EECS388 Final Project – Fall 2020

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 (PCA9695) for driving servo and DC motors. PCA9695 uses I2C to receive commands from HiFive board to generate PWM signals for the servo and DC motors.

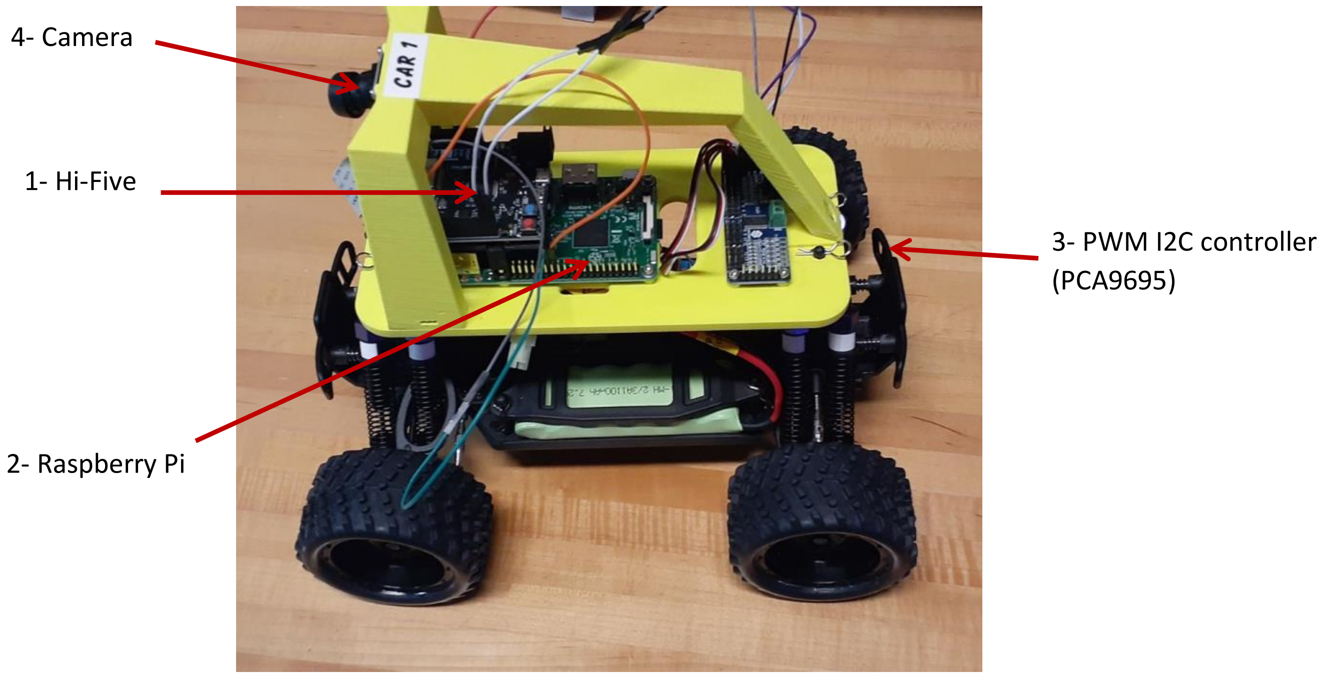


Figure 1: Self-driving car prototype

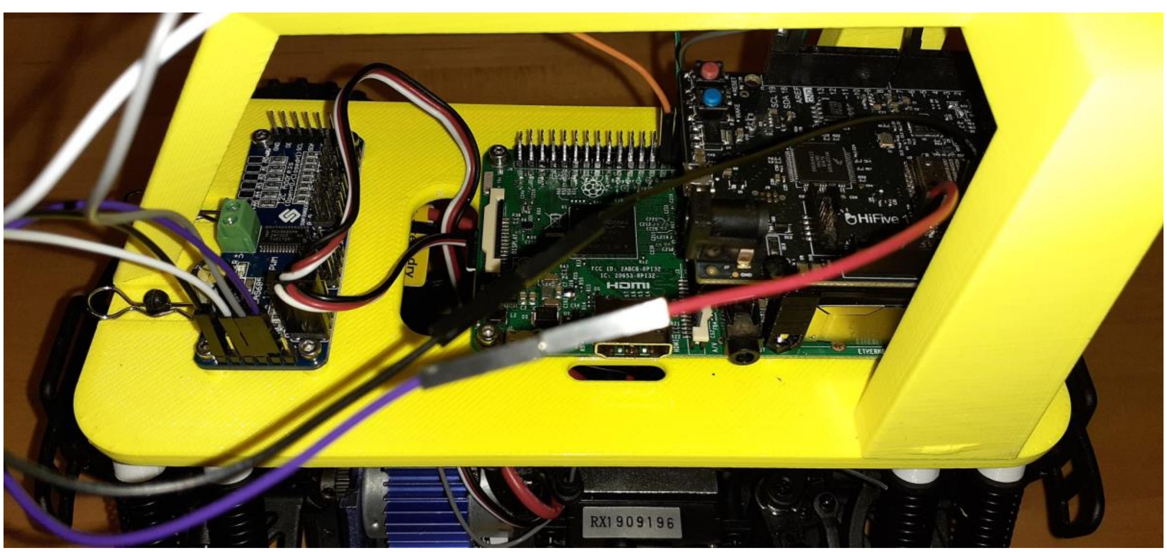


Figure 2: HiFive, Pi, and PWM I2C controller boards

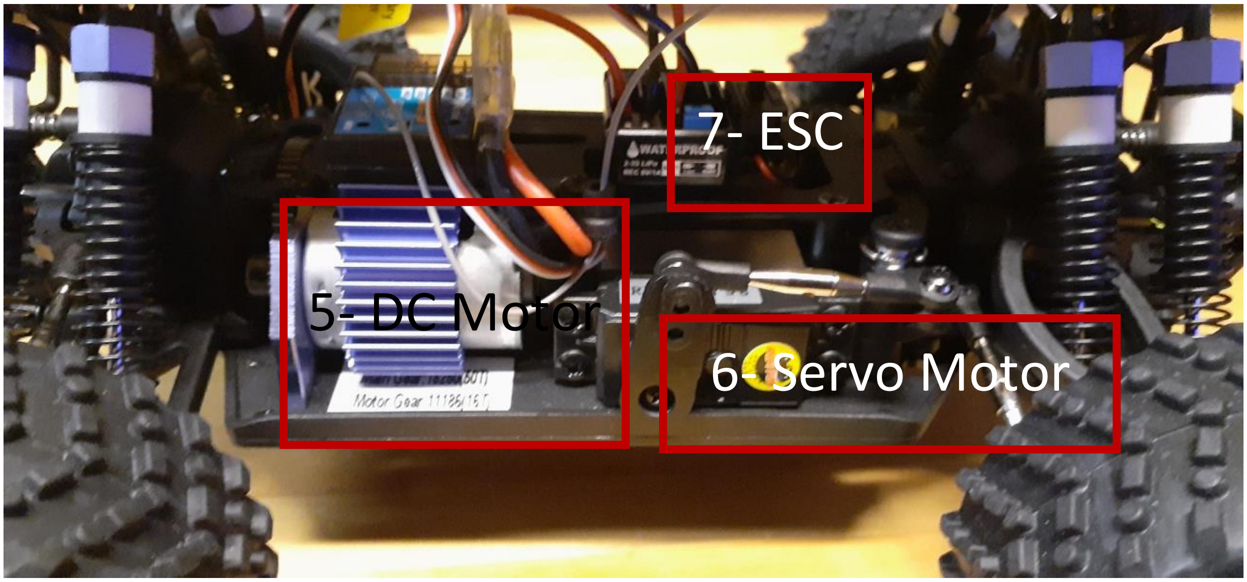


Figure 3: DC motor, Servo motor, and ESC on the car

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

1- HiFive

2- Raspberry Pi 4

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 HiFive board to send I2C commands to PCA9695 to drive the servo motor (for steering) and DC motors (for moving forward) (**Milestone 1**). Then connect the Pi to the HiFive board using UART. This is to set up a connection between the two boards for sending steering commands from the Pi to the HiFive 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 HiFive 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 when the **Milestone 4** section is released. 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 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	Apr 23rd
2	Connect HiFive and Pi	Apr 30th
3	Self-driving car	May 7th
4	Extra credit (open ended)	May 7th

Table 4: Deadline for each milestone

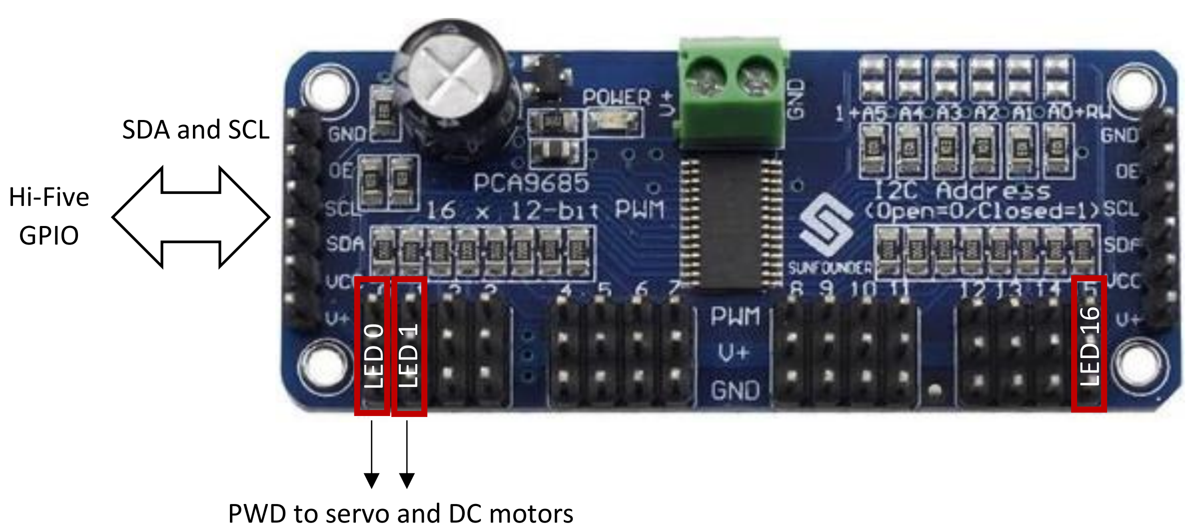
Milestone 1

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

You need to complete 3 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. 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.



On the HiFive board, the I2C core is from a 3rd party provider called OpenCores. If you look in the documentation for the HiFive 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 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 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.

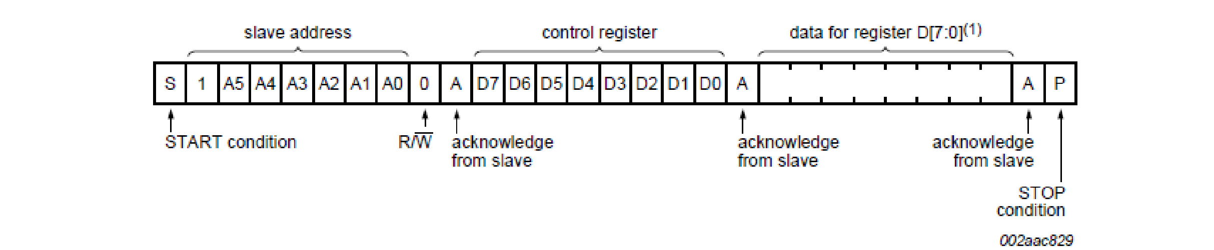
Part 2: Configuring the PCA9685

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):



(1) See Table 4 for register definition.

Fig 20. Write to a specific register

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 will 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_DISAB
LE);//resets the register
```

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

Table 5. MODE1 - Mode register 1 (address 00h) bit description

*Legend: * default value.*

Bit	Symbol	Access	Value	Description
7	RESTART	R		Shows state of RESTART logic. See Section 7.3.1.1 for detail.
		W		User writes logic 1 to this bit to clear it to logic 0. A user write of logic 0 will have no effect. See Section 7.3.1.1 for detail.
			0*	Restart disabled.
			1	Restart enabled.
6	EXTCLK	R/W		To use the EXTCLK pin, this bit must be set by the following sequence: 1. Set the SLEEP bit in MODE1. This turns off the internal oscillator. 2. Write logic 1s to both the SLEEP and EXTCLK bits in MODE1. The switch is now made. The external clock can be active during the switch because the SLEEP bit is set. This bit is a 'sticky bit', that is, it cannot be cleared by writing a logic 0 to it. The EXTCLK bit can **only** be cleared by a power cycle or software reset. EXTCLK range is DC to 50 MHz. $refresh\_rate = \frac{EXTCLK}{4096 \times (prescale + 1)}$
			0*	Use internal clock.
			1	Use EXTCLK pin clock.
5	AI	R/W	0*	Register Auto-Increment disabled[1].
			1	Register Auto-Increment enabled.
4	SLEEP	R/W	0	Normal mode[2].
			1*	Low power mode. Oscillator off[3][4].
3	SUB1	R/W	0*	PCA9685 does not respond to I²C-bus subaddress 1.
			1	PCA9685 responds to I²C-bus subaddress 1.
2	SUB2	R/W	0*	PCA9685 does not respond to I²C-bus subaddress 2.
			1	PCA9685 responds to I²C-bus subaddress 2.
1	SUB3	R/W	0*	PCA9685 does not respond to I²C-bus subaddress 3.
			1	PCA9685 responds to I²C-bus subaddress 3.
0	ALLCALL	R/W	0	PCA9685 does not respond to LED All Call I²C-bus address.
			1*	PCA9685 responds to LED All Call I²C-bus address.

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 pr
escale
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:

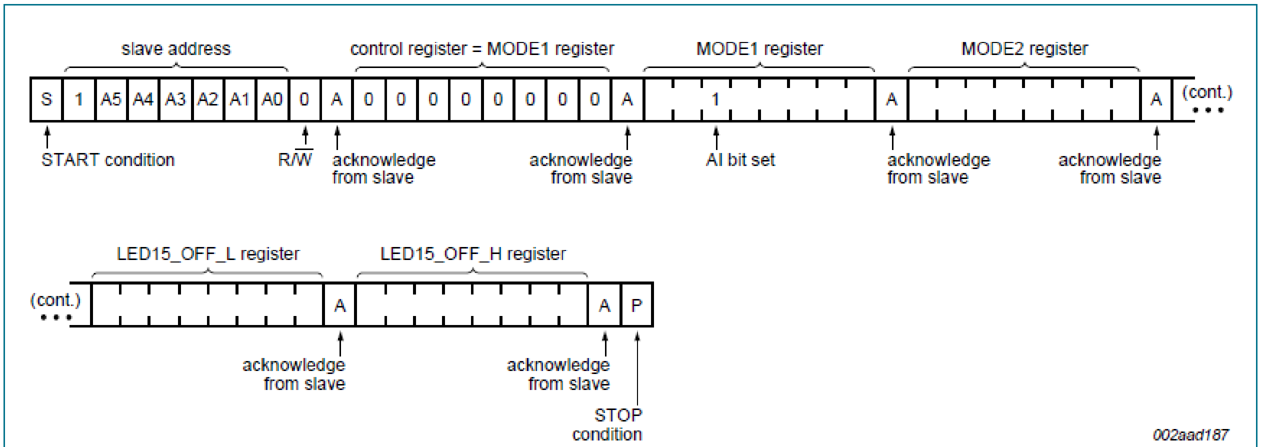


Fig 21. Write to all registers using the Auto-Increment feature; AI initially clear

Part 3: Using 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:

Table 4. Register summary

Register# (decimal)	Register# (hex)	D7	D6	D5	D4	D3	D2	D1	D0	Name	Type	Function
0	00	0	0	0	0	0	0	0	0	MODE1	read/write	Mode register 1
1	01	0	0	0	0	0	0	0	1	MODE2	read/write	Mode register 2
2	02	0	0	0	0	0	0	1	0	SUBADR1	read/write	I^2C-bus subaddress 1
3	03	0	0	0	0	0	0	1	1	SUBADR2	read/write	I^2C-bus subaddress 2
4	04	0	0	0	0	0	1	0	0	SUBADR3	read/write	I^2C-bus subaddress 3
5	05	0	0	0	0	0	1	0	1	ALLCALLADR	read/write	LED All Call I^2C-bus address
6	06	0	0	0	0	0	1	1	0	LED0_ON_L	read/write	LED0 output and brightness control byte 0
7	07	0	0	0	0	0	1	1	1	LED0_ON_H	read/write	LED0 output and brightness control byte 1
8	08	0	0	0	0	1	0	0	0	LED0_OFF_L	read/write	LED0 output and brightness control byte 2
9	09	0	0	0	0	1	0	0	1	LED0_OFF_H	read/write	LED0 output and brightness control byte 3
10	0A	0	0	0	0	1	0	1	0	LED1_ON_L	read/write	LED1 output and brightness control byte 0
11	0B	0	0	0	0	1	0	1	1	LED1_ON_H	read/write	LED1 output and brightness control byte 1
12	0C	0	0	0	0	1	1	0	0	LED1_OFF_L	read/write	LED1 output and brightness control byte 2
13	0D	0	0	0	0	1	1	0	1	LED1_OFF_H	read/write	LED1 output and brightness control byte 3

Task 1: The breakup function

When looking at the register summary, note that each LED 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
```

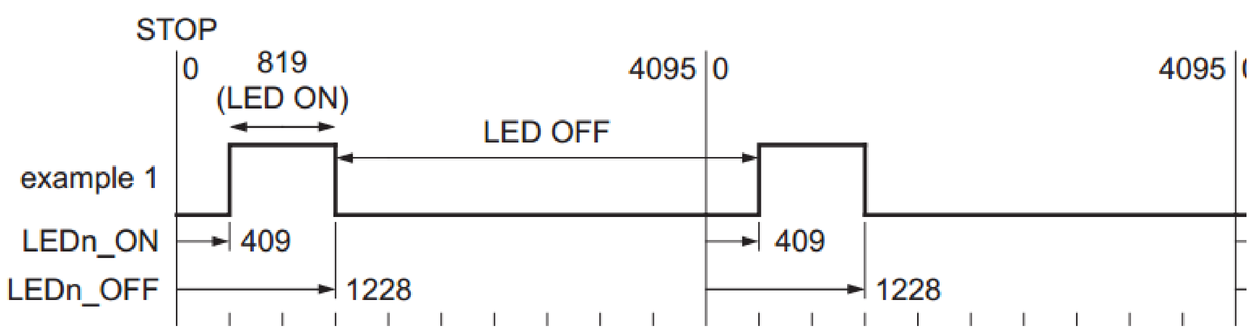
Controlling the servo and motor Electronic Speed Controller (ESC) with PWM is very similar to how we did so in the servo/PWM Lab. However, in this case, we will be converting values ranging from 0 to 20 ms to 0 to 4095 cycles.

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 LED[n]_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
```

Task 3: The write function

You will implement the following function to actually write values with i2c for steering the vehicle. The steering function from above should call this function, and in turn this function should use the metal_i2c_transfer function discussed above.

```
void write(){
      /*
             Write Task 3 code here
      */
}
// writing angle to steering servo
Example Use: write(PCA9685_LED0_ON_L, &variable1, &variable2);
```