Line Follower Robot – Report

Team Samus

Lucas Troy, John Klamut, Daniel Stoller

Team Roles

John Klamut – Motor Controls, Lucas Troy – Collision Detection, Daniel Stoller - FSM

Motor Drivers

High-level Module Overview with Code Samples

In the main.c, we use the motor functions defined in the motor handler to control the robot's movement. The functions control the motors using PWM. The PWM takes advantage of the TimerA1.c file to control the PWM functions. First, we initialize the PWM using the PWM_Init34() function. The function defined below first checks for bad input, if input is valid, we make P2.6, 2.7 outputs, and initialize them with the with the Timer. P2.6 and 2.7 are connected to interrupts CCR3 and CCR4. After this we initialize the CCR 0, 3, and 4 each with respected duty cycle duty3 or duty4 or the period. Additionally, we configure the timer to use SMCLK. The expansion factor is (EX0) is set to divide by 0x0000.

```
void PWM_Init34(uint16_t period, uint16_t duty3, uint16_t duty4){

  if(duty3 >= period) return; // bad input
    if(duty4 >= period) return; // bad input
    P2->DIR |= 0xC0;          // P2.6, P2.7 output
    P2->SEL0 |= 0xC0;         // P2.6, P2.7 Timer0A functions
    P2->SEL1 &= ~0xC0;        // P2.6, P2.7 Timer0A functions
    TIMER_A0->CCTL[0] = 0x0080;      // CCI0 toggle
    TIMER_A0->CCR[0] = period;       // Period is 2*period*8*83.33ns is 1.333*perioc
    TIMER_A0->EX0 = 0x0000;        //    divide by 1
    TIMER_A0->CCTL[3] = 0x0040;      // CCR3 toggle/reset
    TIMER_A0->CCR[3] = duty3;        // CCR3 duty cycle is duty3/period
    TIMER_A0->CCTL[4] = 0x0040;      // CCR4 toggle/reset
    TIMER_A0->CCR[4] = duty4;        // CCR4 duty cycle is duty4/period
    TIMER_A0->CTL = 0x02F0;        // SMCLK=12MHz, divide by 8, up-down mode
}
```

The above implementation can be summarized as the following: a counter count to TA0CCR0 and back down for the period defined as a parameter. Next, the CCR 3 and 4 interrupts count and depending on whether they are counting down or up determines if the P2.6 or P2.7 is a one or zero. For example, when counting down, P2.6 is equal to 1 and P2.6 is equal to 0 on the way down. We set the interrupt time by taking the given duty cycle and dividing it by the period.

Once the main initialization function was implemented, the ability to change the duty cycle for ccr3 and 4 would be crucial in changing the motor functions on the go. To do this both the functions PWM_Duty3 and PWM_Duty4 are implemented. The functions check for a valid duty cycle to make sure the value is less than the period specified in initialization. After validation, the function changes the duty cycle.

```
void PWM_Duty3(uint16_t duty3){

   if(duty3 >= TIMER_A0->CCR[0]) return; // bad input
    TIMER_A0->CCR[3] = duty3;        // CCR3 duty cycle is duty3/period

}

//***************************PWM_Duty4*******************************
// change duty cycle of PWM output on P2.7
// Inputs:  duty4
// Outputs: none// period of P2.7 is 2*period*666.7ns, duty cycle is duty2/period
void PWM_Duty4(uint16_t duty4){

   if(duty4 >= TIMER_A0->CCR[0]) return; // bad input
    TIMER_A0->CCR[4] = duty4;        // CCR4 duty cycle is duty4/period

}
```

Once all the PWM functions are initialized, we can create specific functions to control the motors themselves using the PWM functions. In the Motor initialization, the left motor is controlled by P3.7, 2.7, and 5.4 and the right motor is controlled by 3.6, 2.6, and 5.5. The initialization function sets the pins to the motor as outputs and keeps them off by disabling them through the 3.6 and 3.7 pins. Additionally, we call our PWM initialization function and set the period to 15000 while making the duty to each PWM 0.

```
void Motor_Init(void){
    P3 -> SEL0 &= ~0xC0;
    P3 -> SEL1 &= ~0xC0;
    P3 -> DIR |= 0xC0;

    P5 -> SEL0 &= ~0x30;
    P5 -> SEL1 &= ~0x30;
    P5 -> DIR |= 0x30;

    PWM_Init34(15000, 0, 0);
    P3 -> OUT &= ~0xC0;

    return;
}
```

Once we defined our initialization function, we defined different functions to control each motor. Motor_Foward turns on both motors to be in the forward direction. Motor_Left turns make the left motor go forward and the right motor go backwards. Motor_Right is the opposite of Motor_Left, and Motor_Backwards makes both motors go backwards. These directions are defined by a 1 or a 0(0 backwards and 1 forwards) through pins 5.4 and 5.5. These functions take in as parameters the duty cycles of the left and right motor so we can dynamically change the speed of the motors when using the FSM.

```c
void Motor_Forward(uint16_t leftDuty, uint16_t rightDuty){
    P3 -> OUT |= 0xC0;  // nSleep = 1
    P5 -> OUT &= ~0x30; // PH = 0
    PWM_Duty3(rightDuty);
    PWM_Duty4(leftDuty);
    return;
}

// ------------Motor_Right------------
// Turn the robot to the right by running the
// left wheel forward and the right wheel
// backward with the given duty cycles.
// Input: leftDuty  duty cycle of left wheel (0 to 14,998)
//        rightDuty duty cycle of right wheel (0 to 14,998)
// Output: none
// Assumes: Motor_Init() has been called
void Motor_Right(uint16_t leftDuty, uint16_t rightDuty){
    P3 -> OUT |= 0xC0;  // nSleep = 1
    P5 -> OUT &= ~0x10; // P5.4 PH = 0
    P5 -> OUT |= 0x20; // P5.5 PH = 1
    PWM_Duty4(leftDuty);
    PWM_Duty3(rightDuty);
}

// ------------Motor_Left------------
// Turn the robot to the left by running the
// left wheel backward and the right wheel
// forward with the given duty cycles.
// Input: leftDuty  duty cycle of left wheel (0 to 14,998)
//        rightDuty duty cycle of right wheel (0 to 14,998)
// Output: none
// Assumes: Motor_Init() has been called
void Motor_Left(uint16_t leftDuty, uint16_t rightDuty){
  // write this as part of Lab 13
    P3 -> OUT |= 0xC0;  // nSleep = 1
    P5 -> OUT |= 0x10; // P5.4 PH = 1
    P5 -> OUT &= ~0x20; // P5.5 PH = 0
    PWM_Duty4(leftDuty);
    PWM_Duty3(rightDuty);
}

// ------------Motor_Backward------------
// Drive the robot backward by running left and
// right wheels backward with the given duty
// cycles.
// Input: leftDuty  duty cycle of left wheel (0 to 14,998)
//        rightDuty duty cycle of right wheel (0 to 14,998)
// Output: none
// Assumes: Motor_Init() has been called
void Motor_Backward(uint16_t leftDuty, uint16_t rightDuty){
    P3 -> OUT |= 0xC0;  // nSleep = 1
    P5 -> OUT |= 0x30; // PH = 1
    PWM_Duty3(rightDuty);
    PWM_Duty4(leftDuty);
    return;
}
```

These are the functions that ultimately get used in the FSM in the Motor Handler. As said before, the ability for the motors to take in the left and right duty is key when it comes to the changing the degree of turn.

Testing, Validation and Debugging

When testing and debugging to make sure the motor functions work, I suspended the robot in the air to make sure the wheels would turn as except. I would then either flash or debug the robot using a simple main.c function as seen below.

```c
const int speed;
void main(void){

    Clock_Init48MHz();
    LaunchPad_Init();
    Motor_Init();



    Motor_Forward(speed, speed);
    Clock_Delay1ms(1000);
    Motor_Right(speed, speed);
    Clock_Delay1ms(1000);
    Motor_Left(speed, speed);
    Clock_Delay1ms(1000);
    Motor_Backward(speed,speed);
    Clock_Delay1ms(1000);
```

```
    Motor_Stop();
}
```

Using this simple test function, I would tweak the motor values for the finite state machine to figure out which degree of turn works best or find corresponding bugs in each function described in the above sections. Some bugs I ran into included configuring the right pins for PWM and accidentally switching the left and right PWM duty cycles.

Integration

When it came to integration, we originally created other functions to show case a hard right or a soft left as seen in our commented code or in the code provided below. However, we decided it would just be easier to use the motor functions above as we have more direct control for fine tuning the speed and delay by directly changing the PWM. The functions below were originally used in our old FSM. However, our new FSM uses the motor functions that control the PWM directly in the Motor Handler function.

```
void goStraight(void){
    Motor_Forward(speed,speed);
//    Clock_Delay1ms(time3);
//    Motor_Stop();
}

void goLeft(void){ //very very slightly turns left
    //Motor_Left(speed,speed);
    //Motor_Foward(speed, )
    //Clock_Delay1ms(time3);
//    Motor_Stop();
}

void goRight(void){ //very very slightly turns right
    //Motor_Right(speed,speed);
//    Clock_Delay1ms(time3);
//    Motor_Stop();
}

void goSlightLeft(void){ // turns robot slightly left
    Motor_Left(backup_speed,backup_speed);
//    Clock_Delay1ms(time2);
    Motor_Stop();
}

void goSlightRight(void){ // turns robot slightly right
    Motor_Right(backup_speed,backup_speed);
//    Clock_Delay1ms(time2);
    Motor_Stop();
}

void goHardLeft(void){ // 90 degree turn
    Motor_Left(backup_speed,backup_speed);
    //Clock_Delay1ms(time4);
    Motor_Stop();
}

void goHardRight(void){ // 90 degree turn
    Motor_Right(backup_speed,backup_speed);
    //Clock_Delay1ms(time4);
    Motor_Stop();
}

void goBackwards(void){
    Motor_Backward(backup_speed,backup_speed);
    //Clock_Delay1ms(time_backup);
//    Motor_Stop();
}
```

Collision Detection

High-level Module Overview with Code Samples

In the main.c, main loop, we initialize TimerA1 to be used as a trigger for the bump_interrupt interrupt service routine (ISR) function periodically. The general idea is explained at a high level in the code below.

```
void main(void){
    // Initialization
    // ...

    TimerA1_Init(&bump_interrupt, 48000);  // Initialize TimerA1 with
                                           // bump_interrupt as the user task

    // ...

    while(1){
        WaitForInterrupt();

        // Perform State Actions
          // ...

        // Read Data
          // ...
    }
}
```

When we initialize TimerA1, we assign the timer with a specific task, or a pointer to the function we want to use. The function below sets up TimerA1 in Up mode, configures it to use SMCLK (Sub-main Clock), and enables Timer_A1CCR0 interrupt. The expansion factor (EX0) is set to divide SMCLK by 6. The function also configures the NVIC (Nested Vector Interrupt Controller) to set the interrupt priority and enable TimerA1 interrupts.

```
void TimerA1_Init(void(*task)(void), uint16_t period){
    TimerA1Task = task;                   // Set the user task function
    TIMER_A1->CTL &= ~0x0030;             // Clear existing control settings
    TIMER_A1->CTL = 0x0280;               // Set control: SMCLK, Up mode, Clear Timer
    TIMER_A1->CCTL[0] = 0x0010;           // Enable Timer_A1CCR0 interrupt
    TIMER_A1->CCR[0] = (period - 1);      // Set the period for Timer_A1CCR0
    TIMER_A1->EX0 = 0x0005;               // Set the expansion factor to divide SMCLK by 6

    // Configure interrupt priority and enable TimerA1 interrupt in NVIC
    NVIC->IP[2] = (NVIC->IP[2]&0xFF00FFFF) | 0x00400000;
    NVIC->ISER[0] = 0x00000400;

    TIMER_A1->CTL |= 0x0014;              // Enable TimerA1 and enable interrupt
}
```

We also need a handler for the timer.

```
void TA1_0_IRQHandler(void){
    TIMER_A1->CCTL[0] &= ~0x001;           // Clear Timer_A1CCR0 interrupt flag
    (*TimerA1Task)();                      // Execute the user task
}
```

This function is the interrupt service routine (ISR) for TimerA1. It clears the Timer_A1CCR0 interrupt flag and then calls the user-defined task (TimerA1Task), our bump_interrupt function in the main our code. This ISR is executed each time TimerA1 reaches the specified period, triggering the bump interrupt ISR.

```
void bump_interrupt(void) {
    uint8_t bumpResult = Bump_Read();
    if (bumpResult != 0x3F) {
```

```
        int backup_speed = 2500, backup_time = 300;
        Motor_Backward(backup_speed,backup_speed);
        Clock_Delay1ms(backup_time);
        Motor_Stop();

        int turn_speed = 3000, turn_time = 900;
        Motor_Backward(turn_speed,turn_speed);
        Clock_Delay1ms(turn_time);
        Motor_Stop();

        Motor_Forward(4000,4000);
        Clock_Delay1ms(500);
        Motor_Stop();

    }
    return;
}
```

This function is the function that gets called periodically by TimerA1. Since the bump sensors are pull-down by default, whenever they are not all high, we tell the robot to back up at an angle and then move forward in the new direction.

Testing, Validation and Debugging

To test triggering the ISR, I placed the robot on a flat surface and flashed it with just this code.

```
void main(void){
    // Initialization
    TimerA1_Init(&bump_interrupt, 48000);  // Initialize TimerA1 with
                                            // bump_interrupt as the user task
    while(1){
        WaitForInterrupt();
    }
}
```

When the program runs, it will initialize the timer ISR, and then wait for an interrupt to occur. Whenever one of the bump sensors becomes depressed, the robot backs up at an angle and moves forward. It does this each time the sensors are pressed, and it can only be retriggered once the robot stops moving again. This seemed to work for our purposes, so we moved onto integration.

Integration

For integration we added in the state actions and reflectance reads in the while loop with the `WaitForInterrupt()`. The interrupt didn't seem to work with the first implementation of our finite state machine (FSM) as that caused it to get stuck in a context inaccessible by an interrupt. With our newer finite state machine, the interrupts were able to be called as the context constantly polled the while loop in the main. We can't explain exactly what fixed this issue, but redesigning the FSM surely did the trick.

Finite State Machine

High-level Module Overview with Code Samples

The finite state machine is the main decision maker of the line following robot. The goal of the finite state machine is to keep the robot on the line and when it loses the line, it can make the correct decisions to get back on it. Our team wanted to create an implementation that emphasized:

1. Speed over precision: The goal is to have the robot anywhere on the line, not to have the robot centered on the line all the time. Our group wanted to create an implementation that could go forward at full speed for a longer period and could go around turns faster. This was done by ensuring that there was a line beneath the robot, but that line didn't necessarily need to be directly in the center.
2. Exception handling: Our group wanted to make an FSM that could handle any sort of challenge the maze would throw at us. Whether the width of the line changes, sharp 90 degree turns, consecutive turns, or gaps in the line, we wanted the robot to be equipped with the correct sequences of actions to detect and overcome any such obstacles.
3. Simplified input: We are given an 8-bit sensor input, and we are using a finite state machine that requires a specified next state for every combination of input. To reduce the number of input bits so we can have a more simplified FSM, we wanted to reduce the number of bits to at most 3. This way, each state would only need to have 8 next states.

FSM V1: Difference Bit

Bit 0: Far Left Sensor

Bit 1: Far Right Sensor

Bit 2: Transition in and out of only 0's / 2+ bit difference

This implementation would detect when there are big changes in the line, whether it changes by 2+ total 1's, the line disappears, or the line reappears. It would make certain decisions based on those events. This implementation would adjust based mostly on the outside sensors and accounted for a dozen worst-case scenarios.

Output	**State**	**000**	**010**	**100**	**110**	**001 (Lost or widen)**	**011**	**101**	**111 (All on)**
Forward	**Center**	Center	Line on Right	Line on Left	Center *(Unlikely)* ********	Gap Jump *(line widens by a lot = won't work)*	Sharp Right	Sharp Left	Sharp Right *****
Slight Left	**Line on Left**	Center	Line on Right	Line on Left	Sharp Right	Sharp Left (lost line)	Line on Right ******	Sharp Left	Sharp Right
Slight Right	**Line on Right**	Center	Line on Right	Line on Left	Sharp Left	Sharp Right (lost line)	Sharp Right	Line on Left ******	Sharp Left
Hard Left	**Sharp Left Incoming**	Center	Line on Right	Gap Jump *****	Line on Right (left sensor is old line)	Center	Line on Right Center	Line on Left Center	Line on Right

Hard Right	Sharp Right Incoming	Center	Gap Jump *****	Line on Left	Line on Left (right sensor is old line)	Center	Line on Right Center	Line on Left Center	Line on Left
Forward	**Gap Jump *(Make gap jump L and R to account as "Lost")***	Gap Jump (infinite loop) ****** Circle Around	Line on Right	Line on Left	Sharp Right (not possible) ********	Center	Line on Right	Line on Left	Sharp Right *(lost and confused)* **********

Testing, Validation and Debugging

The first step was turning the 8-bit input into our desired input. Our implementation needed to know the number of 1's from the previous state, so we held it in a global variable called *prev_count*. The function below would check all 8 bits, adding 1 to *curr_count* for every sensor that was underneath a line. If the end bits were 1's, then it would reflect that within our 3-bit data. We would then check to see if we are going in or out of having zero 1's (using an XOR logic) and check to see if the difference between the total number of 1's in the previous and current state is 2 or greater. We use the results from this logic to decide the value of the 3rd bit.

```c
uint8_t Reflectance_FSM(uint8_t data){
    uint8_t curr_count = 0;
    uint8_t i, diff;
    uint8_t pc = 0, cc = 0;
    uint8_t new_data = 0x00;

    for (i=0; i<8; i++){
        if(((data >> i) & 0x01) == 0x01){
            if(i == 0) {    //0 = Rightmost Sensor
                new_data |= 0x02;    //Right Sensor = On
            }
            if(i == 7){     // 7 = Leftmost Sensor
                new_data |= 0x04;    //Left Sensor = On
            }
            curr_count++;     //Any Sensor is On
        }
    }

    diff = curr_count - prev_count;
    if(prev_count){
        pc = 1;
    }
    if(curr_count){
        cc = 1;
    }
    //    0 ! <-> 1   AND    diff = -1       diff = 0        diff = 1
    if((pc ^ cc == 0) & (diff == 0xFF || diff == 0x00 || diff == 0x01)){
        new_data &= ~0x01;
    }
    else{
        new_data |= 0x01;
    }
    prev_count = curr_count;
    return new_data;

}
```

The next step was implementing the FSM and assigning each state with a specified output. This involved creating the 6 desired states and filling out their desired next states depending on the given 3-bit input. Finally, we customized the output to move the motors in our desired way depending on the output bits and show on the LED which state we are currently in for debugging purposes. The connection between output and motor movement is done through a switch case, where the output value corresponds to a specific motor movement.

```
// Linked data structure
struct State {
  uint32_t out;                 // 3-bit output (only 0-5 used)
  uint32_t delay;               // time to delay in 1ms
  const struct State *next[8]; // Next if 3-bit input is 0-7
};
typedef const struct State State_t;

#define Center         &fsm[0]
#define Left           &fsm[1]
#define Right          &fsm[2]
#define SharpLeft      &fsm[3]
#define SharpRight     &fsm[4]
#define GapJump        &fsm[5]
```

```
// Inputs:        000          001      010      011      100       101      110       111
State_t fsm[5]={
  {0x00, 50,   { Center,       Right,   Center,  Right,   Left,     Center,  Left,     Center}},      // Center, time3
  {0x01, 150,  { SharpLeft,    Right,   Center,  Right,   Left,     Center,  Left,     Center}},      // Left, time3
  {0x02, 150,  { SharpRight,   Right,   Center,  Right,   Left,     Center,  Left,     Center}},      // Right, time3
  {0x03, 450,  { Center,       Center,  Center,  Center,  Center,   Center,  Center,   Center}},      // Sharp Left, time4
  {0x04, 450,  { Center,       Center,  Center,  Center,  Center,   Center,  Center,   Center}},       // Sharp Right, time4
  {0x05, 300,  { Center,       Center,  Right,   Right,   Left,     Left,    SharpRight,  SharpRight}}  // Gap Jump, time_backup
```

(These values do not match the table above. This implementation has been deleted)

```
void Motor_Handler(uint8_t data){
    switch(data) {
        case 0x00:
            Motor_Forward(speed, speed);
            Clock_Delay1ms(Spt->delay);
            break;
        case 0x01:
            Motor_Forward(slow_speed, speed);
            Clock_Delay1ms(Spt->delay);
            break;
        case 0x02:
            Motor_Forward(speed, slow_speed);
            Clock_Delay1ms(Spt->delay);
//              Motor_Stop();
            break;
        case 0x03:
            Motor_Left(backup_speed, backup_speed);
            Clock_Delay1ms(Spt->delay);
//              Motor_Stop();
            break;
        case 0x04:
            Motor_Right(backup_speed, backup_speed);
            Clock_Delay1ms(Spt->delay);
            break;
        case 0x05:
            Motor_Backward(backup_speed, backup_speed);
            Clock_Delay1ms(Spt->delay);
            break;
        case 0x06:
            Motor_Stop();
            break;
        default:
            Motor_Stop();
            break;
    }
```

Integration

This implementation proved to be overcomplicated, hard to debug, and caused many infinite loops. Even though the implementation was meant to specialize in exception handling, it caused the FSM to not work in standard cases. There was basically no success in this FSM and required a tweak.

FSM V2: Center Sensors Bit

Most of the issues of the previous iteration were caused by the 3rd bit, as it represented both going onto a line and falling off the line. This caused the robot to be unsure when it was on the line and when it wasn't. Additionally, the bit difference functionality was unhelpful and only accounted for a very few cases. Combining the 2 functionalities of the bit caused the robot to move in unpredictable ways. To simplify this, keep the overall structure, and still hit our desired goals, we changed the 3rd bit to represent all the middle bits instead. This would allow the robot to know when it is on a line vs when it falls off and needs to find its way back. It is ok to have all the middle bits represented by 1 bit because our implementation doesn't care where the line is underneath the robot if it is between the 2 edge sensors. This would allow the robot to keep going fast and only adjust when deemed necessary.

When deciding the next states of the FSM, the goal was to keep it as simple as possible and avoid using sharp turns unless necessary (the robot loses the line while turning). Many issues with the previous iteration involved the FSM predicting specific scenarios when the spacing wouldn't be as predicted. This new model must have more center states, which allows the robot to keep moving forward until it enters a more understandable environment. Overall, this new FSM is much simpler, more predictable, and, most importantly, more flexible.

FSM V2 Table:

Bit 0: Far Left Sensor-

Bit 1: Sensors 1-6 (0 only if all sensors are 0)

Bit 2: Far Right Sensor

Output	**State**	**000**	**001**	**010 (Ideal State)**	**011**	**100**	**101**	**110**	**111 (All sensors on)**
Forward	**Center**	Center	Right	Center	Right	Left	Center ******	Left	Center *******
Slight Left	**Line on Left**	Sharp Left	Right	Center	Right	Left	Center	Left	Center
Slight Right	**Line on Right**	Sharp Right	Right	Center	Right	Left	Center	Left	Center
Hard Left	**Sharp Left Incoming**	Center	Center	Center	Center	Center	Center	Center	Center
Hard Right	**Sharp Right Incoming**	Center	Center	Center	Center	Center	Center	Center	Center

FSM V2 Results:

A quick overview of this model was to turn in the direction of whichever edge sensor was on. If we are turning and the next input is all zeroes (lost), then we take a sharp turn in the direction we were turning. After the sharp turn, we go forward until we find the line again. This model was simple yet effective. It made our speed efficiency lie in the motor tuning and delay implementation which is better because it made our design more easily tunable.

This FSM worked much better than the previous one. The previous FSM relied on predicting the possible edge-cases or obstacles while this one relied on simplicity and consistency. These qualities were proven to be more useful on the tracks because there were many track features we never accounted for and even if we did, it would likely interfere with a different edge case we did account for. The simple model helped us get through tricky obstacles and prioritize getting back on the track rather than predicting the track.

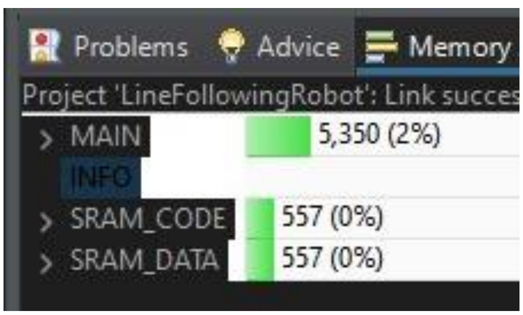
The only thing we might change about this FSM is to add functionality so if we do a sharp turn and don't see any line for a long time, then we do another sharp turn in the same direction. Additionally, we would have liked to have added a functionality that made turns progressively get sharper as the robot recalls its turn function consecutively. This would have helped with sharper turns because in some cases, instead of following the outside of the line the entire time, we can have the chance to get more in the center line and go straight, which is faster. Additionally, after watching the other cars race, it seemed that very fast micro adjustments in place were more effective than rolling turns where one motor is sped up higher than the other. I believe this movement could have been more effective

Overall, this FSM was simple and would be unphased by many of the obstacles in the courses. Additionally, it has enough functionality to find the track again in bad scenarios. Having a solid FSM that wasn't the source of tweaking for efficiency allowed for faster and more reliable testing with results that are easier to understand.

Speed Trial Results

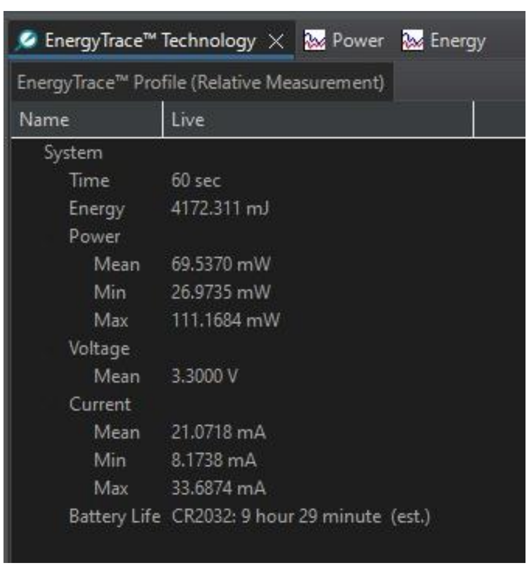
We obtained 7th place out of 15 robots. We completed the first-round track in 32.86 seconds.

Code Efficiency Results



Most of this code comes from variable declarations are for loops. Overall, our code is O(n).

Power Efficiency Results



The power is mostly for the motors.

Lessons Learned

The experience of developing the line-following robot taught our team valuable lessons. One key takeaway was the importance of early and frequent integration throughout the development process. Integrating different modules early on allowed us to identify and address potential issues promptly, leading to a more cohesive and functional final product. Additionally, the emphasis on creating a modular system proved beneficial, as it facilitated individual team members' focus on specific tasks and streamlined collaboration. Communication emerged as a critical aspect, underscoring the need for clear and effective team communication to ensure everyone was on the same page. Overall, the project reinforced the significance of integrating components early, maintaining modularity, and fostering open communication for successful collaborative projects.