**Self-Driving Car**

In this project, you are going to implement a self-driving car by incorporating elements from the previous labs. You are also going to use a new board (i.e., PCA9695) for driving servo and DC motors. PCA9695 uses I2C to receive commands from Hi-Five board to generate PWM signal for the servo- and DC-motors. The final project weight 15% of your total course grade.

A picture containing mower, table, motorcycle

Description automatically generated

4- Camera

1- Hi-Five

2- Raspberry Pi

3- PWM I2C controller (PCA9695)

Figure 1: Self-driving car prototype

A picture containing engine

Description automatically generated

Figure 2: Hi-five, Pi, and PWM I2C controller boards

**A circuit board

Description automatically generated**

5- DC Motor

7- ESC

6- Servo Motor

Figure 3: DC motor, Servo motor, and ESC on the car

Figure 1, Figure 2, and Figure 3 show the car prototype that you are going to use. It has seven main components:

1- Hi-five board

2- Pi board

3- Motor driver (PCA9695)

4- Camera

5- DC motor

6- Servo motor

7- Electronic Speed Controller (ESC)

Your goal in this project is to first use the Hi-five board to send I2C commands to PCA9695 to drive the servo motor (for steering) and DC motors (for moving forward and backwards) (**Milestone 1**). Then connect Pi to Hi-five board using UART. This is to set up a connection between the two boards for sending steering commands from the Pi to the Hi-five board. (**Milestone 2**). Lastly, you are going to put the camera in the loop, run the DNN inference engine on the Pi, and close the loop by sending the steering commands from the Pi to the Hi-five board (using UART) and then to the motors (using PWM I2C controller) (**Milestone 3**).

Optionally, you can develop this project further for extra credit. You can find more information about the extra credit under **Milestone 4** section. The maximum extra points that you can get is 20% of the total final project grade (or 3% of your final grade!). Before you start working on any extra credit project, you should discuss it first with your GTA to evaluate its feasibility and also the number of extra points that you can get.

You can check the deadline for each milestone in Table 4.

|  |  |  |
| --- | --- | --- |
| Milestone | Description | Deadline |
| 1 | Drive motors using PWM | Nov 6th |
| 2 | Connect Hi-five and Pi | Nov 13th |
| 3 | Self driving car | Nov 24th |
| 4 | Extra credit (open ended) | Nov 24th |

Table 4: Deadline for each milestone

You should work on the final project in groups of two. Please fill in the excel sheet in the "Final Project" channel in the Teams with your team info. Here is the link to the excel sheet: [Final Project Groups.xlsx](https://teams.microsoft.com/l/file/8596AC05-F844-46D7-86D5-EF51884A537B?tenantId=3c176536-afe6-43f5-b966-36feabbe3c1a&fileType=xlsx&objectUrl=https%3A%2F%2Fkansas.sharepoint.com%2Fteams%2F4209-110182020Fall-EECS388EmbeddedSystemsLEC%2FShared%20Documents%2FFinal%20Project%2FFinal%20project%20groups.xlsx&baseUrl=https%3A%2F%2Fkansas.sharepoint.com%2Fteams%2F4209-110182020Fall-EECS388EmbeddedSystemsLEC&serviceName=teams&threadId=19:7d0369721ed24474bfff2f87b8cc800f@thread.tacv2&groupId=c6dfd23a-6b64-4263-b92a-9b6116864c78)

# Getting started

Unlike other labs, we have not created a branch for the final project and not pushed the required code and libraries into your private gitlab repository. Instead, there is a central gitlab repository that you need to clone and copy necessary files from there into your project. Here are the instructions for getting started on the final project:

mkdir ~/eecs388-final-project-release

cd ~/eecs388-final-project-release

git clone <http://git.eecs.ku.edu/eecs388-internal/eecs388_final_project_release.git>

After executing the above commands you have a local copy of the final project files under ~/eecs388-final-project-release directory. Note that we gradually release files for each milestone and as of October 26th, you can only find files for milestone 1 in the above repository.

You should create a

You should create a branch called “final\_project” in your local repository and copy the necessary files from the release repository there before start working on each milestone. For example, for milestone 1 you should do the following:

# change directory to you git local repo

cd <your local git repo>

# add and commit all the left over changes from the previous labs (if there is any)

git add .

git commit -m “leftover from previous labs”

git push

# create a new branch for final project

git checkout -b final\_project

# copy milestone 1 files from the release repo

cp -r ~/eecs388-final-project-release/milestone1 .

# add commit and push the files to your remote repo

git add milestone1

git commit -m “added ml1 files”

git push -u origin final\_project

For milestone 2, 3, and 4 you do not need to create a new folder and only need to pull the updated files from the release repo and copy them into your local repository:

cd ~/eecs388-final-project-release

git pull

# copy milestone files from the release repo; e.g, for milestone 2:

cp -r ~/eecs388-final-project-release/milestone2 .

# add commit and push the files to your remote repo

git add milestone2

git commit -m “added ml2 files”

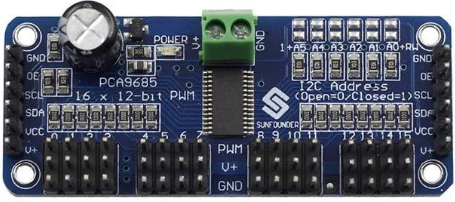
# Milestone 1

In this milestone you configure the PWM controller and use Hi-five board to drive the servo and DC motor.

You need to complete 6 tasks highlighted by green color to finish milestone 1.

Your system will have the raspberry pi feeding output from its DNN inferencing (images provided by the camera) every 50 ms to the HiFive board. The HiFive board will then use this information to control the steering of the system.

In order to accomplish this steering and motor control, the system will require stronger servos than previously used in the lab and subsequently needs an external power supply along with a dedicated servo driver. The PCA9685 is a 16 channel 12 bit PWM servo driver which uses the I2C serial communication protocol. Please take a look at Slides #37 - #39 of Lecture note-8 to refresh your I2C knowledge. In PCA9685 nomenclature, the output channels are called LED channels (because one of the main use-cases of the board is to drive LEDs). We use channel 0 and channel 1 (i.e., LED0 and LED1) for driving servo motor and DC motor, respectively.



SDA and SCL

Hi-Five GPIO

LED 16

LED 0

LED 1

PWD to servo and DC motors

On the HiFive board, the I2C core is from a 3rd party provider called [OpenCores](https://opencores.org/projects/i2c). If you look in the documentation for the Hi-five board, under I2C, you will be redirected to their site. Normally this would then require you to build your I2C interaction from scratch, keeping in mind the instructions provided by both open core and PCA9685. However, the framework used by HiFive, freedom-e-sdk (or "metal library"), includes an I2C library under the metal folder.

First, in the c\_cpp\_properties.json contained in the .vscode folder, confirm if your include path contains the following line:

[user specific info]/.platformio/packages/framework-freedom-e-sdk/freedom-metal"

This line adds the metal library to your include path - allowing you to easily include the metal library into your code by simply adding this line to the top of your code.

#include "metal/i2c.h"

## Part 0: connect the I2C board to Hi-five board

This step is already done for you.

## Part 1: Setting up the I2C via the metal library

The metal library requires three key elements to function properly, a pointer to its instance, and two u\_int8 arrays, which will be used in the reading and writing. While the length of the read array only needs to be 1, the length of the write array should be 5 (this will be explained later). The following code accomplishes this.

struct metal\_i2c \*i2c;

uint8\_t bufWrite[5];

uint8\_t bufRead[1];

The following line gets a handle at device index 0 and assigns it to the I2C pointer:

i2c = metal\_i2c\_get\_device(0);

If I2C == NULL, then the connection to the I2C device, the PCA9685, was unsuccessful.

Finally, we need to initialize the I2C module in the HiFive board as the master with the following line. We are using a baud rate of 100000:

metal\_i2c\_init(i2c,I2C\_BAUDRATE,METAL\_I2C\_MASTER);

This concludes the I2C setup. From here, we will use the write and transfer methods to set/read the values at the various registers within the PCA9685. However, first, we must configure the board with a few key configurations; otherwise, the driver will not work.

## Part 2: Configuring the PCA9685

According to the datasheet found at <http://wiki.sunfounder.cc/images/e/ea/PCA9685_datasheet.pdf>, the address for a single PCA9685 defaults at 0x40. For ease of use, define this in your h file along with the following:

//Setup for PCA9685

#define PCA9685\_I2C\_ADDRESS 0x40

#define PCA9685\_MODE1 0x00      /\*\*< Mode Register 1 \*/

#define PCA9685\_LED0\_ON\_L 0x06  /\*\*< LED0 on tick, low byte\*/

#define PCA9685\_PRESCALE 0xFE     /\*\*< Prescaler for PWM output frequency \*/

// MODE1 bits

#define MODE1\_SLEEP 0x10   /\*\*< Low power mode. Oscillator off \*/

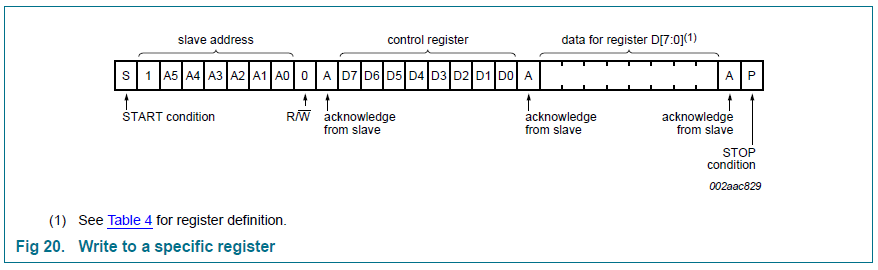
#define MODE1\_AI 0x20      /\*\*< Auto-Increment enabled \*/

#define MODE1\_EXTCLK 0x40  /\*\*< Use EXTCLK pin clock \*/

#define MODE1\_RESTART 0x80 /\*\*< Restart enabled \*/

#define FREQUENCY\_OSCILLATOR 25000000 /\*\*< Int. osc. frequency in datasheet \*/

We will begin by writing a reset command to the mode1 register. The interaction model between the HiFive board and the PCA9695 is as follows (I2C protocol):



The parameters of the write function are as follows:

/\*! @brief Perform a I2C write.

 \* @param i2c The handle for the I2C device to perform the write operation.

 \* @param addr The I2C slave address for the write operation.

 \* @param len The number of bytes to transfer.

 \* @param buf The buffer to send over the I2C bus. Must be len bytes long.

 \* @param stop\_bit Enable / Disable STOP condition.

 \* @return 0 if the write succeeds.

 \*/

inline int metal\_i2c\_write(struct metal\_i2c \*i2c, unsigned int addr,

unsigned int len, unsigned char buf[],

metal\_i2c\_stop\_bit\_t stop\_bit) {

return i2c->vtable->write(i2c, addr, len, buf, stop\_bit);

}

We start by writing a reset command to the PCA9695, which looks like the following. The final parameter for writing is either METAL\_I2C\_STOP\_DISABLE or METAL\_I2C\_STOP\_ENABLE. It is a value defined that sets the stop condition in the I2C protocol. If you ever send a single write or read command to the PCA9685, you would use METAL\_I2C\_STOP\_ENABLE as the final parameter. You would use METAL\_I2C\_STOP\_ENABLE as the final parameter if you plan to do multiple write/reads at the same time, making sure the final write/read has the stop condition as METAL\_I2C\_STOP\_ENABLE.

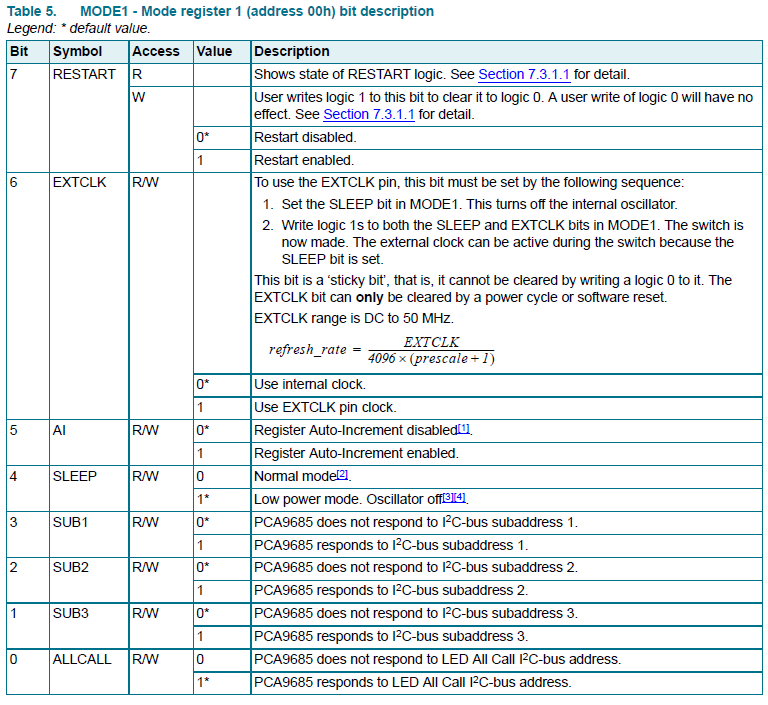
\_Bool success;

bufWrite[0] = PCA9685\_MODE1;

bufWrite[1] = MODE1\_RESTART;

success = metal\_i2c\_write(i2c,PCA9685\_I2C\_ADDRESS,2,bufWrite,METAL\_I2C\_STOP\_DISABLE);//resets the register

We will now send several commands in order to configure the MODE1 register. For reference, the configuration is defined as follows:



The following commands will successfully configure the PCA9695:

bufWrite[0] = PCA9685\_MODE1;

success = metal\_i2c\_transfer(i2c,PCA9685\_I2C\_ADDRESS,bufWrite,1,bufRead,1);//initial read

oldMode = bufRead[0];

newMode = (oldMode & ~MODE1\_RESTART) | MODE1\_SLEEP;

bufWrite[0] = PCA9685\_MODE1;

bufWrite[1] = newMode;

success = metal\_i2c\_write(i2c,PCA9685\_I2C\_ADDRESS,2,bufWrite,METAL\_I2C\_STOP\_DISABLE);//sleep

bufWrite[0] = PCA9685\_PRESCALE;

bufWrite[1] = 0x79;

success = metal\_i2c\_write(i2c,PCA9685\_I2C\_ADDRESS,2,bufWrite,METAL\_I2C\_STOP\_DISABLE);//sets prescale

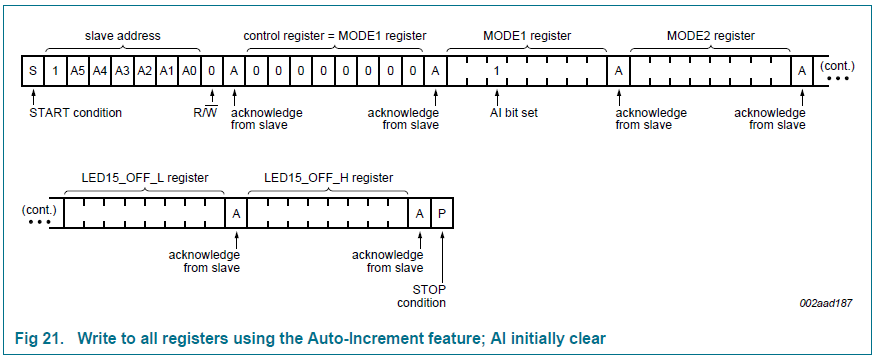
bufWrite[0] = PCA9685\_MODE1;

bufWrite[1] = 0x01 | MODE1\_AI | MODE1\_RESTART;

success = metal\_i2c\_write(i2c,PCA9685\_I2C\_ADDRESS,2,bufWrite,METAL\_I2C\_STOP\_DISABLE);//awake

delay(100);

It is important to know that MODE1\_AI stands for auto-increment. This means that each subsequent write from a defined starting address will automatically write to the next register. Writing under these conditions is shown below:



## Part 3: Using the transfer method to control the PCA9695 (Servo control)

The transfer method in the I2C library allows us to send an arbitrary amount of writes and reads with a single command; this is why we specified our write array to have a length of 5 The parameters of transfer are shown below:

/\*! @brief Performs back to back I2C write and read operations.

 \* @param i2c The handle for the I2C device to perform the transfer operation.

 \* @param addr The I2C slave address for the transfer operation.

 \* @param txbuf The data buffer to be transmitted over I2C bus.

 \* @param txlen The number of bytes to write over I2C.

 \* @param rxbuf The buffer to store data received over I2C bus.

 \* @param rxlen The number of bytes to read over I2C.

 \* @return 0 if the transfer succeeds.

 \*/

inline int metal\_i2c\_transfer(struct metal\_i2c \*i2c, unsigned int addr,

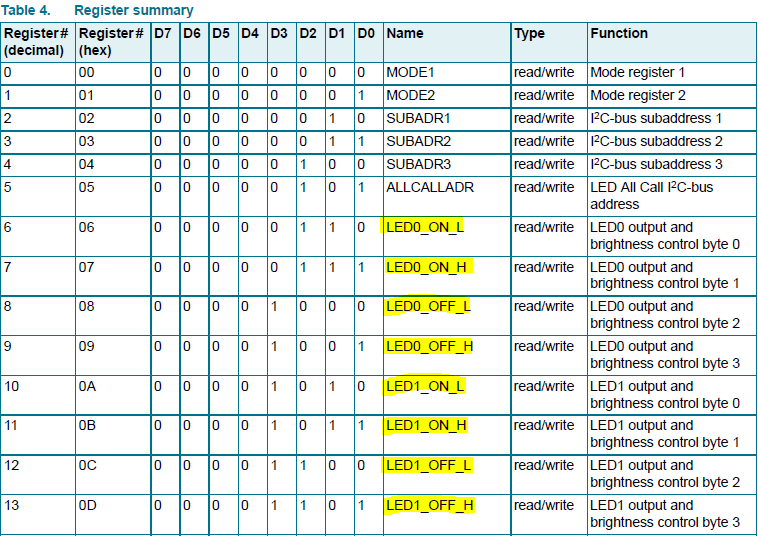
unsigned char txbuf[], unsigned int txlen,

unsigned char rxbuf[], unsigned int rxlen) {

return i2c->vtable->transfer(i2c, addr, txbuf, txlen, rxbuf, rxlen);

}

As we explained earlier, in PCA9685 nomenclature, the output PWM channels are called LEDs. We use only LED0 and LED1 to drive the servo and DC motors. Here are the register addresses that we will edit:



Task 1: The breakup function

When looking at the register summary, note that each LEDn has 4 components, ON\_L, ON\_H, OFF\_L, and OFF\_H. This is because 4096 is a 12-bit number, so it must be broken up to fit into two 8-bit numbers, as I2C writes a byte at a time. L refers to the lower 8 bits, and H refers to the higher 8 bits. Your task will be to define a function that takes an integer and breaks it down into the high 8-bits and low 8-bits, assigning the references high and low to these values:

void breakup(int bigNum, uint8\_t\* low, uint8\_t\* high){

    //Put task 1 code here

}

Example usage:

uint8\_t = variable1

uint8\_t = variable2

breakup(2000,&variable1,&variable2);

Variable1 -> low 8 bits of 2000

Variable2 -> high 8 bits of 2000

Controlling the servo and motor Electronic Speed Controller (ESC) with PWM is very similar to how we did so in the actuator Lab. However, in this case, we will be converting values ranging from 0 to 20 ms to 0 to 4095 cycles.

Using the getServoCycle function will convert an angle from –45 to 45 and return the corresponding value to write to the LED1\_OFF\_L and LED1\_OFF\_H after using your previously defined breakup function:

//A function used to quickly map [-45,45] to [155,355]

int map(int angle,int lowIn, int highIn, int lowOut, int highOut{

int mapped = lowOut + (((float)highOut-lowOut)/((float)highIn-lowIn))\*(angle-lowIn);

return mapped;

}

//only provide an angle ranging from -45 to 45

//Sending values outside this range will cause

//unexpected behavior

int getServoCycle(int angle){

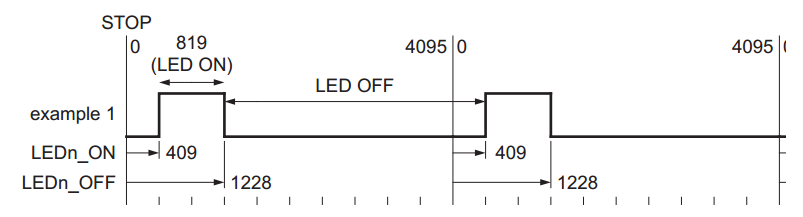
int cycle\_value;

cycle\_value = map(angle, -45, 45, SERVOMIN, SERVOMAX);

return cycle\_value;

}

After the conversion, the PWM relationship can be viewed below. Please note that we will set LEDn\_ON time to zero for this project as it’s the simplest; however, it is technically arbitrary where the On time is, so long that the Off time is offset accordingly. The servos duty cycle ranges from a length of 150/4095 cycles to 600/4095 cycles; however, due to the constraints of the system, this range is closer to 155/4095 cycles to 355/4095 cycles. The getCycleLength function will account for this constraint.



Task 2: The Steering Function

Using the getServoCycle, breakup, and transfer functions, you will implement the following function in order to control the steering of the car by writing to alter the PWM of the servo motor. Angle is between the range -45 to 45.

void steering(int angle){

/\*

Write Task 2 code here

\*/

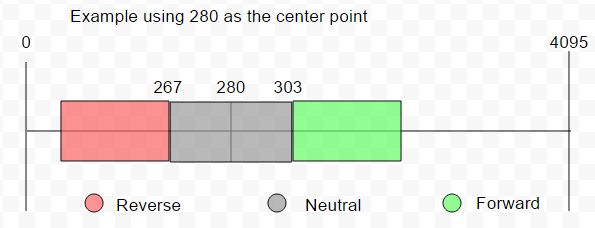
}

Example usage: steering(0); -> driving angle forward

## Part 4: Using the transfer method to control the PCA9695 (Motor control)

Controlling the motor is the exact same as controlling the servo with one key exception; the motor is connected to an ESC, which has its own rules for usage. For starters, It must be configured before it will move. This process is relatively simple; you must write an arbitrary PWM in cycles for the ESC to calibrate around. This is accompanied by a beep (Second beep after turning on the switch within the bot) signaling calibration being completed. From this arbitrary point, there is a dead band of plus or minus 23 cycle lengths, sending a PWM outside this dead band will make the motor drive forward or backwards respectively. This relationship is illustrated below:

**Be Advised: these RC cars are designed to drive at very high speeds. Keep the values around 305 and 265, and be sure the robot is propped up so it can't suddenly drive away!**



Task 3: Calibrating and defining the stop function

You will implement the following function in order to stop the wheels from moving. This will also cause the ESC to calibrate itself when first called with a 2 second delay. This will be accomplished by setting the LED0\_OFF to 280

void stopMotor(){

/\*

Write Task 3 code here

\*/

}

Example Use: stopMotor(); -> sets LED0\_Off to 280

Task 4: Drive Forward function

You will implement the following function in order to make the wheels drive forward. Further details will be provided below regarding the parameters and results.

void driveForward(uint8\_t speedFlag){

/\*

Write Task 4 code here

\*/

}

The given speedFlag will alter the motor speed as follows:

speedFlag = 1 -> value to breakup = 303

speedFlag = 2 -> value to breakup = 305 (optional)

speedFlag = 3 -> value to breakup = 307 (optional)

Example Use: driveForward(3); -> sets LED0\_Off to 307

Task 5: Drive Reverse Function

You will implement the following function in order to make the wheels drive in reverse. Further details will be provided below regarding the parameters and results.

void driveReverse(uint8\_t speedFlag){

/\*

Write Task 5 code here

\*/

}

The given speedFlag will alter the motor speed as follows:

speedFlag = 1 -> value to breakup = 267

speedFlag = 2 -> value to breakup = 265 (optional)

speedFlag = 3 -> value to breakup = 263 (optional)

Example Use: driveReverse(2); -> sets LED0\_Off to 265

## Part 5: Putting it all together7

Task 6: Fully controlling the PCA9685

Using the all of your implemented functions, perform the following sequence of actions:

1. Calibrate the motors
2. Set the steering heading to 0 degrees (wait for 2 seconds)
3. Drive Forward (wait for 2 seconds)
4. Change the steering heading to 20 degrees (wait for 2 seconds)
5. Stop the car (wait for 2 seconds)
6. Drive in reverse (wait for 2 seconds)
7. Set steering heading to 0 degrees (wait for 2 seconds)
8. Stop the motors

# Milestone 2

## Part 1: Board-to-Board Communication

In this part, you will establish UART based communication channels between the Pi 4 and the HiFive 1 boards.

### **Part 1.1: Setup the UART connections**

In this part, we will connect the HiFive1 and the Raspberry Pi 4 boards via two UART channels.

The Pi has 4 UARTs, and we will use two of them (uart2 and uart3). Confirm that the following two lines exist at the end of the /boot/config.txt file to enable uart2 and uart3.

dtoverlay=uart2,115200

dtoverlay=uart3,115200

After rebooting the system, /dev/ttyAMA1 and /dev/ttyAMA2 will be created.



Connect HiFive's UART1 RX (pin7) to Raspberry Pi 4's UART2 TX (pin 27). This is the main communication line between the Pi and the HiFive1. From the Pi, you can access the channel via /dev/ttyAMA1.

For debugging of HiFive 1, connect HiFive1's UART0 TX (pin1) to Pi 4's UART3 RX (pin 29). From the Pi 4, it can be accessed via /dev/ttyAMA2.

In summary, you will be able to access the following two files from the Pi 4.

/dev/ttyAMA1 Pi 4 → HiFive1: Send steering angle to HiFive1 (uart1).

/dev/ttyAMA2 HiFive1 → Pi 4: Receive HiFive1’s console (uart0) output

## Part 2: Programming the HiFive1

In this part of the lab, you will program the HiFive1 to receive data from the Pi 4.

**On your PC** (not Pi 4), you should develop the project based on the code in folder **Hifive1.**

Your task is to receive the data from HiFive1's UART1 channel and send the received data to UART0 channel. The following is a rough pseudo code of the task.

while (1)

{

if (is UART1 ready?)

{

data = read from UART1.

print data to UART0.

}

}

To implement the task, you may need to use the provided serial API shown in the following. Note that devid is 0 to access UART0, while it is 1 to access UART1.

void ser\_setup(int devid);

int ser\_isready(int devid);

void ser\_write(int devid, char c);

void ser\_printline(int devid, char \*str);

char ser\_read(int devid);

int ser\_readline(int devid, int n, char \*str);

In particular, you may need to use ser\_isready() function to check whether a given UART channel has pending data to read. To better understand what the functions are doing, check eecs388\_lib.h and eecs388\_lib.c files.

int ser\_isready(int devid)

{

uint32\_t regval = \*(volatile uint32\_t \*)(UART\_ADDR(devid) + UART\_IP);

return regval;

}

Once you finish programming the HiFive1, **switch to the Raspberry Pi 4** and open two terminals: one for sending data to the HiFive1, and one for seeing the debug message output from the HiFive1.

Sender’s terminal (term1)

$ screen /dev/ttyAMA1 115200

Debug terminal (term2)

$ screen /dev/ttyAMA2 115200

Now, type any strings on the 'term1'.

If you programmed your HiFive 1 correct, you should see the message coming out from the 'term2' terminal.

## Part 3: Programming the Raspberry Pi 4

Instead of using terminals, you now run a python program on the Pi 4 to communicate with the HiFive1. Your task is to extend the dnn.py from the previous lab to be able to send the steering output to the /dev/ttyAMA1 serial channel. The following **pseudo-code** provides a general idea of the modifications you will need to make to dnn.py:

Open serial connections to /dev/ttyAMA1 and /dev/ttyAMA2

While True:

image = camera.read()

angle = dnn\_inference(image)

**Write ‘angle’ to /dev/ttyAMA1**

Wait\_till\_next\_period()

Close serial connections

To achieve the functionality from above, you need to use Python's pySerial API which can be used by importing the serial package:  
 import serial

With it, you should create two separate serial channels, one for writing to the HiFive1 over /dev/ttyAMA1 and another for debugging over /dev/ttyAMA2. Note that both channels should be opened with the baudrate 115200 bps.

ser1 = serial.Serial(...)

ser2 = serial.Serial(...)

The angles received from the DNN as it processes frames can then be sent to the HiFive1 by using the serial write() function:

ser1.write(...)

However, write() requires a byte value while the angle produced by the DNN is a float32 value, so you will have to convert the angle data in order to send it to the HiFive1.

bytes(degree)

Finally, after all of the frames are processed, the serial connections can be closed by invoking the serial close() function:

ser1.close()

ser2.close()

## Appendix for Milestone 2

GPIO mapping of Pi 4.



Source: [https://learn.pi-supply.com/make/raspberry-pi-4-pinout](https://learn.pi-supply.com/make/raspberry-pi-4-pinout/)

# Milestone 3

You will now combine the work done in the previous two milestones. After completing this milestone the DNN running on the Pi should process the video file “epoch-1.avi” and output prediction angles for each frame, these predictions should be sent to the HiFive board using the UART serial communication configured in Milestone2, and finally that angle should be applied to the steering function implemented in Milestone1.

Once this has been verified and demonstrated to the TA, you should modify the DNN and use the camera with live video frames instead of the video file.

## Part 1: Modifying the HiFive1 UART

You should first add your function implementations from previous projects into the main file `eecs388\_m3.c` located in the release for Milestone 3 and then add in the serial code you wrote for Milestone2 to the dnn\_video.py file.

To use the DNN for steering, you need to implement the while loop in the main function such that the prediction angle for each video frame is used to control the car’s steering. In order to do this, you will need to convert the received value from UART into a useable integer value and feed this into the steering function.

Helpful c functions for the above task `[strncmp](https://en.cppreference.com/w/c/string/byte/strncmp)` and `[sscanf](https://en.cppreference.com/w/cpp/io/c/fscanf)`.



You may also find it helpful to convert the prediction angle from a float32 to a different type before transmitting in the python code.

Final Note: On the HiFive, you should modify your code to read from UART 0 – but on the Raspberry Pi 4, you will still write to the serial address `/dev/ttyAMA1`. Because of this change, connect the HiFive's UART0 RX (pin0) to Raspberry Pi 4's TX (pin 27) and connect HiFive1's UART0 TX (pin1) to Pi 4's RX (pin 29). If you have trouble opening the serial port, reboot the Pi.

## Part 2: Modifying the Pi

Now we will stop simulating and use the camera module on the cars. In the milestone3/Raspberry\_Pi4 directory, there is a new file `dnn\_camera.py` that you will use to perform the DNN inference on camera feed. This file is similar in structure to the `dnn\_video.py` except that instead of opening a video file and processing frame by frame, we open the camera module continuously and process picture by picture.

To use the camera with the Pi, import `[picamera](https://picamera.readthedocs.io/en/release-1.13/index.html)` and `[picamera.array](https://picamera.readthedocs.io/en/release-1.13/api_array.html)`.

The following is the primary difference between the two python files. Here we open the camera, create a 3-dimensional array of size 3 x 640 x 480 (three arrays for each red/blue/green), set the resolution and frame rate of the camera, and the for loop opens the camera for continuous streaming.



Like part 1, you will need to modify the python program `dnn\_camera.py` to transmit the prediction angle via UART. After modification, demo your system (camera input used to steer the car) to the TA.

# Milestone 4 (Extra Credit)

Once the self-driving car is completed, you can obtain extra credit by including more functionality to the self-driving car. This milestone can earn you up to a max of 23% extra credit, giving the possibility of obtaining an extra ~3% in the final class grade. Note: Milestones 1-3 must be done before attempting Milestone 4. Note that you get 3 points extra credit just demoing a mobile self-driving car using the front camera module on the track.

## Adding extra functionality to the Self-Driving car.

Before starting, you must discuss the project you will attempt with your GTA. Your GTA will decide how feasible and how much extra credit you will earn based of its complexity. Below are example projects that could be used for this milestone. You can either choose one of the below projects, or a combination of them. You may also discuss implementing a feature that is not on the list with your GTA.

### **Reverse**

The RC car has built in protection after the device is set to forward to avoid accidentally putting the vehicle in reverse. Because of this, a “double click” is needed to activate reverse. This is done by setting to reverse, back to neutral, and back to reverse. Additionally, you must attempt to simulate the “double click” action as it would be done with an RC controller which includes a gradual change between the neutral and reverse values. (e.g., Instead of writing 280, then 267, you must gradually decrease from 280 to 257). This would earn you 2% EC.

### **Bluetooth Remote Control**

Using the pygame library on the PI and a PS4/Xbox controller to manually drive the car. You should be able to toggle between self-driving and manual driving using the controller. View appendix for helpful links. This would earn you 5% EC.

### **Anti-Collision**

Using LIDAR to avoid collision with objects. Since the camera is only used to inference the angle that is to be used to steer, there is no mechanism to stop the car from collision. Setup the car to use Lidar to stop collision with anything directly in front of the car. Refer to Lab 03 for instructions on correctly connecting the LIDAR sensor to the HiFive board. You must use the Lidar to check if the car gets within 0.75 meters within of hitting something. If this occurs, then the car should stop to avoid collision. You need to make sure that you are only using a single UART (preferably UART 0) for the HiFive and PI Communication before using Lidar, as it also uses UART for communication. The HiFive only has 2 UARTs in total. This would earn you 10% EC.

### **Course Mapping**

Have the path of the vehicle be mapped and saved as a file on the pi. Keeping in mind the car speed, and the steering angle, you will use that information to store information of the path into a file on the PI. You would have to send information about the steering angle as well as the speed being used on the car to the PI through UART, and then add python code to write the information to a file. Additionally, you must include the functionality to read the file again and use the information to feed back the path information to the HiFive. If Implemented correctly, the car should be able to do the course again following the same path with only using the information from the file. This would earn you 10% EC.

### **Blinker Lights**

Use interrupts to blink LED lights when the vehicle is turning. If the vehicle angle is more than |10 degrees|, you should then blink an LED at a rate of 2 Hz until the angle is back below |10 degrees|. You must use timer interrupts to blink the LED at this rate. Use different LED colors on the HiFive board to show whether the car is turning left or right. This would earn you 5% EC.

## Combination of Multiple Projects

Combining multiple of the above sample projects can help you earn extra EC points. Below are a couple examples on how you could combine them and how they would give more EC.

* Course Mapping and Reverse: Implementing both can allow you to have the car run the course in reverse. Using the file obtained from mapping the course while driving forwards, use the information to make the car drive the course in reverse. EC points: 10 + 2 + 3 extra = 15% EC.
* Anti-Collision, Reverse, and Bluetooth Remote Control: Implementing all three should give the car more flexibility on how it runs. You should be able to toggle the Anti-Collision feature using the controller and make the car automatically switch to manual mode if it detects collision. One in manual mode you should be able to use the controller to move the car with reverse available to a safer location. Self-driving can then be resumed with the controller. EC points: 10 + 2 + 5 + 3 extra = 20% EC.
* Anti-Collision, Reverse, and Blinker Lights: Implementing all three allows for an automatic way to attempt not colliding. When the car is stopped due to a detection of possible collision, the car is set to reverse and both the right and left turn blinker lights will blink while the car is in reverse. After reversing, the car functionality will attempt again to self-drive. If detection of possible collision, it will again attempt to reverse. This will keep occurring until the car manages to properly avoid collision. Extra Points: 10 + 2 + 5 + 3 extra = 20% EC.
* Reverse and Blinker Lights: After implementing both, also implement a white LED to blink at 2 Hz when the car is set to reverse. EC Points: 2 + 5 + 2 extra = 9% EC.

## Appendix for Milestone 4

Helful Links:

HiFive Documentation (Getting Started, Manual, Datasheet): <https://www.sifive.com/boards/hifive1-rev-b>

PI UART Configuration: <https://www.raspberrypi.org/documentation/configuration/uart.md>

TFmini Lidar: <https://cdn.sparkfun.com/assets/5/e/4/7/b/benewake-tfmini-datasheet.pdf>

Pigame Joystick: <https://www.pygame.org/docs/ref/joystick.html>