# 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 them pleasantly. This could perhaps be coupled with therapy lighting!

The Device has a cloud functionality, that pairs up with a Google 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. I 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.

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

# Requirements Analysis

I 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 **w**ill 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 one will likely entail the other.

# 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

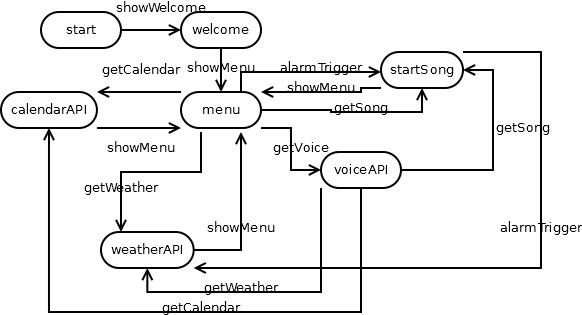


Figure 1 - FSM diagram of the system to be.

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 man 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 1, 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.

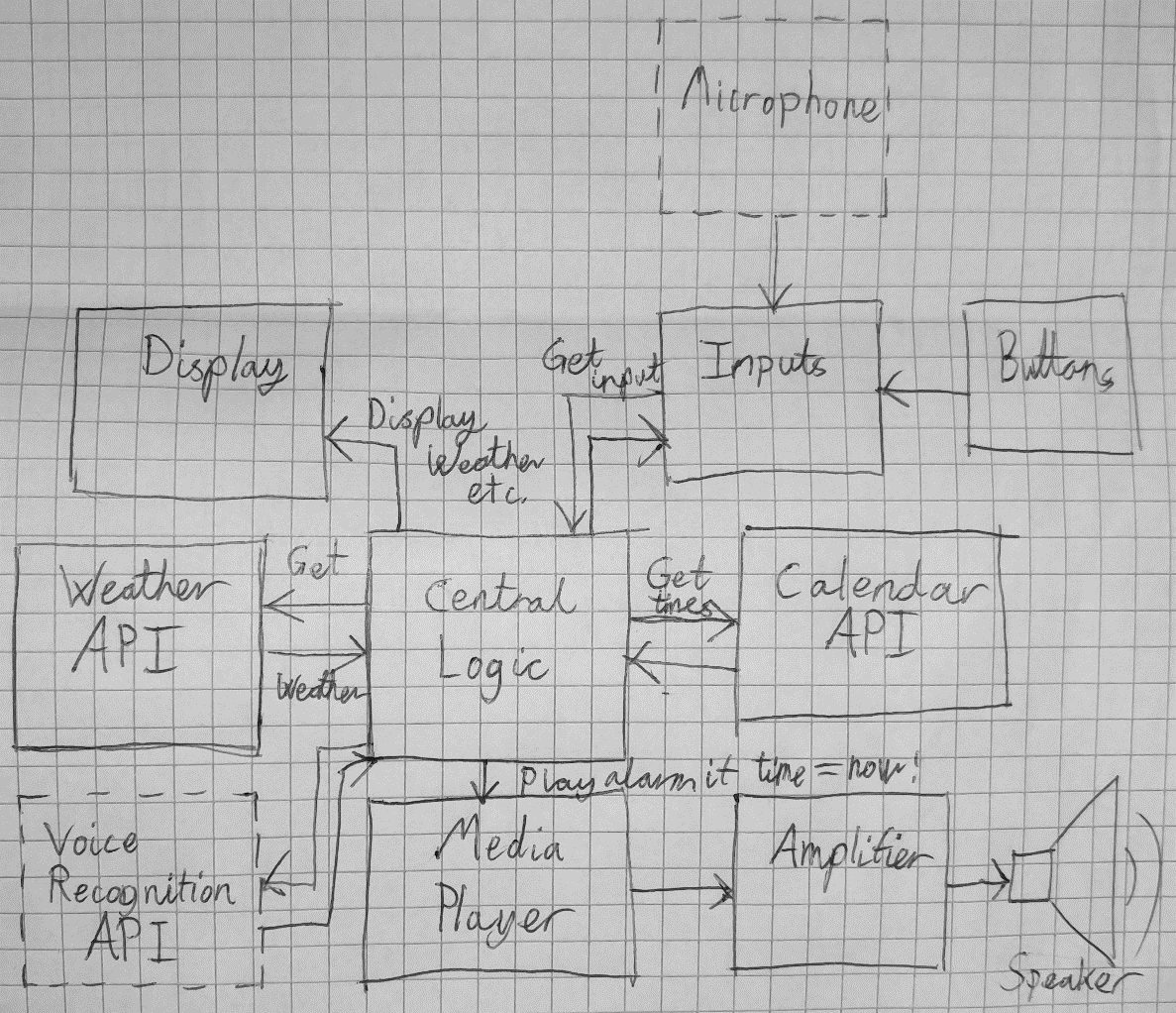


Figure 2 - A simple block diagram illustrating the construction of the system-to-be.

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.

## Hardware Considerations

One thing however, is abundantly clear. Since the device needs to access both WiFi, web API’s and play music, it would make the most sense to have an operating system on it. 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 supplied overlays 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, along with a TRS output.
  + 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.

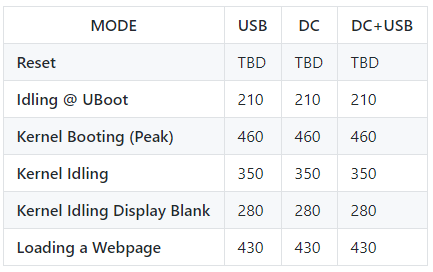


Table 2 - BeagleBone Black power copnmsumption, taken from the BeagleBoard Wiki [4].

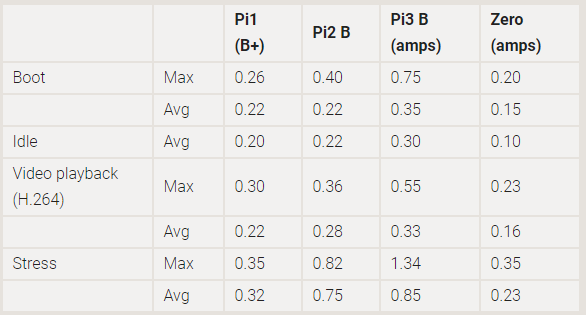


Table 3 - Raspberry Pi platform power consumption. Taken from the Raspberry Pi FAQs [5]. Note that Zero's draw is without the WiFi and bluetooth module in Zero W.

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

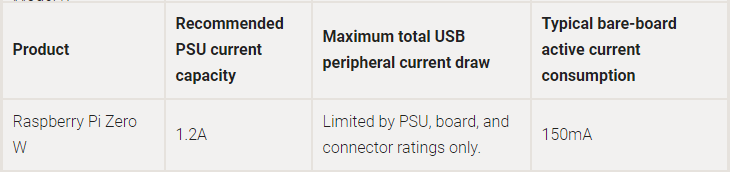


Table 4 - Raspberry Pi Zero W bareboard active consumption.

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 scenariuo only arises whenever

## 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 [4], the most widely used music platform for streaming is Pandora, which carries around 32% of all listeners in America as shown in Figure 2. While this is an impressive statistic, it carries the unfortunate caveat that this only applies to the United States.

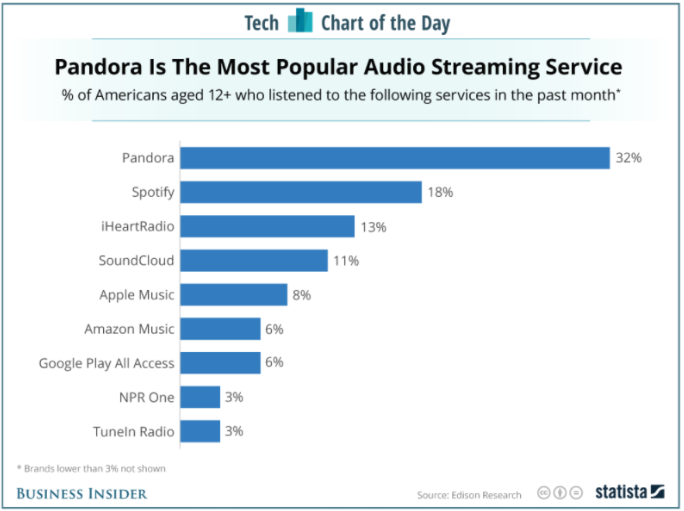


Figure 3 - Market shares divided between the music streaming services in America.

A more recent article by Reuters on Fortune points this out as well, and also lists Spotify as the biggest player on the worldwide market, with over 140 million active users [5].

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 the 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 function, 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, 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 [4], that even includes a wiring diagram for their cobbler GPIO breakout board:

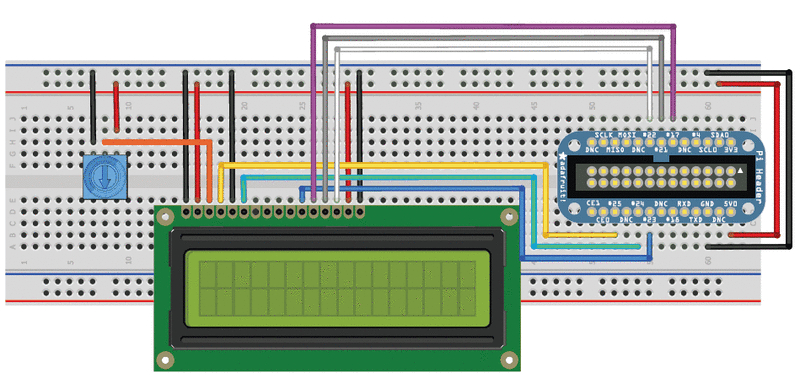


Figure 4 - LCD connected to an Adafruit Cobbler Board.

The wiring itself is perfectly fine, and can interact with several different libraries. Other implementations exist, either through pre-compiled C-based ones [5], or Node.js implemented ones [6].

### 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 esample, the wiringPi library [11] for C/C++ by Gordon Henderson and his contributors. There is also the RPi.GPIO module [12] for Python by Ben Cronston available to keep going out the Python route.

### 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, the easier development gets.

There are extensive tutorials and documentation available as well [13], which helps out a lot.

### Weather API

OpenWeatherMap and Weather Underground seem to be two big players on the field of weather APIs. OpenWeatherMap wins on value for the free pricing range, however [14] [15]. 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.

Running Spotify in Linux:

<https://www.spotify.com/dk/download/linux/>

<https://github.com/pwittchen/spotify-cli-linux>

Spotify playback in JS:

<https://developer.spotify.com/web-api/>

Google assistant API:

<https://developers.google.com/assistant/sdk/develop/python/>

<https://developers.google.com/assistant/sdk/develop/grpc/integrate>

pyAudio:

<http://people.csail.mit.edu/hubert/pyaudio/>

Finite State Machine:

<https://github.com/oxplot/fysom/blob/master/fysom.py>