# RasPi-PrudensJS User Guide

Yiannis Michael

September 1, 2023

## 1 Introduction

This document contains how to execute the various functionalities of the RasPi-PrudensJS Smart Home AI Assistant. It is a functioning early prototype that can be built upon for later versions. All work was done during the CYENS SCRAT Internship. For a full understanding of certain functionalities please refer to the project report and project code on the GitHub repository [1].

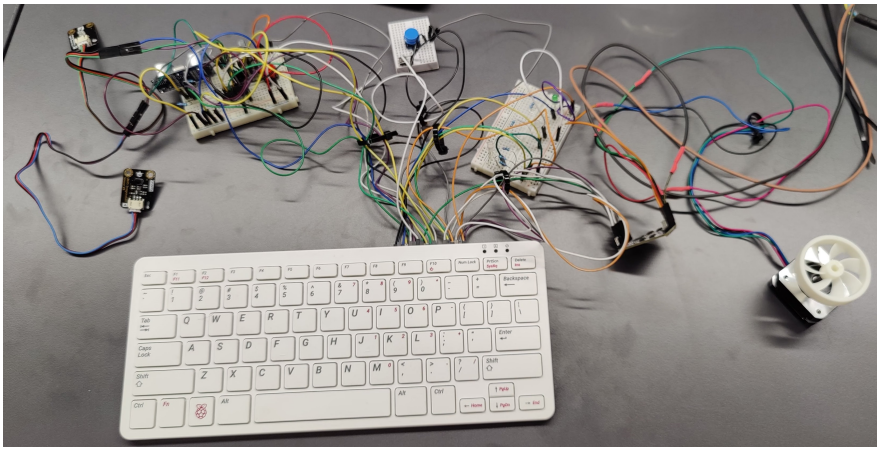

Figure 1: Full RasPiPrudensJS System

## 2 Supported Devices

1. Button
2. HC-SR04 Ultrasonic Sensor [2]
3. LEDs
4. RGB LEDs
5. Analogue devices (with MCP3008),
   (a) Potentiometres
   (b) DFR0023 LM35 Linear Temperature Sensor [3]
   (c) DFR0026 Ambient Light Sensor [4]
6. TMC5160-BOB Stepper Motor Controller [5]

# 3 Installation

To install the RasPi-PrudensJS system, clone the GitHub repository with *–recursive* to also clone the submodules. Install the `spidev` and `RPi.GPIO` Python libraries. Python 3.9 was used.

```
$ git clone https://github.com/ymich9963/RasPi-PrudensJS --recursive
$ pip install spidev
$ pip install RPi.GPIO
```

In the *blank_system* folder, is a system with no drivers, device instantiations (only the mandatory system buttons) or Policy versions. From that a new assistant can be built based on the users needs. Use the following commands to start the system,

```
$ cd path/to/RasPi-PrudensJS
$ python dev.py
```

## 3.1 Correcting the File Paths

Due to the current project being an early prototype, the cloning of the submodules, and the RasPi username, some manual changes to file paths have to be made in order for the system to work. This should be checked for **both** the main directory and the blank system.

1. Make sure *prudens-js/node/app.js* is pointing to the correct policy file.
2. Changes in *sys_functions.py*
   (a) Correct the path of the *subproc()* function to point to the correct *app.js* file.
   (b) Correct the path of the *change_policy_version()* function to point to the correct policy file.
   (c) Correct the path of the *copy_from_USB()* function to point to the correct USB drive. Also make sure the USB drive has the correct directory set up.

# 4 Adding A New Device

A device can either be a *Sensor* or an *Actuator* and requires a *Driver File* to allow it to interface with the system. It must be declared as an Object in the *tech.py* file by either the technician (or the user). Device pins are advised to be indicated using the Broadcomm Pin numbering system, which uses the number *X* in *GPIO X* instead of the board pin numbers. It is defined using the following lines of code,

```
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
```

This is only required for digital peripherals (e.g. HC-SR04, Buttons, LEDs) and not analogue peripgerals (e.g. potentiometre).

It is strongly advised to use pinout.xyz for figuring out which pin to choose when connecting a new device. This website also shows the alternative uses for certain pins which is vital for understanding which pins are used for the Analogue to Digital Converter (ADC on SPI0) and the motor (TMC5160-BOB on SPI1). The steps for adding and implementing new devices are using the blank system as the starting point.

## 4.1 Driver File

Driver files have to be coded with a *setup function* and an *action function* in order to be used when instantiating the device. Manually copying a driver file to the *drivers* folder is one way of installing, but they can be automatically copied by having a USB drive named *DRIVER_USB* and the path */media/{RasPi Username}/DRIVER_USB/RasPi-PrudensJS/drivers/*. The {*RasPi Username*} must be the username used on the system and should therefore be changed accordingly. Using this feature with the restart button, allows for uninterrupted installation of new devices. The *drivers* directory is considered a module so it can be utilised by using,

```
#driver imports
from drivers import your_driver_file as device
```

## 4.2 New Sensor

To add anew sensor, the driver file first has to be imported. The example in the *Driver File* section will be used. In the *tech.py* file, is where the technician would instantiate the new device for the user (can also be done by the user).

A digital sensor must be instantiated with the following format inside the *sens_array[]*,

```
cf.Sensor(
            Sensor ID,
            pin or channel,
            positive  literal ,
            negative  literal ,
            action function,
            setup function,
            adc_function
        )
```

Two buttons are already instantiated as examples in the blank system. Using the *driver* imported from before it would look like,

```
cf.Sensor("Sensor1",[1 2 3 4], " device_literal ", "-device_literal", device.action, device.setup)
```

if it is a digital sensor. The list with the numbers 1, 2, 3, and 4 represent the GPIO pins used by the sensor. For an analogue sensor it would be similar to,

```
cf.Sensor("Sensor1",0, "device(X)", "", device.action, adc_fcn = fcn.adc_read)
```

Here the number 0 represents the channels 0 to 7 which can be used as analogue inputs on the MCP3008 (channel 7 is broken from testing). Furthermore, here the value read by the sensor will be inserted in the position of *X*. This is also seen in the case of the HC-SR04 where it is a digital sensor, but it also send continuous data. The strings in the *literal* positions, are the ones that will be used in the Prudens Context. Furthermore, Sensor ID is a unique string that should be set by the technician for better debugging.

## 4.3 New Actuator

For a new actuator to be added, a similar process like for adding a sensor must be taken. Using the previous example, it is assumed the driver is imported in *tech.py*. Then the instantiation of the *Actuator* object has to be inside the *act_array[]* in the following format,

```
cf.Actuator(
            Actuator ID,
            pin,
             literal ,
            action function,
            setup function
        )
```

Therefore using the *driver* example, it would be written as such,

```
cf.Actuator("Actuator1", [1 2 3], " literal ", drvr.action, drvr.setup),
```

The functionality of each argument is the same as in the digital sensor case. For each actuator action, a new object has to be created. For example, in the case of an RGB LED,

```
cf.Actuator("RGB_led",[13, 12, 6], "rgb_led_red_on", rgb.on_red,rgb.rgb_setup),
cf.Actuator("RGB_led",[13, 12, 6], " rgb_led_red_off ", rgb. off_red ,None),
cf.Actuator("RGB_led",[13, 12, 6], "rgb_led_yellow_on", rgb.on_yellow,None),
cf.Actuator("RGB_led",[13, 12, 6], " rgb_led_yellow_off ", rgb. off_yellow ,None),
```

Each colour outputted has a different entry in the array, with the setup function being only written for the first entry. In the case of a motor,

```
cf.Actuator("Motor1", None, "move_motor_1", motor.spin1, motor.setup),
```

There are no pins to be used due to them being hardcoded to the SPI1 bus, it is therefore declared as *None* for all entries.

## 4.4 Implementing New Device

To implement the new sensor or actuator object in the system and allow them to interface with Prudens they have to be included in the Context and Policy. Using the previously instantiated objects, an example Policy is shown below,

```
@KnowledgeBase
    R1 :: device(X), ?check(X) implies device1_action1;
    R2 :: device(X), −?check(X) implies device1_action2;
    C1 :: device1_action1 # device1_action2;
    R3 :: device_literal implies device2_action1

@Code
function check(x) {
    return parseFloat(x) > 10;
}
```

The literals used for the sensor objects are automatically added to the Context which is passed to Prudens for deduction. The constraint is used due to certain cases where an actuator is used more than one time, either by the same sensor or a different one. This limits the actuator to only one action function at a time. To understand more about Prudens and specific formatting, please read the corresponding paper [6].

Due to no actual natural language user input being available, the inclusion of new rules is made by having different policy levels. If a new device is added, the newest policy text file is copied and renamed to be the next policy version. For example, if *policy4.txt* is the currently newest policy file, it should be copied, renamed to *policy5.txt* and then the new rules should be appended in the new file. These files are cycled through by either user input to the terminal or by pressing the blue button on the user input board (default). The *total_policy_files* in variable in *tech.py* should then be changed to the new total policy files in the *txt* directory.

# 5 Running Built-In Example

Using the main *RasPi-Prudens* directory, there is a built in example to showcase the capabilities of the system. The devices have to be wired according to the following diagrams,

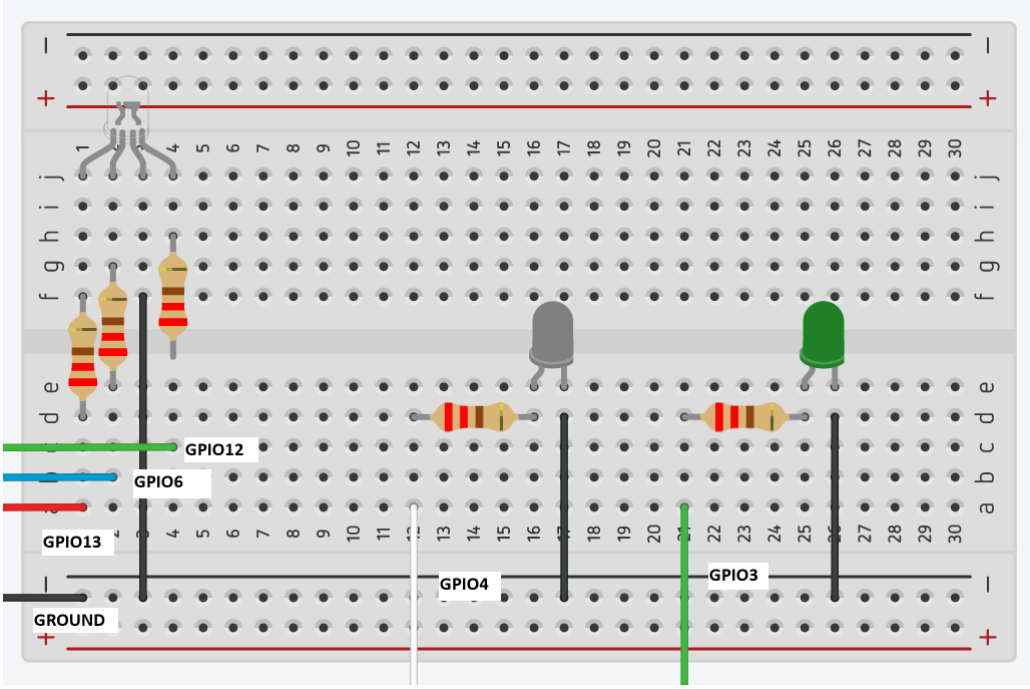


Figure 2: Actuator Board Diagram

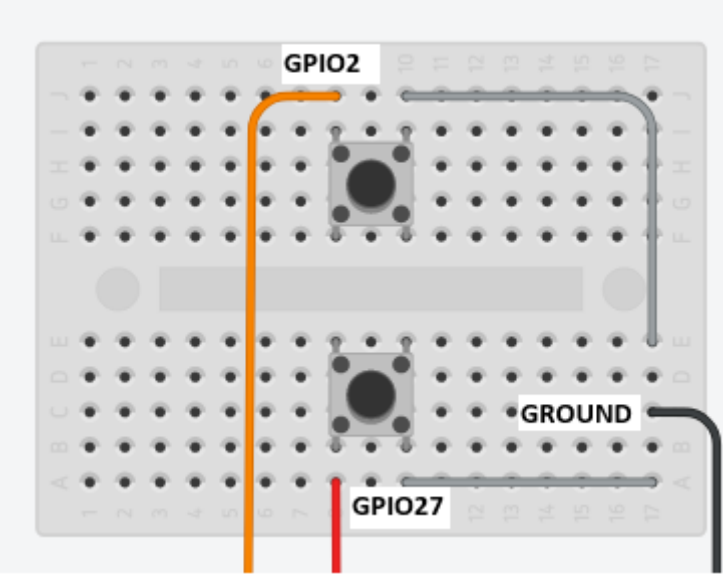


Figure 3: User Input Board Diagram

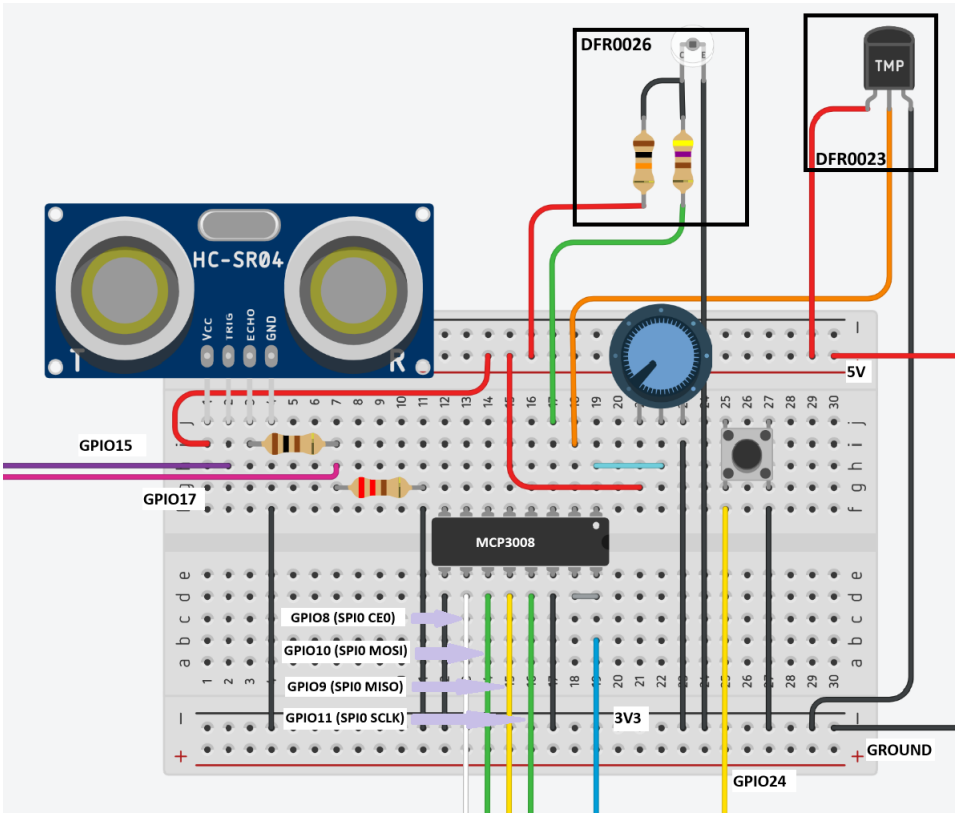


Figure 4: Sensor Board Diagram. The black outlines for the sensors are there to symbolise them but do not represent the sensor schematic.

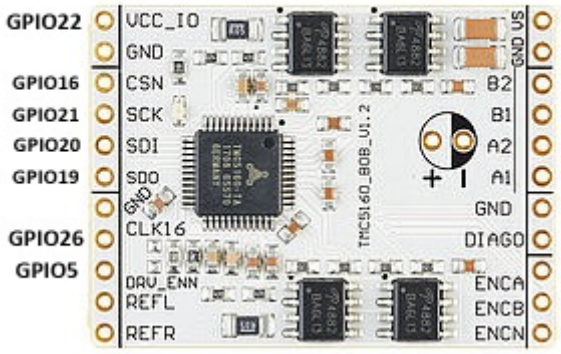


Figure 5: TMC5160-BOB Diagram

In the above figures, the layout can vary but it must have the correct GPIO pin connections. In Figure 4, the SPI0 bus is used whereas in Figure 5 SP1 is used. The TMC5160 power supply wires and motor wires were soldered with female headers for a safe connection to the male headers on the breakout board (BOB). The motor wire pairs also have to be inserted correctly into A1/2 and B1/2. Use the following commands to start the system,

```
$ cd path/to/RasPi—PrudensJS
$ python dev.py
```

### 5.1 Policy Version Changes

The way the Policy changes in the built-in example is based by these inputs to the system by the user. In the actual program they serve no purpose and only exist to explain the additions at each version.

- Policy Version 1: If a car is detected over 10cm, blink the Green LED slowly.
- Policy Version 2: If a car is detected closer than 10cm, blink the Green LED fast.
- Policy Version 3: If I am home please switch off the Green LED.
- Policy Version 4: If I am home and you detect something closer than 10cm, turn on the White LED.
- Policy Version 5: If it's starting to go dark, please turn on the lights.
- Policy Version 6: If I turn the dial, please turn on the fan.
- Policy Version 7: If the temperature is more than 28 C and the fan is on, please increase the fan speed.
- Policy Version 8: If it's really dark please turn on the red lights.

## 6 Program Start at System Startup With No Monitor

By modifying some configuration files in the RasPi directories, the program can be made to run at startup and therefore removing the need to execute commands and being connected to a monitor. This has the downside of not allowing user input through the terminal and can only be cycled through the user input button. This can be accomplished by writing the following commands in the terminal,

```
$ sudo nano /etc/rc.local
$ sudo python /path/to/dev.py
```

The last line should be written above the *exit 0* line. Now when turning on the RasPi, the program will automatically start. This can be disabled by adding a # at the start of the added line. To stop the program, run the following command in the terminal,

```
$ ps aux | grep /home/RasPi Username/cyens/
```

and then use the following with the found process ID, to stop the Python process.

```
$ sudo kill {PID}
```

## References


[1] Y. Michael, "Raspi-prudensjs," 2023.

[2] Sparkfun, "Hc-sr04," 2017.

[3] DFRobot, "Dfr0023 temperature sensor," 2000.

[4] DFRobot, "Dfr0026 ambient light sensor,"

[5] Trinamic, "Tmc5160-bob," 2019.

[6] V. T. Markos and L. Michael, "Prudens: An argumentation-based language for cognitive assistants," *ResearchGate*, 2022.