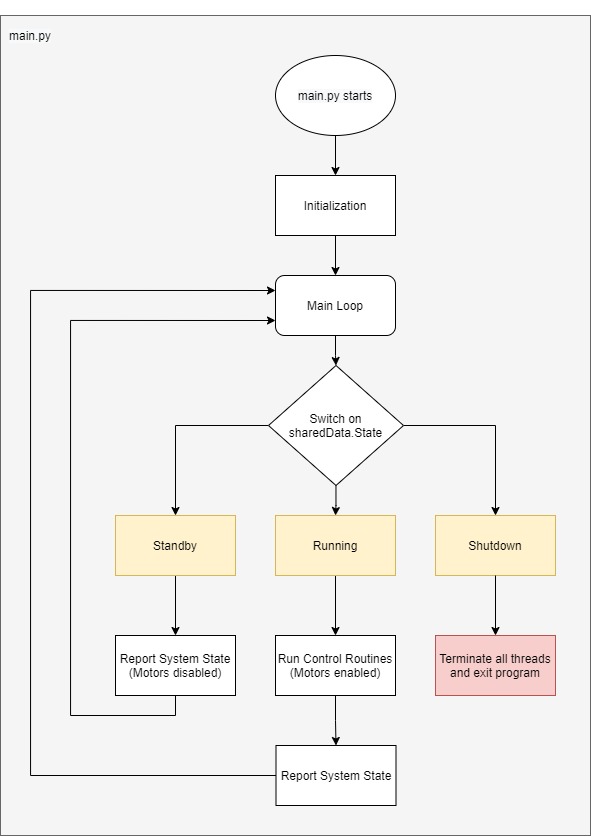
## Introduction and Overview

Prudentia is started by running *main.py*. Alternatively, a button press is configured to run a batch file to start *main.py*. Prudentia then runs initialization and enters its main loop. This loop is run off a state machine with three modes: **standby, running, shutdown**. By default, Prudentia starts in **standby.**

* **standby:** No control routines are called, but all sensor data is being refreshed. The GUI can interact with the system by reading data and sending commands.
* **running:** Control routines are called – motors are enabled, and control laws are in effect.
* **shutdown:** When set to this state, Prudentia closes all threads and exits cleanly.

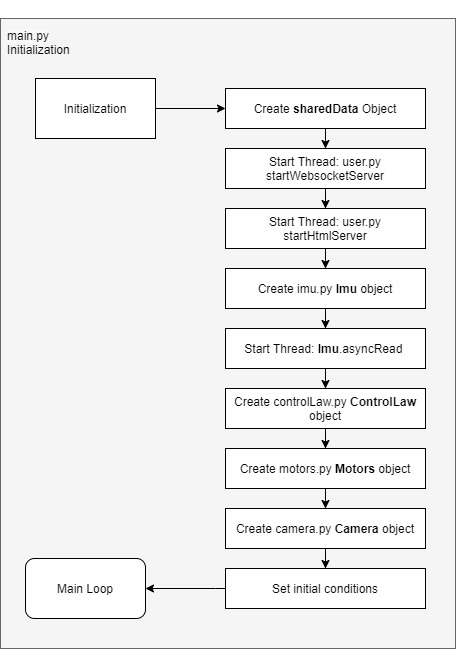


Note that the system state is stored in a variable called **sharedData**. This is an object defined in user.py that stores a shared copy of any information that is needed between the GUI and Prudentia. We will cover this in more detail later.

## Initialization

During initialization, several objects are created.

* The WebSocket server is created on a new thread. This allows the GUI to connect to Prudentia.
* The **IMU** object is created. This opens a serial port and starts reading data. Another thread is created to read IMU data asynchronously.
* The **ControlLaw** object is created. This object holds control routines that execute Simulink models.
* The **Motors** object is created. This object holds motor related functions.
* The **Camera** object is created. This object holds camera related functions.
* Variables are initialized, such as update frequency and initial states.



## Main Loop

The main loop’s behavior is modified by the *sharedData*.*state* variable. When set to *shutdown*, all threads are sent stop signals and the program is terminated. When set to standby, the *Imu,* *Camera, and ControlLaw* objects are run to collect data about the system state. This information is also stored in sharedData, which is distributed to the user GUI asynchronously. When set to *running*, Prudentia runs the same processes as standby as well as enabling motor signals.

By using objects (*Imu, Camera, ControlLaw, Motors)* to store different functions, we can reduce how much code is in our main loop. For example, running the RTC control law from the main loop would look like this:



Here, we input attitude data from the IMU. Then, we get a response as motor torques. Everything in between happens in *controlLaw.py* within the ControlLaw class.

This is beneficial; debugging faulty code is faster when you know exactly which object caused an error. It also allows for any of these processes to be easily added or removed to different states. This architecture should be consistent throughout the main loop.

Note that changing the state to *shutdown* causes the program to end. To restart Prudentia, you must rerun main.py.

The next sections will cover each object and its responsibilities.

## Imu Object (imu.py)

The *Imu* object holds IMU output data, such as *position* and *velocity.* The *Imu* object also holds two functions:

* openConnection() – this is called after object creation and attempts to establish a serial connection on the port and baudrate supplied. It sets a connection object named *conn*. We use this to read from the serial output later. During initialization in the main thread, *conn* is also returned to assert a connection was established.
  + Port and baudrate parameters have not been decided yet.
* asyncRead() – this is run as a new thread in the main thread after connection has been established. This enters a loop that checks the serial buffer for data at 200Hz. The IMU should be outputting data at that rate or higher.
  + asyncRead needs a function to read expected serial strings.
  + asyncRead needs to process data from the IMU quickly and efficiently.

## Control Law Object (controlLaw.py)

The ControlLaw object is used to run control law routines. Control law routines include:

* *standby* - Do nothing.
* *stabilize* – Create a target vector at [0, 0, 0] and supply to Simulink model.
* *realTimeControl –* Create a target vector based on RTC input. This vector should be in the supplied direction away from the current attitude. The angle between the target and current attitude can be increased or decreased to match a input angular speed. Pass this vector to the Simulink model.
* *attitudeInput* – Create a target vector from input and supply to Simulink model.
* *Search –* if the camera supplies a target vector, supply that to the Simulink model. If no target vector was found, supply a vector that is offset in the yaw direction from current attitude. This will cause Prudentia to yaw until the camera finds a target.

## Motors Object (motors.py)

The Motors object is used to send the required PWM signals to specific motors when given an input RPM or Torque. Hardware PWM pins are coded as variables under this object. Several functions help drive this process:

* setAllMotorRpm (motorArray) – This function expects an array of motor RPMs (ex: [1000, 2000, -2000, 3000]) and then calls setMotorRpm(motor) for each motor.
* setMotorRpm(motor, rpm) – This function expects an integer representing a motor (0 to 3) and an associated rpm.
  + First, the motor number is converted to a hardware PWM pin.
  + Second, the duty cycle is calculated by getDutyFromRpm(rpm)
  + Third, the duty cycle is applied to the hardware PWM pin through setPWM(hardwarePin, dutyCycle)
* getDutyFromRpm(rpm) – This function converts an input RPM to a duty cycle.
* setPWM(hardwarePin, dutyCycle) – This function sets the hardware pin to a PWM signal with a specified duty cycle.

This is quite subject to change, as we need to concretely decide what outputs the Simulink model provides and how to convert that to a corresponding duty cycle.

## Camera Object (camera.py)

The camera object is responsible for taking pictures, identifying a target, and translating the target on the screen to a relative attitude. This is accomplished with the functions below:

* getPicture() – This takes a picture.
* findLocation(picture) – This processes a picture to find a target, if one exists.
* translatePosition(location) – This converts a screen location to a relative attitude.

## Shared Data and WebSocket Server (user.py)

Two threads are created to support the GUI. The first is a WebSocket server that handles data transfer. The second is a simple HTML server to deliver our custom GUI to the user. This section will cover usage pertinent to anybody modifying information sent to the GUI or commands sent to Prudentia.

### HTML Server

The HTML server is started on a new thread. The sole purpose of this server is to return our custom HTML content when a user connects to the machine. A user can access the GUI by typing in the ipAddress:port in their browser. This requires them to be on the same network; the ip address and port must be known as well. The ip address should be a static ip address assigned by the school. The port can be modified as needed and is currently running on port 8009.  
  
Example: (This ip should be updated when a static ip is assigned) Alternatively, we could write a simple program to find the ip based on the Raspberry Pi’s MAC.



### Sending Data to the GUI

Data is shared through a WebSocket server that is run on another thread. Any information that needs to be shared between server threads and the main thread is stored on the *sharedData* object. This object is a singleton (only one instance should be made) and defined under *SharedDataPackage* in user.py.

Any information you want to share between the GUI and main thread should be stored in this object. For example, we define/modify a variable in the main thread:

sharedData.angularPosition = [0, 0, 0]

This sets a variable named *angularPosition* to [0, 0, 0]. When the GUI requests data, the WebSocket thread calls sharedData.getDataJson() shown below.

def getDataJson(self):  
 dataObject = { "state" : self.state.name,  
 "angularPosition" : self.angularPosition,  
 "angularVelocity" : self.angularVelocity,  
 "target" : self.target}  
 return json.dumps(dataObject)

This then converts stored variables to JSON format and sends the data to a browser. Finally, to complete the transfer, the browser parses the sent JSON. This is done in script.js in the user’s browser:

websocket.onmessage = function (event) {  
 data = JSON.parse(event.data);  
 console.log(data["angularPosition"])

Output: 

From here, you can bind data[“angularPosition”] to the relevant HTML component.

### Sending commands to Prudentia

This works similarly to getting data, but in reverse. First, the user presses some command button on the GUI that calls a function in script.js.

runningBtn.onclick = function (event) {  
 setState("running")  
}  
  
function setState(state){  
 var msg = {"messageType":"setState", "state":state}  
 websocket.send(JSON.stringify(msg));  
}

This sends a JSON to Prudentia with two attributes: *messageType* and *state*. Every message transmitted to Prudentia should have a *messageType*. In this case, it is *setState*. The other attribute *state* holds the state we are trying to set.

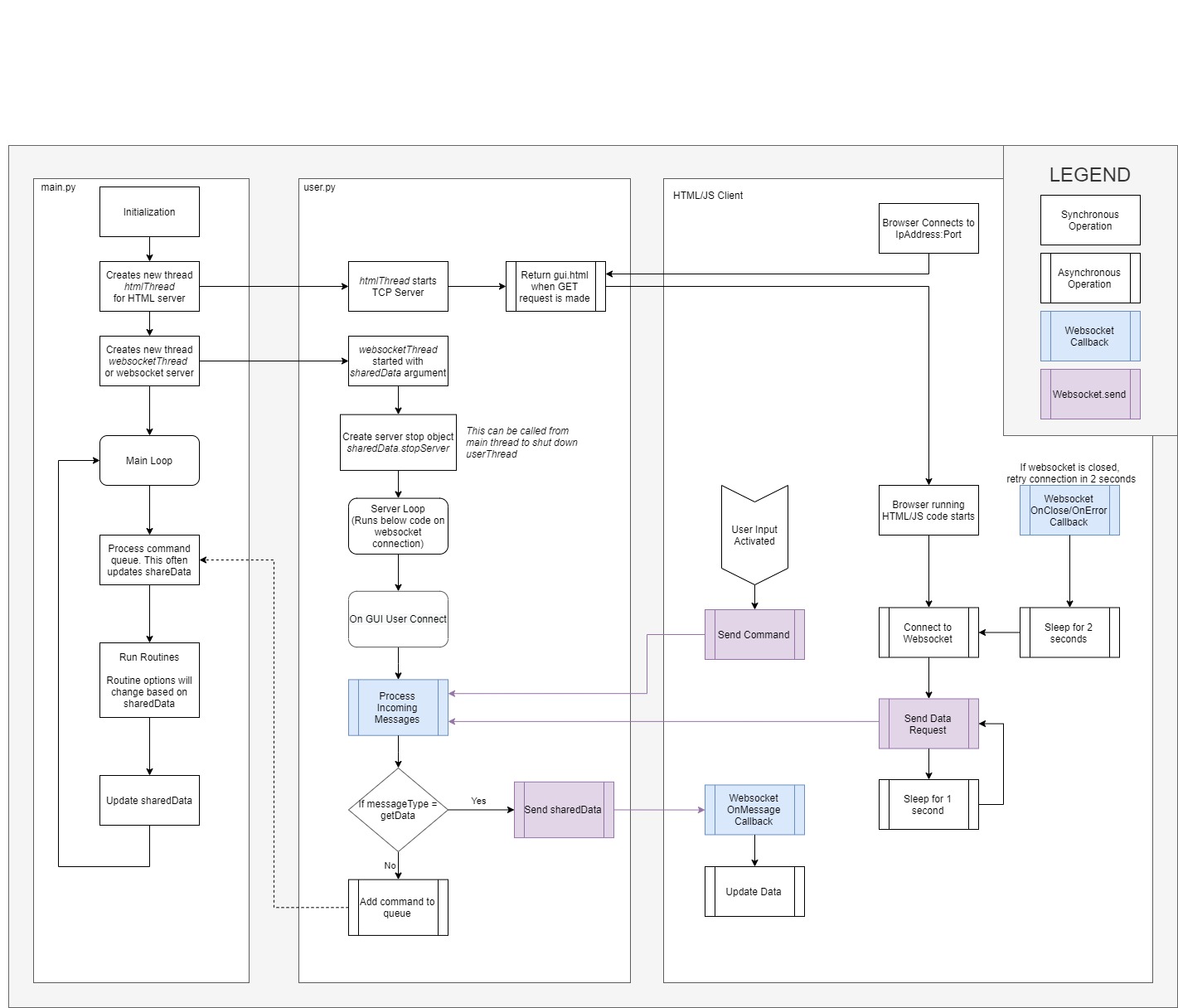
Prudentia collects any commands and stores them in *sharedData.commandQueue*. This queue is processed at the beginning of the main loop:

def processCommands():  
 while not sharedData.commandQueue.empty():  
 msgJSON = sharedData.commandQueue.get()  
  
 # State change was issued  
 if msgJSON["messageType"] == "setState":  
  
 if msgJSON["state"] == "shutdown":  
 sharedData.state = State.shutdown  
  
 elif msgJSON["state"] == "standby":  
 sharedData.state = State.standby  
  
 elif msgJSON["state"] == "running":  
 sharedData.state = State.running

This then modifies the current state of Prudentia. This can be applied to adjust any state variable or issue other commands to Prudentia as needed.

### User Control Flow Diagram

Shown below is a control flow diagram for the above processes. The main thread in Prudentia is shown on the left, the two server threads (HTML and WebSocket) are created and run in the middle, and the browser side code is shown on the right.



What we need in terms of coding:

**controlLaw.py - Brad**

* What inputs does a standalone C Simulink model take? (ex: control parameters, IMU data, etc)
* How to call the C code from python? (May need to make a wrapper)
* What outputs are needed from Simulink model, and how to get them?
* We need to make sure it's computationally reasonable ASAP
* Start writing control routines (stabilize, point at coordinate, etc)
  + This is pretty much taking input data and setting an appropriate target vector.

**imu.py – Bria**

* What serial port does the IMU is connect on?
* What baudrate does the IMU runs on?
  + We can probably set both to what we want.
* Determine what data the IMU should send and what to look for in the serial output.

**motors.py – Brenden**

* Ensure PWM is a viable choice.
* Do we need another signal to toggle direction?
* Figure out how to convert from torque (and rpm) to duty cycle.
* Lots of testing.

**camera.py - Grace**

* Write a function to get a picture.
* Write a function to determine target location from picture.
* Write a function to translate target position to relative attitude.
* We probably do not need to run the camera very fast as long as the target isn’t moving.
  + Take a picture every second instead of 20 fps.
  + If this takes too long and blocks the main loop, we will have to run this on a background thread.
* We also would need to develop the IR sensor further.

**user.py – Joselyn (And Gui.html /.js /.css)**

* Most of user.py is set up, we just need to merge our software.
* What variables are we sharing with the GUI? *Defined in user.py SharedDataPackage* class
* What commands are we expecting from the GUI? *Defined in main.py processCommands()*
* Test backend server on rPi (Works on windows, but have not tested Unix systems)
* Continue front end development and merging.