Auto-RC Car Documentation 2025

Concepts

Pulse Width Modulation

Pulse Width Modulation, or PWM, is the act of sending short signals in a very quick succession. This creates an overtime average voltage that is used to communicate with technology. Think of it like applying throttle to an engine at a certain rate to get a certain average speed. There are three attributes to a PWM signal: The frequency, the duty, and the millisecond.

The frequency is how often the signal is sent. For example, a 50 Hz signal is sent 50 times a second. The duty Percent (duty %) is how much of the signal is sent. For example, a 50 Hz signal has a period of 20 milliseconds. If this signal had a duty of 50%, it would be active for 10 milliseconds. A duty of 100% would be a constantly active voltage of your given amplitude.

Now paying attention to the Milliseconds (Ms). This is what the hardware actually reads. They will have an operational frequency range but will respond to the actual active Ms of the signal.

Programming

PIGPIO

PIGPIO is the python library used to create and manage Raspberry Pi's GPIO Pins by way of a Python script.

Common Uses Involved in AC Car

Setup:

Start by importing the PIGPIO Library to your project. This is done with an import statement at the top of your program.

```
import pigpio
import pygame as pg
import subprocess
import PiRecording as Pir
import time
import threading
```

Start integrating PIGPIO by evaluating your pins for usage and the frequency you want to operate at.

```
#PWM Variables
pin1 = 12 # GPIO Pin Number
pin2 = 13 # ^
Freq = 50 # in Hz

#REC Light Variables
pinREC = 25
```

The AutoRC uses GPIO Pins 12,13, and 25.

This can be done by setting up integers or floats for your desired values. These pins will relate specifically to the numbered GPIO pins.

Then initialize your Pi as a PIGPIO Pi object. This will be a digital clone of your Pi's GPIO pins; it is how you will control your Pi.

```
ensure_pigpiod_running()
time.sleep(1)
pi = pigpio.pi()
if not pi.connected:
    raise RuntimeError("Failed to connect to pigpio daemon.")
```

However, the PIGPIO Daemon (PIGPIOD) must be running first. So, a function similar to `ensure_pigpiod_running()` must be called.

PIGPIOD is the daemon that allows access to the Pi's hardware that lets the GPIO pins be used by a program. The issue with it is that PIGPIOD is booted in the terminal and needs to be turned on every time the Pi is turned off and on again, requiring a visual environment to access and command the Pi to do so.
So to get around this, we had the AutoRC Main Script boot PIGPIOD itself using the subprocess library and help of ChatGPT to create a function that would be called and make sure that PIGPIOD is running before anything with PIGPIO can be used.

```
def ensure_pigpiod_running():
    try:
        result = subprocess.run(['pgrep', 'pigpiod'], stdout=subprocess.PIPE,
stderr=subprocess.PIPE)

        if result.returncode != 0:
            print("pigpiod is not running. Starting it now...")
            subprocess.run(['sudo', 'pigpiod'], check=True)
        else:
            print("pigpiod is already running.")
    except Exception as e:
        print(f"Error ensuring pigpiod is running: {e}")
```

```
ensure_pigpiod_running()
time.sleep(1)
pi = pigpio.pi()
if not pi.connected:
    raise RuntimeError("Failed to connect to pigpio daemon.")
```

Though it requires the subprocess library:

```
import pigpio
import pygame as pg
import subprocess
import PiRecording as Pir
import time
import threading
```

As well always call `time.sleep(1)` afterwards to have the code wait a second for the Daemon to boot.

Now, in the case of the AutoRC, we would initialize our PWM's Frequency and their Scale for precision. These PWM's start at a duty % of 0. Which is no signal.

```
pi.set_PWM_range(pin1,1000) #Scales the Range of the Steering PWM Function
pi.set_PWM_frequency(pin1, Freq)
pi.set_PWM_range(pin2,1000) #Scales the Range of the Throttle PWM Function
pi.set_PWM_frequency(pin2, Freq)
```

Unfortunately, PIPGIO does not let you pick any frequency. You may input any frequency, and it will be rounded to the closest available frequency. Luckily, for AutoRC, 50Hz is one of the available options. More can be found in the online documentation of PIGPIO.

Usage:

We would then call a function in our main loop that would change the duty % with the inputs of the controller. Using PIGPIO's most important function, `set_PWM_dutycycle(int)`, which can do live updates to the duty % of a pin's PWM.

```
def updateControlDuty() : #Upates teh steering based on the desired ms output
calculated
    global ControlMsOut,Freq
    Duty = ControlMsOut / (1/pi.get_PWM_frequency(pin1) * 1000)
    pi.set_PWM_dutycycle(pin1, (Duty*1000 ) ) # this is what changes the PWM
duty, mathed to the given Ms output.
    #print(Duty * 1000)
```

This is where the scale set earlier would come in. The set scale sets the max value of a scale. So that if the scale set were to 100, `set_PWM_dutycycle` would have 0 as 0% and 100 as 100%, if it were 500, an input value of 500 would be 100%.

We will use a scale of 1000. Working at 50 Ms allows us to work at about 100 microseconds or 0.1 Ms to have more precise control over the hardware.

As for pins that do not use PWM and instead work on a binary signal, they can be used in fashion such as:

```
def startup_signal(): #Siqi made this signal to tell when the car is ready
    for _ in range(3):
        pi.write(pinREC, 1)
        time.sleep(0.5)
        pi.write(pinREC, 0)
        time.sleep(0.5)
```

Using `pi.write`, you can set the pin state as 1 (on) or 0 (off) at about 3.3 Volts.

Closing:

At the end of a script, turn off all the pins involved to make sure that they don't stay on past the end of the script. Since PIGPIOD is a daemon (a separate program) your set PWMs or pin states would continue after the script. Disable them to neutralize your Pi using the following:

```
#Closing Commands, turns off PWMS,
pg.quit()
pi.set_PWM_dutycycle(pin1,  0)
pi.set_PWM_dutycycle(pin2,  0)
pi.set_mode(pinREC, 0)
if(camConnected) :

    threading.Thread(target=Pir.CameraEnd()).start() #turn off camera
```

Online Library

If you want full documentation on PIGPIO, please check out their online library: https://abyz.me.uk/rpi/pigpio/ .

PiCamera2

PiCamera2 is the library used to control a camera module via Raspberry Pi.

For AutoRC, the recorded functionality was adapted from a ChatGPT Script that was appropriated into a more functional library.
Starting off:

```
from picamera2 import Picamera2, Preview
import picamera2.encoders
import time
from datetime import datetime
import os
```

These are the libraries used to create our recording library.

```
def CameraSetupHandler() :
    camInit()
    global SAVE_DIRECTORY, camera, camera_config, output_file
    if not os.path.exists(SAVE_DIRECTORY):
        os.makedirs(SAVE_DIRECTORY)
        print(f"Created directory: {SAVE_DIRECTORY}")


    output_file = generate_filename()
    camera.start()
    time.sleep(0.5)
```

This function must be called first. It initializes the camera and the save directory.

```
def start_recording(output_file):
    """Starts recording to the specified file."""
    generate_filename()
    print(f"Starting recording: {output_file}")
    encoder = picamera2.encoders.H264Encoder() # Use the H264 encoder
    camera.start_recording( encoder ,output_file)  # Pass encoder and output file

def stop_recording():
    """Stops recording."""
    print("Stopping recording")
    camera.stop_recording()
```

These functions do as their namesake. They start and stop the recording.

```
def generate_filename():
    """Generates a filename based on the current date and time, including the
save directory."""
    timestamp = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")  # e.g., "2025-01-
24_15-45-30"
    return os.path.join(SAVE_DIRECTORY, f"recording_{timestamp}.h264")
```

This function creates the file names by date and time of their recording.

```
def camInit() :
    global camera, camera_config
    camera = Picamera2()

# Configure camera settings
    camera_config = camera.create_video_configuration(main={"size": (640, 480),
"format": 'YUV420'})
    #camera.sensor_resolution = (640,480)
    camera.configure(camera_config)
```

This function is where the camera is identified and configured. Without proper protection and there is not an available camera, it will create an error. It also configures the resolution of the recorded video. Here that resolution is minimized to reduce the processing load on the Pi.

To avoid the prementioned error, try something along the lines of this:

```
camConnected = False #This variable is used to manage camera connection adn avoid errors, if there is a camera then camera functions will be called. if not, skip them. Avoids the intrinsic error of the picamera2 library and there not being a camera

try:
    threading.Thread(target=Pir.CameraSetupHandler()).start() #Multithread the Functions relating to the camera
    camConnected = True


except :
    if(camConnected != True) :
        print("No Camera Connected")
        camConnected = False
```

Where these lines of code attempt to call the setup function and through that the initialization function. Should an error occur, Camera not found), it will set camConnected to false, then ignoring all calls of PiRecording and therefore avoiding any error prone instructions.

```
def CameraFlipFlopHandler() : # Starts recording using the GPT generated library
    global Recording

    if  not Recording :
        if(camConnected) : #only do it when cam is connected, avoids errors
            threading.Thread(target=Pir.start_recording(Pir.output_file)).start()

            print("Started Recording")

    else :
        if(camConnected) :
            threading.Thread(target=Pir.stop_recording()).start()
            print("Stopped Recording")

    Recording = not Recording
```

Online Library

The online library for PiCamera2 can be found here:
https://datasheets.raspberrypi.com/camera/picamera2-manual.pdf

If any errors occur with your code, please consult the manual to understand your problem better.

PyGame And Controller Usage

PyGame is the library used to get controller and keyboard inputs in live time. Usually used for video games, it has control of updates and key input as well as the option of a visual display in a graphical environment. Using this library, AutoRC manages both Controller and Keyboard inputs to control PWM.

Setup

PyGame can be set up by first importing then initializing it and the wanted elements.

```
import pigpio
import pygame as pg
import subprocess
import PiRecording as Pir
import time
import threading
```

Then to start using pygame:

```
pg.init() #Initialzie the pygame library
pg.joystick. init() #Inititalize the pyganme controller reader
#pg.display.set_mode((300,300))
#pg.display.set_caption("Drivin Controller")
Clock = pg.time.Clock() #Needed to add an update rate to the program to reduce
CPU oad
if (ControllerConectionCheck()) : # Initialzies the connection with a controller
thats connected.
    joystick = pg.joystick.Joystick(0)
    joystick.init()
```

Pg – initializes and its subsidiary functions.

Joystick – initializes controller functionality.

Display – the visual window used during development.

Clock – an applied update rate to reduce CPU load of the program. Infinite updates per second mean infinite CPU Usage.

After setting up you are free to work with your initialized elements.

Input

```
def getKeyboardInputs() : # manages the control axis and throttle axis based off
the arrowkey inputs

    global ControlAxis, ThrottleAxis, ThrottleKey

    keys = pg.key.get_pressed() # gets input keys

    a = 0
    b = 0


    if (keys[pg.K_LEFT]) : # left key, etc
        #print("left is Pressed")
        a -= 1
    if ((keys[pg.K_RIGHT]) ) :
        #print("Right is Pressed")
        a += 1




    if (keys[pg.K_DOWN]) :
        #print("left is Pressed")
        b -= 1
    if ((keys[pg.K_UP]) ) :
        #print("Right is Pressed")
        b += 1

    if(keys[pg.K_SPACE]) :
        ThrottleKey = 1

    ControlAxis = a # NEED TO BE HERE TO RESET, WILL NOT RESET IN FUNCTION
    ThrottleAxis = b
```

This gets our keyboard input. PyGame keeps a list of all keys pressed at one time. Checking if any matches the ones we want, when we do get a desired input, we offset either the Control Axis used for steering or the Throttle Axis which is used for driving.

```
def getControllerInputs() : #obtains controller inputs with pygame
    global DEADZONE, ControlAxis, ThrottleAxis,ThrottleKey,
ThrottleMsCentered,ThrottleMsOut
    ControlAxis = apply_deadzone(joystick.get_axis(0),DEADZONE)
    ThrottleAxis = joystick.get_axis(4)/2  +.5 #gets number here
    #print(ThrottleAxis)

    if(handle_one_shot_button(4)): #Start recording
        CameraFlipFlopHandler()

    dpad = joystick.get_hat(0) # Handles dpad input for transmission usage
    if(dpad[1] == 1) :
        ThrottleKey = 1

    elif(dpad[1] == -1) :
        ThrottleKey = -0.625
    elif(dpad[0] == 1 or dpad[0] == -1):
        ThrottleKey = 0

    a = ThrottleMsOut/(ThrottleMsCentered + ThrottleMsRange) #Controls rumble for
fun
    #print(a)
    if(a > .85):
        joystick.rumble(0,a,0)
    elif(a < .83) :
        joystick.rumble(a,0,0)
    else :
        joystick.stop_rumble()
```

This is how we get input from our controller. AutoRC applies the left joystick's horizontal axis to the Control Axis, meaning that as the stick moves towards the right, the axis increases to 1, to the left, -1.

Then there is getting input from the D-Pad, or Hat, or Cross Button. It is stored as a 2-part array. Where 0 is the horizontal, and 1 is the vertical. When up is pressed (dpad[1] == 1), then set our throttle key to 1.

How are these axes applied?

```
def SteeringHandler() : # Calculates the desired Steering Ms output basd of
collected data and the user's input
    global ControlMsCentered, ControlAxis, ControlMsRange, ControlMsOut
    ControlMsOut = ControlMsCentered + ControlMsRange * ControlAxis
    #print(ControlAxis)
    #print(ControlMsOut)
def ThrottleHandler(): # ^^^ but for thottle
    global ThrottleMsCentered, ThrottleAxis, ThrottleMsRange,ThrottleMsOut,
ThrottleKey
    ThrottleMsOut = ThrottleMsCentered + ThrottleMsRange * ThrottleAxis *
ThrottleKey
    #print(ThrottleMsOut)
```

AutoRC used the Axis of -1 to 1 to apply them to some formulas. Simply put they are regressions of what we found when testing hardware, with set goal positions and limitations. Control Axis works around the midpoint of the servo to calculate the Ms signal needed to output at an angle of -23 to 23 degrees to manage the steering. While throttle sits between forwards and reverse for the drive Ms signal. The throttle key is being used to invert the direction of the axis, allowing the throttle signal to change direction.

```
def updateControlDuty() : #Upates teh steering based on the desired ms output
calculated
    global ControlMsOut,Freq
    Duty = ControlMsOut / (1/pi.get_PWM_frequency(pin1) * 1000)
    pi.set_PWM_dutycycle(pin1, (Duty*1000 ) ) # this is what changes the PWM
duty, mathed to the given Ms output.
    #print(Duty * 1000)
```

Then we use the Ms signal to calculate the duty % needed to get that desired output from the hardware.

Online Library

The online library for PyGame can be found: https://www.pygame.org/docs/
A better understanding of the available components can be found in the online documentation.

Hardware

Servo

The servo turns a Ms signal to a desired held angle. In AutoRC, the signal is held at angle x to keep the car straight, then the signal is changed to be at angle y or -y to control the steering.

Speed Controller

It is a programmed relay. It has different settings based on how its jumper is physically configured. Currently it is set in its Forward/Brake/Reverse. Where it has a neutral range, a forward range, and a reverse range. We set the signal to be within these ranges based off the user's input.

Controller/Gamepad

It is an Xbox One Controller.

Conclusion

This documentation should have provided an understanding of how AutoRC uses its libraries to:

Control GPIO Pins for PWM Control.

Using an object such as a keyboard or controller for controlling the car with PyGame.

Recording using PiCamera2 or the PiRecording Script.

If you are unfamiliar with these libraries, feel free to explore their online documentation libraries attached in each section.

Enjoy your time with AutoRC.