session.
12
13 # A login state tracker thread.
14 def spotifyConnectionState(session):
15     if session.connection.state is spotify.ConnectionState.LOGGED_IN:
16         logged_in_event.set()
17
18 # Logs into Spotify
19 def spotifyLogin():
20     loop.start() # Starts the spotify session loop.
21     session.on(
22         spotify.SessionEvent.CONNECTION_STATE_UPDATED,
23         connection_state_listener)
24     session.login('1217631550', 's3cretpassword')
25     logged_in_event.wait()
26
27 # Plays random Spotify songs from playlist. Query format = u'Discover Weekly'
28 def spotifyPlayRandom(playlistQuery):
29     # Load the playlist from search query.
30     playlist = spotify.search(session, query=playlistQuery).load(timeout=60) # Search with timeout of 60s.
31     tracks = playlist.tracks()
32     playlist.load(60) # Timeout the loading in 60s.
33     playTrack = random.randint(1,len(tracks)) # Randomly pick the track to be played.
34
35     # Play the specific Track:
36     track = session.get_track(playTrack)
37     track.load()
38     session.player.load(track)
39     session.player.play()

```

Figure 7 - Functions wrapping the pySpotify module.

However, the module breaks as soon as it tries to create the session variable. The reason is a missing AppKey. As it turns out, even though several other software packages rely upon this library, Spotify discontinued it in 2015 for unknown reasons. Although developers can still be lucky enough to get an AppKey through a Google Form, the library that pySpotify wraps is considered deprecated [21]. They may eventually release an embedded library for Linux, but until then this solution is unfortunately off the table.

Local Audio

Whether the streaming solution works or not, the local song player will need to be implemented, to ensure graceful degradation of the system. Using the pyAudio [9] module, the “startSong” state is implemented as shown in Figure 8, to begin with using only one song. In the future, a random selection of songs in the provided directory will be implemented. For now, this implementation suits the needs of the system.

```

100 elif fsm.isstate('startSong'):
101     wf = wave.open(SOUNDS_PATH + 'Daytona_USA_Theme_.wav', 'rb')
102     p = pyaudio.PyAudio()
103     stream = p.open(format=p.get_format_from_width(wf.getsampwidth()),
104                     channels=wf.getnchannels(),
105                     rate=wf.getframerate(),
106                     output=True,
107                     stream_callback=callback)
108     stream.start_stream()
109     fsm.showMenu()

```

Figure 8 - "startSong" state in the FSM. This state starts the song stream.

The interface to the physical actuator is secured through the use of the ALSA drivers with the settings shown in Figure 9, and the pHAT DAC [22] for Raspberry Pi Zero W. These settings also allow for recording with a microphone that could also be used as an input source for the system.

```

GNU nano 2.7.4
pcm.!default{
    type asym
    capture.pcm "mic"
    playback.pcm "speaker"
}
pcm.mic{
    type plug
    slave{
        pcm "hw:1,0"
    }
}
pcm.speaker{
    type plug
    slave{
        pcm "hw:0,0"
    }
}

```

Figure 9 - ~/.asoundrc configuration file for the ALSA drivers of the pHAT DAC.

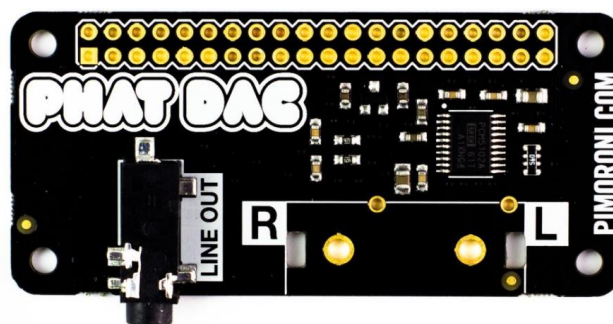


Figure 10 - pHAT DAC

In order to not waste any time on configuring an amplifier and speaker pair, the best solution is to use an active speaker directly together with the pHAT DAC, so that the implementation is the smoothest possible. The hardware configuration can be seen in Figure 11:



Figure 11 - Raspberry Pi Zero W, connected to a pHAT DAC and an active speaker.

Google Calendar API

As mentioned earlier, Google has extensive documentation and tutorials for their API's. With a lot of inspiration pulled from their Quickstart example [15], a tiny Python module script was created, a piece of which can be seen on Figure 12:

```
49 def getCalPosts():
50     credentials = getCalCredentials() # Gets the credentials if they exist.
51     http = credentials.authorize(httplib2.Http()) # Authorizes the credentials using httplib2.
52     service = discovery.build('calendar', 'v3', http=http) # Defines the service, that provides the calendar dates.
53
54     # Get the upcoming calendar dates for the next 10 days:
55     # Define "now".
56     now = datetime.datetime.utcnow()
57     span = datetime.timedelta(days=10)
58     end = now + span
59     end = end.isoformat() + 'Z'
60     now = now.isoformat() + 'Z'
61     # Write out to LCD:
62     eventsResult = service.events().list(
63         calendarId='primary', timeMin=now, timeMax=end, singleEvents=True,
64         orderBy='startTime').execute()
65     events = eventsResult.get('items', [])
66     return events
```

Figure 12 - getCalPosts function. Uses the credentials stored on the disk after authorization.

The purpose of the getCalPosts() function is to be called within the "calendarAPI" state, and block the script with calls to the Google Calendar API. Most of the actual data fetching, and preprocessing is done within the wrappers of the Python Module, which is very convenient. As can be told from the return value, the Calendar events themselves are returned directly, for further processing by the calling

radIoT_mainScript.py as seen in Figure 13. The event data is stored in a global variable, which can be used to trigger the alarm.

```

74 elif fsm.isstate('calendarAPI'):
75     lcd.clear()
76     lcd.message('Getting Google\nCalendar info...')
77     events = getCalPosts()
78     # Check if anything was received:
79     if not events:
80         # clear LCD, and write that nothing was found.
81         lcd.clear()
82         lcd.message('Nothing found...')
83     for event in events:
84         # Print out the start of the event:
85         start = event['start'].get('dateTime', event['start'].get('date')).encode('ascii','replace')
86         title = event['summary'].encode('ascii','replace')
87         # Clear LCD, and print the time and title of the events gotten:
88         lcd.clear()
89         lcd.message(start + '\n' + title)
90         time.sleep(1)
91     lcd.clear()
92     lcd.message('Done!')
93     time.sleep(1)
94     fsm.showMenu()

```

Figure 13 - State "calendarAPI" calls the getCalPosts() function, saves the events in a global variable, and then processes the returned data.

In order to use the Google Calendar API, certain credentials are needed. Firstly, OAuth2 authentication Client secrets have to be made and transferred to the Raspberry Pi Zero W. This can of course not be done over Github, since the repository is public. For this reason, any non-public data will be stored on an external flash-drive mounted to /media/certs-n-sounds/. This both ensures that no one other than users themselves have access their client secrets and/or local music files. The flash drive is automounted in the manner shown on Figure 14 by editing the /etc/fstab file with the bottommost line shown. Now the client secrets file needed to certify the application can be found in /media/certs_n_sounds/certs/client_secret.json.

```

GNU nano 2.7.4
proc /proc proc defaults 0 0
PARTUUID=207e2dfe-01 /boot vfat defaults 0 2
PARTUUID=207e2dfe-02 / ext4 defaults,noatime 0 1
# a swapfile is not a swap partition, no line here
# use dphys-swapfile swap[on|off] for that
UUID=6602-CD6D /media/certs_n_sounds vfat auto,nofail,noatime,users,rw,uid=pi,gid=pi 0 0

```

Figure 14 - /etc/fstab to automount the /media/certs_n_sounds folder.

The function getCalCredentials() also included in the radIoT_googleCal.py module makes use of the placement of the drive, and will not be able to complete its first run without it.

```
18 # Setup client secrets and Application Name for Calendar.
19 SCOPES = 'https://www.googleapis.com/auth/calendar.readonly'
20 CLIENT_SECRET_PATH = '/media/certs_n_sounds/certs/'
21 CLIENT_SECRET_FILE = 'client_secret.json'
22 APPLICATION_NAME = 'radioIoT Alarm Clock'
23
24 def getCalCredentials():
25     home_dir = os.path.expanduser('~') # Sets the home directory to the user home directory.
26     credential_dir = os.path.join(home_dir, '.credentials') # Appends the credentials directory to the user directory.
27     # Checks if the path exists. If not, os method creates the dir.
28     if not os.path.exists(credential_dir):
29         os.makedirs(credential_dir)
30     # Adds the file to the credential path.
31     credential_path = os.path.join(credential_dir,
32                                   'radioIoT_Alarm_Clock.json')
33
34     store = Storage(credential_path) # Stores secret in the path.
35     credentials = store.get() # Retrieve credentials.
36     # If no credentials are found, or they are invalid:
37     if not credentials or credentials.invalid:
38         # Create the credentials from the Client Secret File.
39         flow = client.flow_from_clientsecrets(CLIENT_SECRET_PATH + CLIENT_SECRET_FILE, SCOPES)
40         flow.user_agent = APPLICATION_NAME
41         # Depending on the import of the argparse:
42         if flags:
43             credentials = tools.run_flow(flow, store, flags) # Store the credentials.
44         else: # Needed only for compatibility with Python 2.6
45             credentials = tools.run(flow, store)
46         print('Storing credentials to ' + credential_path)
47     return credentials
```

Figure 15 - getCalCredentials() function.

When the first run is made in headless mode, the text on Figure 16 appears. The hyperlink provided takes the external PC to a webpage that allows the user to accept the applications usage of their data, After which the verification code is entered in the terminal, and the certificate is made and saved.

```
Go to the following link in your browser:
https://accounts.google.com/o/oauth2/auth?scope=https%3A%2F%2Fwww.googleusercontent.com&access_type=offline
Enter verification code: _
```

Figure 16 - Verification process for using the Google Calendar API.

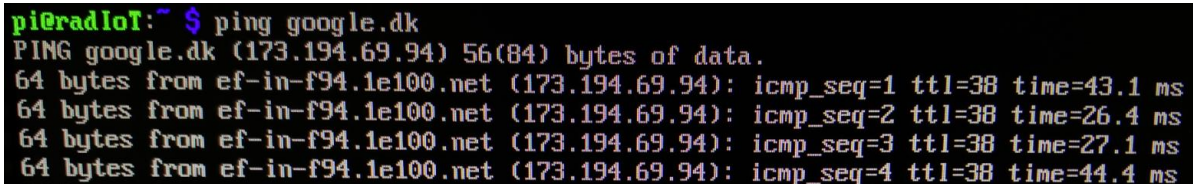
Test and Verification

Internet Connection

The following requirement can be verified:

- The system **shall** be capable of connecting to the internet.

By using the ping command, an internet connection can be verified easily, as shown in Figure 17:



```
pi@radloT:~$ ping google.dk
PING google.dk (173.194.69.94) 56(84) bytes of data.
64 bytes from ef-in-f94.1e100.net (173.194.69.94): icmp_seq=1 ttl=38 time=43.1 ms
64 bytes from ef-in-f94.1e100.net (173.194.69.94): icmp_seq=2 ttl=38 time=26.4 ms
64 bytes from ef-in-f94.1e100.net (173.194.69.94): icmp_seq=3 ttl=38 time=27.1 ms
64 bytes from ef-in-f94.1e100.net (173.194.69.94): icmp_seq=4 ttl=38 time=44.4 ms
```

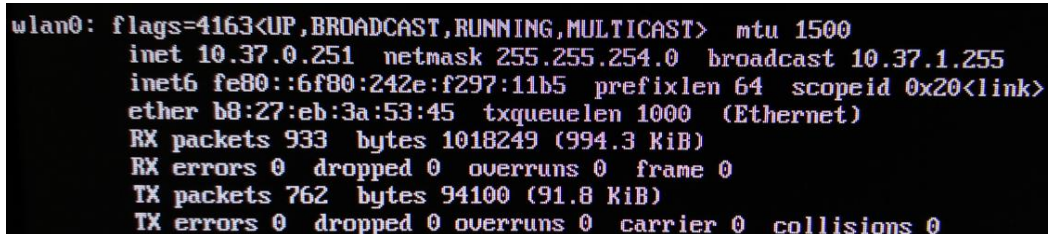
Figure 17 - Pinging google.dk.

In addition to this, the following course-required technical requirements, can be verified:

1 The technical platform can be a suited embedded platform of your choice, e.g. the Particle Photon, an ESP8266, a raspberry pi, beagle bone black or similar.

- 1.1. The platform shall have Wifi connectivity
- 1.2. The device should preferably be able to connect to AU's "AU Gadget network"

The connection method is WiFi as can be seen on Figure 18, that displays the wlan0 interface detailed in ifconfig.



```
wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.37.0.251 netmask 255.255.254.0 broadcast 10.37.1.255
    inet6 fe80::6f80:242e:f297:11b5 prefixlen 64 scopeid 0x20<link>
    ether b8:27:eb:3a:53:45 txqueuelen 1000 (Ethernet)
    RX packets 933 bytes 1018249 (994.3 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 762 bytes 94100 (91.8 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Figure 18 - ifconfig showing the wlan0 WiFi interface.

Time Keeping

- The system **shall** be capable of telling time accurately, and with minimal drift.

While there is always a certain amount of drift within especially computer systems, a lot of these unintended drifts can be mitigated by utilizing NTP (Network Time Protocol), which the Raspberry Pi Zero W supports. Verification that it is up and running on the device can be seen on Figure 19.

While NTP helps get the freshest time available, at a precision of up to 20ms. What happens when the device powers off and no network is enabled? For this a Realtime battery backed clock will be needed. However, as long as the Raspberry Pi is powered on, the timekeeping will not degrade past an unusable level.

```

pi@radioT:~$ systemctl status ntp
■ ntp.service - LSB: Start NTP daemon
   Loaded: loaded (/etc/init.d/ntp; generated; vendor preset: enabled)
   Active: active (running) since Wed 2017-12-13 00:52:36 CET; 56s ago
     Docs: man:systemd-sysv-generator(8)
    CGroup: /system.slice/ntp.service
            └─884 /usr/sbin/ntpd -p /var/run/ntpd.pid -g -u 110:113

Dec 13 00:52:36 radioT ntpd[884]: Listen and drop on 1 v4wildcard 0.0.0.0:123
Dec 13 00:52:36 radioT ntpd[884]: Listen normally on 2 lo 127.0.0.1:123
Dec 13 00:52:36 radioT ntpd[884]: Listen normally on 3 wlan0 10.37.0.251:123
Dec 13 00:52:36 radioT ntpd[884]: Listen normally on 4 lo [::]:123
Dec 13 00:52:36 radioT ntpd[884]: Listen normally on 5 wlan0 [fe80::6f80:242e:f297:11b5%3]:123
Dec 13 00:52:36 radioT ntpd[884]: Listening on routing socket on fd #22 for interface updates
Dec 13 00:52:37 radioT ntpd[884]: Soliciting pool server 212.99.225.86
Dec 13 00:52:38 radioT ntpd[884]: Soliciting pool server 86.48.99.37
Dec 13 00:52:39 radioT ntpd[884]: Soliciting pool server 217.63.114.22
Dec 13 00:52:40 radioT ntpd[884]: Soliciting pool server 78.156.100.202

```

Figure 19 - Status on the NTP daemon running on the Raspberry Pi Zero W.

Finite State Machine

A test of the Finite state machine held the logic for transitions shown in Figure 20 below:

```

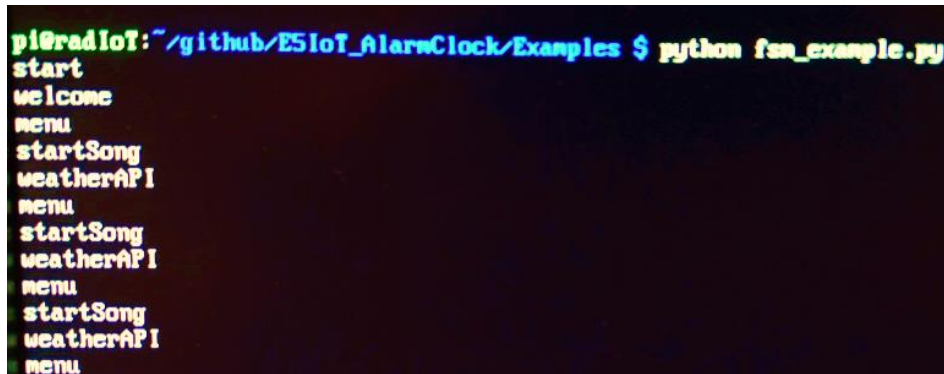
28 def main():
29     alarmFlag = False
30     while True:
31         if fsm.isstate('start'):
32             print(fsm.current)
33             time.sleep(1)
34             fsm.showWelcome()
35
36         elif fsm.isstate('welcome'):
37             print(fsm.current)
38             time.sleep(1)
39             fsm.showMenu()
40
41         elif fsm.isstate('menu'):
42             print(fsm.current)
43             time.sleep(1)
44             if alarmFlag == True:
45                 fsm.alarmTrigger()
46
47         elif fsm.isstate('calendarAPI'):
48             print(fsm.current)
49             time.sleep(1)
50             fsm.showMenu()
51
52         elif fsm.isstate('weatherAPI'):
53             print(fsm.current)
54             time.sleep(1)
55             fsm.showMenu()
56
57         elif fsm.isstate('startSong'):
58             print(fsm.current)
59             time.sleep(1)
60             if alarmFlag == True:
61                 alarmFlag = False
62                 fsm.alarmTrigger()
63             else:
64                 fsm.showMenu()
65
66         alarmFlag = True
67
68 if __name__ == '__main__':
69     main()

```

Figure 20 - Simple FSM test script. calendarAPI is never entered. Similarly, voiceAPI is not featured yet, as the implementation window is uncertain.

An alarmFlag is raised with each loop, to ensure that the “menu” state transitions into “startSong”. The expected output is this: start, welcome, menu, startSong, weatherAPI, menu, startSong, weatherAPI, etc.

The output is shown on Figure 21, and it quite clearly follows the expected pattern. With the FSM in place, the rest of the implementation can more easily be fit into a natural flow.



```
pi@rad101:~/github/ESIoT_AlarmClock/Examples $ python fsm_example.py
start
welcome
menu
startSong
weatherAPI
menu
startSong
weatherAPI
menu
startSong
weatherAPI
menu
```

Figure 21 - Output of the fsm_example.py script.

Google Calendar API

- While connected to an online calendar, the system **shall** use the “wake-up” times specified therein.

4 Your device must be capable of using data from a web service, to augment “what it does”, this could be weather data, traffic data, stock prices, twitter feeds, emails, rss-feeds or something different.

Using the LCD along with the call to getCalPosts() shown in Figure 13 within the “calendarAPI”-state, the API-methods return events, that are output to the LCD, in order for the user to verify that their calendar events were loaded properly onto the device as shown in Figure 22.



Figure 22 - Proper output of a loaded Calendar Event onto the LCD.

Audio Output

1.2. The platform shall have available digital or analog I/O con connect sensors and actuators

3 Your device must be able to control an actuator

3.1 An actuator can be anything that translates an electrical signal into a physical quantity, such as, motors, servos, valves, heaters, displays, lamps, etc.

- While the playlists are unavailable, on-device songs **shall** be played.

Running the script with the pyAudio methods and the song loaded on flash, produces the following output seen in Figure 23:

```
pi@radio: ~/github $ cd ESloT_AlarmClock/
pi@radio: ~/github/ESloT_AlarmClock $ cd Main_Script/
pi@radio: ~/github/ESloT_AlarmClock/Main_Script $ python radioT_mainScript.py
ALSA lib confmisc.c:1281:(snd_func_refer) Unable to find definition 'cards.HifiberryDac.pcm.front.0:CARD=0'
ALSA lib conf.c:4528:(_snd_config_evaluate) function snd_func_refer returned error: No such file or directory
ALSA lib conf.c:5007:(snd_config_expand) Evaluate error: No such file or directory
ALSA lib pcm.c:2495:(snd_pcm_open_noupdate) Unknown PCM front
ALSA lib pcm.c:2495:(snd_pcm_open_noupdate) Unknown PCM cards.pcm.rear
ALSA lib pcm.c:2495:(snd_pcm_open_noupdate) Unknown PCM cards.pcm.center_lfe
ALSA lib pcm.c:2495:(snd_pcm_open_noupdate) Unknown PCM cards.pcm.side
```

Figure 23 - Showing activity in the ALSA library.

So the songs play within the “startSong” state of the main script. As such, the actuator is in place to output an audio clip, whenever the user needs to wake up.

Audio Input

2 Your device must be able to read data from a connected sensor, local to the device

2.1 a sensor can be anything that quantifies a physical measure, into an electrical signal, such as temperature, light, humidity, presence, movement, magnetism, pollution, etc.

Using the arecord command, a recording can be made of the user speaking into a USB microphone as seen in Figure 24.

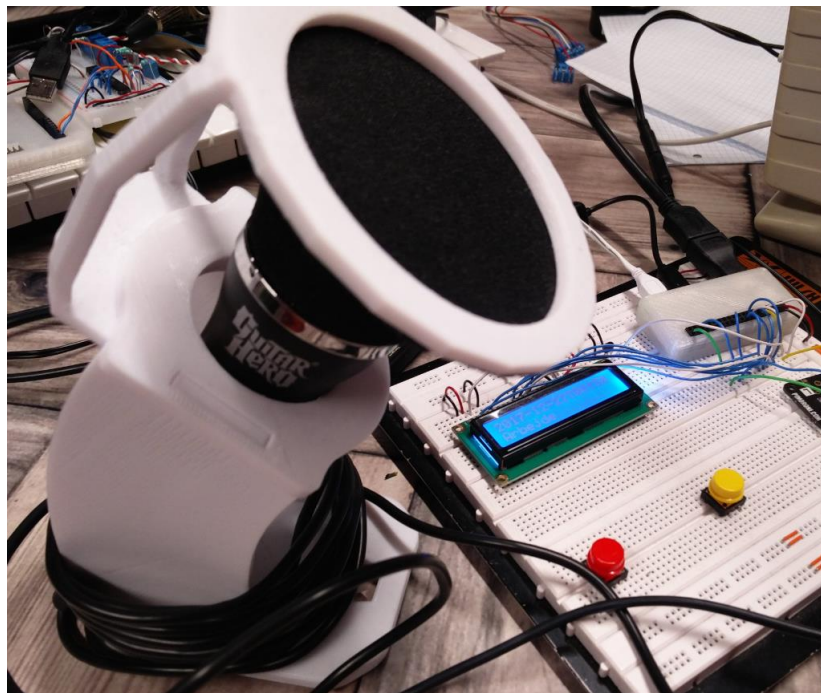


Figure 24 - Logitech USB Microphone, with a Guitar Hero logo on it. Works exceptionally well!!

The terminal shows the following in Figure 25 when recording and playing back, respectively:

```
pi@radIoT:~$ arecord --format=S16_LE --duration=5 --rate 16000 --file-type=wav test.wav
Recording WAVE 'test.wav' : Signed 16 bit Little Endian, Rate 16000 Hz, Mono
pi@radIoT:~$ aplay --format=S16_LE --rate=16000 test.wav
Playing WAVE 'test.wav' : Signed 16 bit Little Endian, Rate 16000 Hz, Mono
```

Figure 25 - Terminal output for recording and playing back audio using ALSA.

Any interested parties can find the Audio sample recorded using these methods in the Examples folder within the Github repository.

Verified Requirements

Out of the specified requirements to the project, the following have been verified by testing within this chapter:

- The system **shall** be capable of connecting to the internet.

1 The technical platform can be a suited embedded platform of your choice, e.g. the Particle Photon, an ESP8266, a raspberry pi, beagle bone black or similar.

1.1 The platform shall have Wifi connectivity

1.2 The device should preferably be able to connect to AU's "AU Gadget network"

- The system **shall** be capable of telling time accurately, and with minimal drift.
- While connected to an online calendar, the system **shall** use the "wake-up" times specified therein.

4 Your device must be capable of using data from a web service, to augment "what it does", this could be weather data, traffic data, stock prices, twitter feeds, emails, rss-feeds or something different.

1.2. The platform shall have available digital or analog I/O con connect sensors and actuators

3 Your device must be able to control an actuator

3.1 An actuator can be anything that translates an electrical signal into a physical quantity, such as, motors, servos, valves, heaters, displays, lamps, etc.

- While the playlists are unavailable, on-device songs **shall** be played.

2 Your device must be able to read data from a connected sensor, local to the device

2.1 a sensor can be anything that quantifies a physical measure, into an electrical signal, such as temperature, light, humidity, presence, movement, magnetism, pollution, etc.

The remaining requirements to be verified are the following:

- The system **shall** wake up the user, at the specified times.
- The alarm **will** be pleasant to awaken to.
- Voice commands using online voice recognition is **to be** implemented.
- When a specified time arrives, the system **shall** start the alarm.
- When the alarm starts playing, the system **will** slowly ramp up the sound.
- When the user has awoken, a weather message **will** be displayed.
 - When unavailable, the system **will** display a "weather unavailable" message.
- While the calendar is unavailable, the system **shall** use hardcoded "wake-up" times.
 - If possible, the system **must** store the latest calendar updates to use instead.
- While connected to an online playlist service, the system **must** stream songs from there as alarms.
- Where possible, tough calculations **will** be performed off-device.
- Where user input is available, the input **must** be possible through button interface at least.
 - A microphone as interface is **to be** implemented if possible.
- If system cannot access information while online, a debug message is **to be** implemented.

Functional requirements

5 Your software and hardware design must be shared

5.1 You must create a public **github** account, and add relevant project files here

5.2 Hardware documentation, schematics, datasheets and pcb layouts are to be uploaded in pdf format

5.3 Software files are to be uploaded in raw source code format, e.g. **.C, CPP, .h, .py**, etc.

Please note that while not directly verified herein, the Functional requirements under part 5 are all fulfilled, as the sources for this project are available on https://github.com/lakop/E5IoT_AlarmClock/ at the time of writing.

Discussion

While not everything was implemented throughout this project, some good headway was made.

For example, the usability of the Microphone input was verified in preparation for Google Assistant compatibility, and the Libspotify SDK was discovered dead, but has hope for the future in the form of a new embedded music player for use with Linux among other OS'es.

Regarding the full current implementation, it is still severely lacking in terms of pure usability. However, the project was also bound by harsh time constraints and shortage of manpower. Many of these issues could stem from the project being carried out by only a single developer, who undertook this assignment alone instead of working in a larger group.

The lesson learned, is to not take these tasks lightly. As just as a team is required to maintain the RESTful API's at Google, Amazon or wherever, a team might be needed to tie them well together in a timely fashion.

Future development includes a fully tested and fleshed out State Machine, with button inputs to control the GUI on the LCD. Then the Weather API will be added, and finally the project will look towards Google Assistant for voice commands.

Hopefully a few free weekends in the future will provide the time needed to see these changes through!

Conclusion

The Project fulfills the bare minimum requirements set in the course description. This is in itself satisfactory to a point; however, a lot of future development is required in order to create a commercially viable product.

The current point of development is a hobby level system at best. A lot of the development hinged on Linux specific problems that arise when working on an IoT device of this complexity.

It can be concluded, that the simplest, easiest to develop and most robust IoT devices are the ones that are employed on a simple platform, like the Particle Photon [23] or similar, that have well defined backends and interfaces to the rest of the web, and don't rely on much internal setup to function.

This project is by no means a failure, but serves well to illustrate how more complex systems tend to make development, of especially small IoT devices, very troublesome.

References

- [1] Aarhus University Herning, "EUDP in Detail - EUDP," 21 januar 2015. [Online]. Available: http://eudp.dk/index.php/EUDP_in_detail. [Accessed 8 december 2017].
- [2] Aarhus University Herning, "EARS Requirements Capture - EUDP," 13 august 2012. [Online]. Available: http://eudp.dk/index.php/EARS_Requirement_Capture. [Accessed 8 december 2017].
- [3] Aarhus University Herning, "Requirements Analysis - EUDP," 21 september 2015. [Online]. Available: http://eudp.dk/index.php/Requirements_Analysis. [Accessed 8 december 2017].
- [4] BeagleBoard, "System Reference Manual · beagleboard/beaglebone-black Wiki," 18 oktober 2017. [Online]. Available: https://github.com/beagleboard/beaglebone-black/wiki/System-Reference-Manual#617_Power_Consumption. [Accessed 12 december 2017].
- [5] Raspberry Pi Foundation, "Raspberry Pi FAQs - Frequently Asked Questions," 2017. [Online]. Available: <https://www.raspberrypi.org/help/faqs/#powerReqs>. [Accessed 12 december 2017].
- [6] J. Dunn, "Most popular music streaming services: CHART - Business Insider," 13 marts 2017. [Online]. Available: <http://www.businessinsider.com/pandora-spotify-most-popular-music-streaming-service-chart-2017-3?r=US&IR=T&IR=T>. [Accessed 12 december 2017].
- [7] Reuters, "Apple Music, Spotify, Tidal: A Guide to Music Streaming | Fortune," 11 september 2017. [Online]. Available: <http://fortune.com/2017/09/11/spotify-apple-music-tidal-streaming/>. [Accessed 12 december 2017].
- [8] S. M. Jodal, "pyspotify — pyspotify 2.0.5 documentation," 2015. [Online]. Available: <https://pyspotify.mopidy.com/en/latest/>. [Accessed 12 december 2017].
- [9] H. Pham, "PyAudio: PortAudio v19 Python Bindings," 2006. [Online]. Available: <http://people.csail.mit.edu/hubert/pyaudio/>. [Accessed 12 december 2017].
- [10] Adafruit, "Python Script | Drive a 16x2 LCD with the Raspberry Pi | Adafruit Learning System," 13 november 2015. [Online]. Available: <https://learn.adafruit.com/drive-a-16x2-lcd-directly-with-a-raspberry-pi/python-code>. [Accessed 12 december 2017].
- [11] G. Henderson, "Raspberry Pi | WiringPi | LCD Library | Gordons Projects," 2016. [Online]. Available: <https://projects.drogon.net/raspberry-pi/wiringpi/lcd-library/>. [Accessed 12 december 2017].
- [12] B. Cooke, "fivdi/lcd: Node.js Hitachi HD44780 LCD driver," 4 november 2017. [Online]. Available: <https://github.com/fivdi/lcd>. [Accessed 12 december 2017].
- [13] G. Henderson, "Drogon Projects | Git - wiringPi/summary," 3 marts 2017. [Online]. Available: <https://git.drogon.net/?p=wiringPi;a=summary>. [Accessed 12 december 2017].
- [14] B. Cronston, "RPi.GPIO 0.6.3 : Python Package Index," 30 oktober 2016. [Online]. Available: <https://pypi.python.org/pypi/RPi.GPIO>. [Accessed 12 december 2017].

- [15] Google, "Python Quickstart | Calendar API | Google Development," 2 august 2017. [Online]. Available: <https://developers.google.com/google-apps/calendar/quickstart/python>. [Accessed 12 december 2017].
- [16] OpenWeatherMap, "Pirce - OpenWeatherMap," 2017. [Online]. Available: <https://openweathermap.org/price>. [Accessed 12 december 2017].
- [17] Weather Underground, "API | Weather Underground," [Online]. Available: <https://www.wunderground.com/weather/api/d/pricing.html>.
- [18] Google, "Overview of the Google Assistant Library for Python | Google Assistant SDK | Google Developers," 20 juli 2017. [Online]. Available: <https://developers.google.com/assistant/sdk/develop/python/>. [Accessed 12 december 2017].
- [19] Amazon.com, "Alexa Voice Service Overview (v20160207) | Alexa Voice Service," 2017. [Online]. Available: <https://developer.amazon.com/docs/alexa-voice-service/api-overview.html>. [Accessed 12 december 2017].
- [20] M. Behabadi, "oxplot/fysom: Finite State Machine for Python (based on Jake Gordon's javascript-state-machine)," 22 august 2013. [Online]. Available: <https://github.com/oxplot/fysom>. [Accessed 12 december 2017].
- [21] Spotify, "Libspotify SDK - Spotify Developer," 2016. [Online]. Available: <https://developer.spotify.com/technologies/libspotify/#libspotify-and-cocoalibspotify-downloads>. [Accessed 13 december 2017].
- [22] Pimoroni, "pHAT DAC - Pimoroni," 2017. [Online]. Available: <https://shop.pimoroni.com/products/phant-dac>. [Accessed 13 december 2017].
- [23] Particle, "Particle Photon series Wi-Fi development kits and connectivity modules.," 2016. [Online]. Available: <https://www.particle.io/products/hardware/photon-wifi-dev-kit>. [Accessed 13 december 2017].