IT집중교육1
프로젝트명: “Easy-Follow”
3조

Final Project Report

제목: Easy-Follow
담당 교수: 이정태 교수님
담당 조교: 박효승 조교
제출일: 2020년 10월 9일 (금요일)

3조
소프트웨어학과 남도현 201620921
소프트웨어학과 윤태섭 201620896
소프트웨어학과 최호영 201620929
소프트웨어학과 최형택 201620989

Contents

1. 요구 사항

1.1. 물리적 요구사항

시스템을 만들 때 현실에 어떻게 적용할 수 있을지를 고민하였다. 우리는 세 가지 항목을 만족해야 한다고 생각했다.

1.1.1. 정지

사고를 피하고자 차를 멈추기 위해서는 두 가지 요소가 고려되어야 하는데, Reaction distance(반응 거리)와 Braking distance(제동거리)가 그것들이다. 저 두 개를 합쳐서 stopping distance 라고 부를 수 있다.

반응 거리는 위험을 감지한 시점부터 운전자가 위험을 회피하기 위한 행동을 취하기까지 차가 움직인 거리이다. 제동거리는 운전자가 페달을 밟고 나서부터 차가 멈출 때까지의 거리이다.

반응 거리는 당연히 속도에 의해서 영향을 받는다. 거기에 더해 시스템이 브레이크를 밟는 반응 시간에도 영향을 받는다. 시스템의 안전성을 위해서는 반응 시간이 적어도 인간 반응시간만큼은 빨라야 한다.
개개인별로 차이가 있기 때문에 정확히 정할 수는 없지만, 인간 반응속도는 0.15초에서 3.5초 사이의 값을 가진다. 안전을 보장하기 위해서는 시스템이 적어도 인간 반응속도 중 제일 빠른 값을 가져야 한다.

1.1.2. 조향

시스템이 설치된 차량은 앞서가는 차를 따라갈 수 있어야 한다. 이것은 앞차가 좌, 우 회전을 할 때도 추적할 수 있어야 함을 의미한다. 그리고 이 프로젝트에서는 앞차 외의, 조향에 방해가 되는 다른 차량도 없고 교차로도 없고 급회전도 없다. 또한 조향장치의 사용이후 조향장치는 원래 위치로 돌아와야 하며 이 위치는 조향장치를 사용하기 이전의 위치와 완벽하게 동일해야 한다. 초기 위치가 조향장치 사용 이후의 위치와 달라지면 이는 주행에 있어서 조향장치를 계속 사용하는 결과로 이어지며 조향장치는 주행에서 불안정한 모습으로 이어진다.

1.1.3. 직선 주행

조향만큼 중요한 것이 직선 주행이다. 이것은 단지 핸들을 정위치에 놓고 가속 페달을 밟는 것만으로 해결되지 않는다. 시스템상에서는 두 가지 요소가 만족하여야 한다. 맨 처음 자동차의 시동을 켰을 때, 핸들이 정방향 & 정위치에 놓이는 것이 보장되어야 하고, 후륜 가속을 하는 양쪽 모터의 출력이 배터리양에 따라 다르기 때문에 같은 속도로 앞으로 가기 위해서는 속도를 양쪽에 다르게 해야 한다.
따라서 직선 주행을 위해 설계에서 확인해야 하는 부분은 조향장치가 확실한 정면을 향하고 흔들리지 않는것과 시스템의 무게를 고려한 좌우 바퀴의 회전 수를 같게 하는 것이다. 이 두가지를 우선적으로 해결해야 원활한 직선 주행이 가능하다고 판단하였다.

1.2. 구현 시스템 정의

이 시스템이 설치된 차(이하 follower)는 앞에 가는 차(이하 leader)를 따라갈 수 있다. 시스템이 실행되고 있을 때는 인간 운전자처럼 충돌을 피할 수 있다.

1.2.1. 전제 조건

Freeway, 즉 급속한 회전이나 교차로 등이 존재하지 않는 길에서 차량이 운행한다(편도 1차선 도로).

1차선 도로임을 가정하고, 마주 보고 오는 차나 추월하는 차가 없다(시스템이 설치된 차와 그 앞차 단 두 대만 존재한다)

시스템이 설치된 차 앞에 항상 따라갈 차가 존재한다.

nxtOSEK의 task들은 우선순위를 가지고 있다. 이 task들의 scheduler는 가장 높은 우선순위를 가진 task를 먼저 실행한다는 원칙을 가지고 있다. scheduler는 task에 공유할 수 있는 리소스를 정의할 수도 있다. 이 리소스들은 task들이 변수를 한 번에 한 개씩만 읽고 쓸 수 있게 할 때 사용할 수 있다.

리소스를 점유하고 있는 task는 리소스의 우선순위만큼 자신의 우선순위를 올려야 한다. 그리고 리소스의 우선순위는 가장 높은 우선순위를 가진 task의 것과 동일해야 한다. nxtOSEK은 PCP를 사용하고, 따라서 task의 우선순위는 리소스를 가질 때까지 계속 높아진다.

2. 분석

2.1. 장치 분석

2.1.1. 초음파 센서

초음파 센서는 앞에 있는 물체의 거리를 측정한다. 초음파를 방출해 전방의 물체에 부딪힌 후 돌아오는 시간을 측정해 거리를 계산한다. 최적의 초음파 센서를 고르기 위해서는 거리, 각도, 간섭 테스트를 설계하고 진행해야 한다. 또, 초음파 센서는 차에 여러 방법으로 고정이 가능하기 때문에 수직, 수평 방향으로 설치했을 때의 정확도도 실험해야 한다.

5개의 센서가 있었고, 거리 테스트를 진행해서 가장 정확도가 높은 2개를 골라 각도 테스트와 간섭 테스트를 진행했다.

2.1.1.1. 거리 테스트

종이 한 장을 센서 앞에 놓고 진행한다. 물체의 표면이 흡음재로 작용해서 큰 영향을 끼칠 수 있다는 점을 유념해야 한다. 이것은 센서의 결과값이 표면에 따라 달라질 수 있음을 의미한다. 그러므로 같은 표면에서 다른 센서들을 이용해 비교하는 것이 아주 효과적이다.

지표면과 센서를 수평 방향으로 놓고 수행한 테스트에서 센서에 따라 신뢰 범위가 다양하다. 어떤 센서들은 100cm까지 신뢰성 있는 결과가 나오고, 어떤 것들은 그 반도 안 되는 것도 있다.

수직 방향의 테스트에서도 결과가 비슷하다. 하지만 가장 정확한 센서도 75cm까지만 신뢰성 있는 결과가 나온다.

초음파 센서 거리 테스트 - 센서가 지면과 수직 방향
테스트 방식: 센서를 지표면과 수직으로 놓고, 아래 '눈'은 지표면에서 3.8cm 위에 배치하고 위 '눈' 은 7.3cm 위에 배치한다. 센서의 눈앞으로부터 직선거리를 측정한다. 10cm, 20cm, 40cm, 50cm, 75cm, 100cm 그리고 200cm의 거리마다 각각 종이를 앞에 놓고 10초간 대기한다.

5개의 센서에 각각 테스트를 진행했다. 여기서는 가장 정확하게 동작한 센서를 1번과 2번으로 설정하고, 그 센서들의 결과를 소개한다.

1번 센서는 40cm까지는 약 1~2cm의 오차를 고려하면 정확하게 측정한다. 실험의 설계가 정밀한 환경에서 이루어지지 않은 점을 고려할 때 매우 정확한 편이다. 50cm부터는 종이를 인식하지 못하기 시작하고, 100cm부터는 거의 인식하지 못한다.

2번 센서는 약 35cm까지 정확하게 측정하고 50cm와 75cm 사이 구간부터는 부정확하게 측정하거나 종이를 인식하지 못한다. 100cm 이후는 종이를 인식할 때도 있으나, 보통 매우 부정확하게 감지한다.

결과: 지면과 수직 방향으로 센서를 설치했을 때 가장 정확한 센서 2개는 각각 40cm와 35cm까지는 신뢰도를 보장할 수 있지만, 그 이후 범위는 매우 부정확하다.

초음파 센서 거리 테스트 - 센서가 지면과 수평 방향
테스트 방식: 센서를 지표면과 수평으로 놓고, 지표면에서 4.2cm 위에 배치한다. 나머지 실험 설계는 수직 방향 테스트와 동일하다.

역시 5개의 센서에 각각 테스트를 진행했고, 가장 정확한 센서 2개를 1, 2번 센서로 이름 붙여 이 센서들의 결과만 소개한다.

1번 센서는 50cm까지는 약 1~2cm의 오차를 고려하면 정상적으로 측정한다. 75cm 부터는 종이를 인식하지 못하기 시작하고, 100cm부터는 거의 인식하지 못한다.

2번 센서는 약 45cm까지 정상적으로 측정하고, 150cm부터는 정확한 측정이 불가능하다.

결과: 지면과 수평 방향으로 센서를 설치할 때 가장 정확한 센서 2개는 각각 50cm와 45cm까지는 신뢰도를 보장할 수 있지만, 그보다 먼 거리에서는 신뢰성을 보장할 수 없다.

2.1.1.2. 각도 테스트

초음파 센서는 바로 정면에 있는 물체뿐만 아니라 옆에 있는 물체도 측정이 가능하다. 그렇다면 대략 몇 도 옆에 있는 것까지 측정이 가능할지 테스트해 보았다. 이전 거리 테스트에서는 종이를 센서에서 가까이했다가 멀리하는 식으로 했지만 이번에는 거기에 더해 종이의 위치를 센서에서 왼쪽, 오른쪽을 반복하며 테스트 할 수 있다.

역시 각각의 센서에 따라 큰 편차를 보인다. 일부는 큰 편차를 보이고 일부는 거리가 달라져도 좀 더 일관성 있는 값을 유지한다. 또한 센서가 지표면과 수평 방향인지, 수직 방향인지에 따라서도 달라진다.
가장 좋은 센서는 정확한 값을 추출할 수 있는 최대 거리에 종이가 있을 때 좌우 7cm까지는 정면에 있을 때와 같은 값을 출력하는 것이다.

초음파 센서 각도 테스트 - 센서가 지표면과 수평 방향
테스트 방식: 거리 테스트와 동일하지만, 센서 양쪽 눈 사이 정중앙으로부터 1cm씩 종이를 이동해 가면서 테스트했다.

결과: 우리가 예상한 가설은 종이의 이동에 따라서 센서의 인식 범위가 원뿔 모양을 그린다는 것이었다. 그러나 실제 테스트 결과는 그와 달라서, 중앙선에서 일정 거리까지 멀어질 때는 정면 거리와 같은 값을 측정한다. 그리고 일정 거리 이후부터는 인식하지 못한다.

왼쪽으로 이동할 때와 오른쪽으로 이동할 때 인식할 수 있는 최대범위가 다르다. 따라서 두 경우의 평균을 내어서 값을 서로 비교했다.

또한 초음파 센서가 지표면에서 수직일 때와 수평일 때의 값이 다른 것도 확인했다. 수직 방향으로 테스트를 진행했을 때는 인식 가능한 최대거리에서 대체로 좌우 5에서 6cm까지 인식할 수 있고, 수평 방향일 때는 센서 간 편차가 크지만 3~8cm까지 인식할 수 있다.

2.1.1.3. 간섭 테스트

앞서 초음파 센서는 음파를 방출하고 그것이 다시 돌아올 때까지의 시간을 측정하는 방식으로 동작한다고 설명했다. 그렇기 때문에, 센서는 다른 센서에서 방출된 음파를 받아들일 수도 있다. 여러 개의 초음파 센서가 쓰인다면 보다 정확한 측정을 위해서 서로 어떻게 간섭이 일어나는가를 아는 것이 필수적이다.
한 센서를 벽 쪽으로 향하도록 고정해놓고 다른 센서를 위나 옆에 놓고 여러 방향으로 돌려가면서 각도를 조절한다. 정지된 센서의 값을 읽으면서 움직이는 센서가 변함에 따라 값에 어떤 변화가 있는지 파악한다.
각 센서 간 샘플링 시간이 50ms이면 회전하는 센서가 고정된 센서와 같은 방향을 볼 때 측정 거리가 반으로 감소한다.

만약 샘플링 시간이 40ms로 내려가면 고정된 센서가 움직이는 센서의 위치에 영향을 받지 않는다. 주기를 최소로 줄인다면 12ms까지 줄일 수 있고, 10ms 아래로 내려가면 음파를 보내고 반사되어 받는 시간보다 더 빠르게 샘플링하게 되어서 부정확한 값이 측정된다.

초음파 센서 간섭 테스트
테스트 방식: 초음파 센서 하나를 모터에 부착하고 다른 하나를 고정한다. 센서는 수평면 내에서 동일한 좌표가 되도록 부착해야 한다. 모터는 -90도부터 90도 사이를 원점 기준으로 회전하도록 프로그래밍한다. 고정된 센서를 기준으로 모터의 각도를 결정한다. 센서들은 벽면을 향하게 되어 있다. 센서의 샘플링 속도와 간섭을 일으키는 각도를 다양하게 해서 테스트를 진행한다. 이제 고정된 센서에서 간섭 정도가 다양하게 나타나는 것을 확인할 수 있다.

결과: 양 센서의 샘플링 대기 시간이 40ms일 때 중심부와 주변부에서 모두 간섭이 최소화된다. 테스트 결과 12ms가 정확한 값을 보장하는 가장 짧은 주기이다.

2.1.1.4. 실제 주행에 적용하기

앞서 수행한 세 종류의 테스트의 결과는 모두 차량에 수평 방향으로 센서를 부착하는 것이 수직의 경우보다 더 정확하다고 말하고 있다. 그러나 실제 시연 시에는 수직으로 센서를 부착하였다.

그 이유는 초음파 센서가 수직 방향으로 설치되었을 때 수평 방향으로 센서를 부착했을 때보다 더 정밀한 값을 받아 앞차를 더 잘 따라갔기 때문이다. 그렇다면 왜 실험 결과와 실제 주행 시가 차이가 날까?

우리는 실험 조건의 차이라고 생각한다. 앞서 수행한 테스트는 모두 센서가 정지해 있을 때 수행했다. 그런데 실제 주행 시에 센서의 공간에서의 위치는 계속 변하고, 주행 시 발생하는 진동이 정밀한 측정을 방해하며, 앞차와의 거리와 각도도 일정하지 않다. 실제 주행 시에는 이런 여러 가지 요인들이 존재한다.

이런 요인들로 인해 테스트 결과와 실제 주행의 센서 방향에 따른 신뢰도가 반대되었고, 실제 주행 시 더 안정적인 주행을 보장하는 수직 방향으로의 센서 배치를 선택했다.

2.1.2 레고 마인드스톰 바퀴

2.1.2.1. 타이어의 크기

주어진 타이어는 두 종류가 있었고, 크기와 타이어 무늬에 차이가 있었다. 크기가 크고, 험지 주행에 적합한 타이어 무늬를 가진(마찰력이 큰) 타이어를 A, 상대적으로 작고 고속 주행에 적합한 타이어 무늬를 가진(마찰력이 작은) 타이어를 B라고 부르자.

A를 선택했을 때의 장점은 B보다 지면과의 마찰력이 더 커서 급정거 시 정지거리를 줄일 수 있다. B의 장점으로는 A보다 고속&저속 모드의 차이를 확실히 보여줄 수 있고, 마찰이 적어서 좌우 방향 전환에 더 유리하다.

실제 주행 테스트 시 주행, 방향 전환, 정지 등의 항목을 모두 고려해서 뒤차에 A와 B 모두를 시험해 봤다. A의 경우 정지가 용이하고, 정지거리가 짧은 이점이 있었다. 하지만 LEGO Mindstorm 으로 제작한 차량은 차체 무게가 가벼워서 그 차이가 그리 크지 않았다. 반대로 조향과 속도 측면에서는 B가 A보다 같은 조향 모터 출력에도 더 빠르게 방향을 바꿀 수 있었고, 같은 후륜 모터 출력에도 더 빠르게 주행이 가능했다. 종합적으로 A가 B보다 실제 주행에 더 의미 있는 장점들이 있어서 우리는 타이어 B를 선택했다.

2.1.2.2 완충재의 유무

우리에게 주어진 4개 타이어 중 일부가 타이어 휠과 완전히 밀착되어 있지 않았다. 밀착되지 않은 타이어가 후륜 모터에 설치되면 모터로부터 제대로 힘을 받지 못해서 직선 주행 시에 그 타이어 방향으로 차가 밀리게 된다. 또한 조향 모터에 설치될 때는 우리가 설정한 회전값보다 더 곡선을 크게 그리고, 이로 인해 앞차를 놓치게 된다.

이 문제를 해결하기 위해서 우리는 휴지를 구해와서 휠과 타이어 사이의 빈 곳에 채워 넣어, 모터의 출력이 온전히 타이어에 전해지도록 했다. 완충재에 의해서 타이어와 휠 사이의 결합 문제를 해결하여, 안정적인 주행에 도움이 되었다.

3. 설계

3.1. Tasks

Task name	AutoStart	Priority	Period	Resource	Event
Initialize	TRUE	6	-	R1	-
SonarSensing	FALSE	5	12ms	-	-
EventDispatcher	FALSE	4	5ms	-	-
BrakeTask	TRUE	8	-	R1	event2
SpeedTask	TRUE	7	-	R1	event1

3.1.1. 코드 유사성

뒷차가 앞차를 따라갈 때 조향을 하고, 전진, 정지를 하는 부분이 2인 과제의 bt_receive_buf [32]를 통해 기능 구현이 이미 되어 있었기 때문에 코드를 재사용하여 구현하고자 했다. 이로 인해 앞차와 뒷차의 코드가 대부분 일치한다. 뒷차에 추가된 부분은 초음파 센서의 값을 이용해서 조향장치를 움직이는 부분과 거리에 따라서 속도를 조절하는 부분이다.

뒷차와 앞차의 코드가 유사하기 때문에 나눠 설명하지 않고 한 번에 설명한다.

3.1.2. Initialize

이 TASK의 역할은 시스템 실행 시 제일 먼저 수행되어 모터들의 출력을 바탕으로 조향과 동력을 제어해서 차량 주행에 신뢰성을 보장하는 것이다.

이 TASK의 목적은 두 가지이다. 첫 번째로 차량 주행 시 방향을 전환하지 않을 때는 스티어링 휠이 정위치에 오도록 하는 것이다. 두 번째로 모터의 출력에 따른 회전 각도의 차이를 고려해서 좌우 바퀴의 회전 속도를 같게 하는 것이다.

만약 조향모터와 동력 모터의 출력값이 주행 전에 계산되지 않으면 우리가 코드에서 모터에 부여한 속도와 실제로 모터가 돌아가는 속도가 일치하지 않는다. 원하는 대로 차량을 움직이기 위해서는 반드시 할당한 속도와 실제 속도가 일치해야 하기 때문에 높은 우선순위인 priority 6를 줬다.

Resource R1을 initializeTask에 할당하는 이유는, R1에 ceiling priority를 부여해서 이 task가 한 번 실행되고 나서 우선순위 역전이 일어나지 않고(preempt가 일어나지 않고) terminate 할 때 까지, 즉 자기 일을 모두 끝마칠 때 까지 실행되어야 하기 때문이다.

추가적으로 앞차의 경우 조향장치가 조금 흔들리는 모습이 보였고 조향장치를 중앙으로 이동후에도 흔들리는 것 때문에 주행에 영향을 줄 수 있다고 판단, 이를 initialize에서 보정해야 한다는 결론을 얻었다.

3.1.3. SonarSensing

이 TASK의 목적은 초음파 센서로부터 값을 받는 것이다. 짧은 주기로 지속적으로 값을 받아오면 좋지만 초음파 센서가 초음파 펄스를 송신하고 물체로부터 반사되어 수신되는 시간보다 task의 period가 짧으면 정확한 값을 받지 못한다. 따라서 정확한 값을 받으면서 최대한 짧은 주기로 판단되는 12ms를 period로 정하고, 센서가 측정하는 유효 거리를 50cm이하로 정한다. 12ms마다 SonarSensing Task를 Activate하는 cyclic_alarm2를 통해 호출된다.

초음파 센서의 period를 더 짧게 할 수 없기 때문에 최대한 빠르게 초음파 센서의 값을 받아오기 위해서 EventDispatcher보다 priority를 높게 주어 우선적으로 처리되도록 했다. 또한 50cm로 설정한 이유는 앞서 분석부분에서 설명했듯 신뢰가능한 센서 최대 측정 범위가 50cm 인 것과, 50cm보다 큰 값으로 했을 때 다른 물체로 부터 받는 간섭 때문이다.

두 번째 이유에 대해 부연설명을 하자면, 실내에서 주행을 생각하면 다른 물체가 전혀 없다는 가정을 할 수 없고, 벽이나 다른 물체에 가까이 가면 필연적으로 그 물체에 영향을 받는데 50cm보다 멀리하면 이 간섭을 받을 확률이 증가한다. 따라서 최대한 앞차의 판만 인식하는 동시에 다른 간섭을 최소화하는 거리로 50cm를 선정하게 되었다.

보다 정확한 값을 위해 센서로부터 받아들인 값이 2번 연속으로 같을 경우에만 사용하였다. 또한 초음파 센서의 값을 동시에 받아오지 않고 한 번에 하나씩 교대로 받아와 두 초음파 센서 사이의 간섭을 최소화했다.

3.1.4. EventDispatcher

이 TASK는 Easy-follow의 전반적인 구동 처리를 하는 것이 목적이다. 주된 처리를 하는 TASK이기 때문에 period를 5ms로 짧게 주어 자주 실행되게 하며 SpeedTask, BrakeTask가 실행될 상황에서 빠르게 처리되도록 한다. 측정되는 초음파 센서에 맞게 처리해야 하기 때문에 SonarSensing Task보다 priority를 낮게 주어 센서값을 받은 후에 실행되도록 한다.

조향장치의 경우 목표한 각도까지 조향장치를 이동하는 것과 목표 각도에서 중지, 이후에 원 위치로 돌아오는 기능을 모두 포함해야 한다. 시작전에 set_count에 0을 넣어서 현재의 위치를 기억해야 한다.

3.1.5. SpeedTask

event1을 발생시키는 이 TASK는 현재 속도가 저속이면 고속으로, 고속이면 저속으로 토글해주고 Motor_Run_Fun()을 호출하여 모터를 동작하게 한다. R1이라는 resource를 통해 실행중에 다른 TASK가 실행되지 않도록 한다.

EventDispatcher에서 속도를 바꿔야 할 상황이라 판단하면 SpeedTask의 event1을 발생시켜 속도를 바꾼다. 속도를 바꾸는 상황은 빠르게 처리하지 않으면 충돌 상황, 앞차를 잃어버리는 상황이 일어나기 때문에 Easy-Follow에서 가장 우선시 처리되어야 하는 것 중 하나이기 때문에 priority를 높은 7로 준다.

실행 주기를 따로 주면 속도를 바꾸어야 하는 상황에서도 주기에 맞춰 실행되기 때문에 오히려 더 늦게 발생할 수도 있고, 속도를 바꾸지 않아도 되는 상황에서는 의미없이 다른 Task들의 실행 순서를 바꿀 수 있기 때문에 event로 만들어 해당 상황에서 최대한 빠르게 처리하는 방식으로 한다.

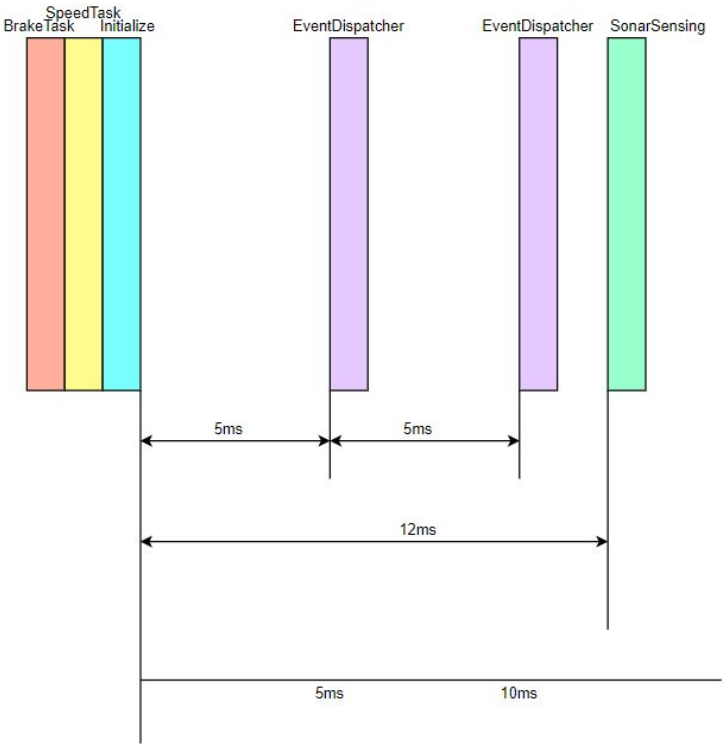
3.1.6. BrakeTask

event2를 발생시키는 이 TASK의 목적은 즉시 또는 천천히 정지하도록 하는 것이다. EventDispatcher에서 정지를 해야한다고 판단하면 BrakeTask의 event2를 발생시켜 정지한다. 정지하는 상황은 다른 상황보다 우선시 하지 않으면 충돌하는 사건이 발생하기 때문에 제일 먼저 처리되어야 하기 때문에 priority를 가장 높은 8로 주어 다른 task보다 먼저 실행되도록 해야 한다.

SpeedTask와 마찬가지로 실행 주기를 따로 주면 실행하지 않아도 될 때에 다른 TASK들의 실행을 방해할 수 있고, 이 점을 고려해 실행 주기를 길게 하면 반응이 느려지기 때문에 event2를 통해 실행해야할 상황을 EventDispatcher에서 판단하고 그에 따라 바로 실행되도록 한다.

정지해야 하는 순간에 다른 Task가 뺏어가면 안되기 때문에 R1이라는 resource를 통해 다른 Task의 간섭없이 실행되도록 한다.

3.1.7. Task Scheduling



autostart를 true로 해놓은 Initialize, BrakeTask, SpeedTask 3개의 task들은 시스템이 시작되자마자 Activate된다. priority가 가장 높은 BrakeTask가 먼저 Activate되고 priority 높은 순서인 SpeedTask, Initialize 가 차례로 실행된다. Initialize는 처음에 실행되었다가 그 뒤로는 실행되지 않고, BrakeTask와 SpeedTask는 EventDispatcher에서 해당 Task를 SetEvent할 때만 실행된다. SpeedTask와 BrakeTask는 Wait 상태로 기다리다가 EventDispatcher가 Task를 SetEvent하면 실행된다.

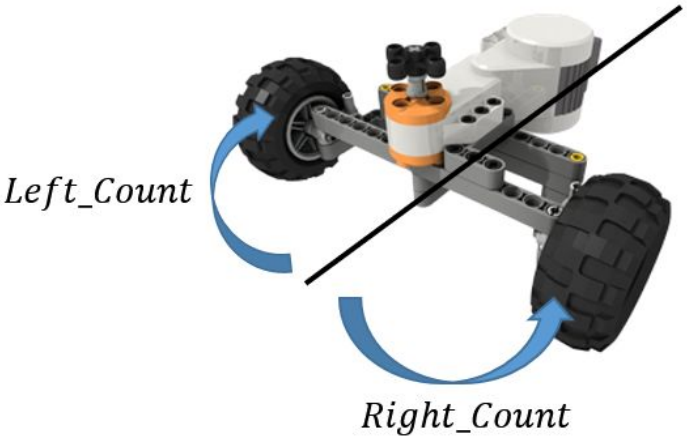
4. 구현

4.1. Tasks

4.1.1. Initialize

Initialize는 차량이 가동될 때 원활한 주행이 가능하도록 차량의 초기 상태를 설정하는 Task이며, 조향 장치 영점 조정과 배터리 잔량에 따른 양쪽 후륜 모터 속도 보정 두 가지 역할을 수행한다.

먼저, Initialize는 차량의 조향 장치가 직진 방향을 정확하게 바라보도록 정위치 시킨다. 차량의 조향 장치가 좌우 대칭이기 때문에, 모터를 양쪽 끝까지 한번씩 이동시킨 후 두 지점의 중앙을 구해 정확한 직진 방향을 찾을 수 있다.



$$(center) = (Left\_Count - Right\_Count) / 2 + Right\_Count$$

코드에서는 조향 모터의 nxt_motor_get_count() 리턴값이 변하지 않는 동안 while문을 실행한다. nxt_motor_get_count()의 리턴값이 변하지 않으면 조향 장치가 한쪽 방향으로 완전히 이동한 상황이고, 이때 조향 모터를 중지시킨 후 Right_Count에 현재 각도를 저장한다.

```
//모터를 한쪽으로 끝까지 돌린다.
nxt_motor_set_speed(NXT_PORT_A, 35, 1);
//더이상 회전하지 못하면 끝이라고 인식하고
 while문을 끝낸다.
while (temp != nxt_motor_get_count(NXT_PORT_A))
{
  temp = nxt_motor_get_count(NXT_PORT_A);
  systick_wait_ms(50);
}
//더이상 돌지 못하면 속도를 0으로하고
 Right_Count에 get_count를 통해서
//얼마나 돌았는지 저장한다.
nxt_motor_set_speed(NXT_PORT_A, 0, 1);
Right_Count = nxt_motor_get_count(NXT_PORT_A);
```

Initialize Task 코드 중

마찬가지 방법으로 Left_Count를 저장한 후, Count_temp에 중앙을 나타내는 각도를 저장한다. 그 후, nxt_motor_get_count()의 리턴값이 Count_temp와 일치할 때까지 조향 모터를 작동시킨다. while문을 탈출하면 조향 모터를 정지시키고 현재 위치를 0으로 재설정해준다. 이제 조향 모터는 전방을 가리키고 0으로 재설정 된 상태다. 또한 앞차의 경우 조향장치를 중앙으로 이동하여도 조향장치가 좌측으로 조금 흔들리는 모습을 보였고 직진 시에 이 영향으로 조금 좌측으로 이동하는 모습을 보였다. 따라서 조향장치의 set_count(2)를 넣어서 반대방향으로 아주 조금 이동하게 했고 이후에는 직선 주행이 가능했다.

```
Left_Count = nxt_motor_get_count(NXT_PORT_A);
//Count_temp에 목표 각도를 저장한다.
Count_temp = (Right_Count - Left_Count) / 2 + Left_Count;
nxt_motor_set_speed(NXT_PORT_A, 35, 1);
temp = nxt_motor_get_count(NXT_PORT_A);
//현재각도가 목표 각도가 될때까지 이동시킨다.
while (temp != Count_temp)
{
  temp = nxt_motor_get_count(NXT_PORT_A);
}
```

Initialize Task 코드 중

```
//조향 측정이 완료된 A포트를 정지한다.
nxt_motor_set_speed(NXT_PORT_A, 0, 1);
nxt_motor_set_count(NXT_PORT_A, 2);
```

앞차

```
//조향 측정이 완료된 A포트를 정지한다.
nxt_motor_set_speed(NXT_PORT_A, 0, 1);
nxt_motor_set_count(NXT_PORT_A, 0);
```

뒷차

조향 장치 설정이 마무리되면, 배터리 잔량에 따른 후륜 모터 속도 보정이 이루어진다. 양쪽 후륜 모터의 revolution count값을 0으로 초기화한 후, 0.5초 동안 최대 출력을 준 후 그때의 회전각을 구해 이전에 사용했던 Right_Count와 Left_Count에 저장한다. 임베디드 시스템 특성상 메모리 할당을 최소화하고자 다른 목적으로 사용했던 변수를 재사용했다.

```
//B와 C포트에 최대출력을 0.5초동안 준다.
nxt_motor_set_speed(NXT_PORT_B, 0, 1);
nxt_motor_set_speed(NXT_PORT_C, 0, 1);
nxt_motor_set_count(NXT_PORT_B, 0);
nxt_motor_set_count(NXT_PORT_C, 0);
systick_wait_ms(500);
nxt_motor_set_speed(NXT_PORT_B, 100, 1);
nxt_motor_set_speed(NXT_PORT_C, 100, 1);
systick_wait_ms(1000);

//그 때의 각도를 저장한다.
Right_Count = nxt_motor_get_count(NXT_PORT_B) / 10;
Left_Count = nxt_motor_get_count(NXT_PORT_C) / 10;
```

Initialize Task 코드 중

다양한 배터리를 사용해 배터리 잔량을 바꿔가며 실험한 결과 Steering이라는 변수에 배터리 잔량에 따른 모터 보정 상수를 계산한 결과를 넣었다. 실험 결과에 따라 Steering은 아래와 같이 정의된다.

```
Steering = 38 + ((Right_Count + Left_Count) / 2) * 2;
```

Steering에 저장된 값은 이후 Motor_Run_Fun() 함수에서 사용된다.

4.1.2. SonarSensing

구현 단계에서는 먼저 연속적인 값을 확인하기 위해 Sonar_Check라는 전역 변수를 사용했다. 범위에 맞는 값이 들어오면 이 변수의 값을 하나 줄이고 범위가 아닐 경우 다시 초기화해서 연속적인 값인지를 확인했다. 범위에 맞고 연속적인 값이면 해당하는 값을 Actual_Sonar_Num_1에 넣어준다.

```
//해당 초음파 센서의 값이 이전 초음파 센서의 값과 같고(연속적이고)
// 초기 값인 0이 아니면 유효한 값으로 판단한다.
if (Sonar_1 == temp && temp != 0)
{
  //Sonar_Check_1는 연속적으로 초음파 센서의 값이 같은 경우를 확인한다.
  --Sonar_Check_1;
  if (Sonar_Check_1 == 0)
  {
    Actual_Sonar_Num_1 = temp;
    Sonar_Check_1 = 1;
  }
}
```

SonarSensing 코드 중

이전 값과 같지 않은 경우 이후의 비교를 위해서 현재 값은 Sonar_1에 저장한다.

```
//만약 다를경우 지금 초음파 센서의 값을 이후의 비교를 위해 저장한다.
else
{
  Sonar_1 = temp;
  Sonar_Check_1 = 2;
}
```

SonarSensing 코드 중

Active_Only_One_Sonar	0 혹은 1	한번에 하나의 센서가 실행 되도록 한다.
temp	기계에서 받은 센서 값	기계에서 받은 센서의 값을 임시로 저장한다.
Sonar_1	이전의 센서 값	이후의 센서 값과의 비교를 위한 이전 센서의 값
Sonar_Check_1	0~2	연속적인 것을 확인하기 위한 변수
Actual_Sonar_Num_1	확실한 센서 값	범위 내에 있고 연속적인 것이 확인된 센서 값 = 유효한 센서 값

4.1.3. EventDispatcher

EventDispatcher는 차량 구동의 전반적인 처리를 담당한다. 앞차와 뒷차의 코드는 상당 부분 일치하며 뒷차의 코드는 bt_packet을 변경하는 형태로 조향장치의 이동과 모터 구동을 한다. 앞차의 모든 코드는 뒷차 코드에 포함된다.

이 task는 크게 두 부분으로 나눌 수 있는데, 뒷차의 초음파 센서에서 읽어온 값을 바탕으로 조향 모터와 후륜모터의 출력을 결정하는 부분과, 역시 센서값에 따라 조향장치 방향을 결정해 차를 좌우 회전시키는 부분이다.

먼저 초음파 센서의 값을 비교해서 조향장치를 움직일 지 확인하는 부분이다. 초음파 센서 값의 차이는 앞차와 뒷차의 차이에 따라서 달라진다. 다시말해 두 차 사이의 거리가 가까우면 각도가 커져도 초음파 센서간의 값 차이는 많이 나지 않았고, 두 차 사이의 거리가 멀어지면 각도에 따라서 초음파 센서의 값 차이도 커졌다. 이런 상황에 대응하기 위해 처음에는 조향장치 이동의 범위를 나누에서 구현하였다. 먼저 두 센서의 값이 모두 23보다 작은 범위와 그 이외의 범위로 나누었다.

```
/*
if(Actual_Sonar_Num_1-Actual_Sonar_Num_2>
  3 && Actual_Sonar_Num_1-Actual_Sonar_Num_2< 7 &&
  Actual_Sonar_Num_2<=23 && Actual_Sonar_Num_1<=23)
{
 bt_receive_buf[4] = 3;
}
else
  if(Actual_Sonar_Num_2-Actual_Sonar_Num_1> 3 &&
  Actual_Sonar_Num_2-Actual_Sonar_Num_1< 7 &&
  Actual_Sonar_Num_2<=23 && Actual_Sonar_Num_1<=23)
{
 bt_receive_buf[4] = 4;
}
*/
```

EventDispatcher 코드 중

하지만 이 범위는 최종적으로 위의 범위를 아래 범위가 모두 포함한다는 것을 확인했다. 따라서 이는 성공적이지 못했고 아래의 범위에서 조향장치를 움직이게 설계하였다.

결과적으로 조향장치를 이동하는 부분은 아래의 부분이며 조향장치를 이동하는 경우는 한쪽 초음파 센서의 값만 존재하는 경우 그 방향으로 이동하였고, 조향장치가 너무 자주 작동하면 직선 주행에 방해가 되는 동시에 방향 변경을 너무 자주하게 되어서 두 센서의 값이 4이상 차이가 나는 경우에만 조향장치가 움직이게 설계하였다.

```
else if (Actual_Sonar_Num_1 - Actual_Sonar_Num_2 > 3 ||
(Actual_Sonar_Num_1 == 100 && Actual_Sonar_Num_2 < 50))
 {
   bt_receive_buf[4] = 3;
 }
 else if
  (Actual_Sonar_Num_2 - Actual_Sonar_Num_1 > 3 || (Actual_Sonar_Num_2 ==
100 && Actual_Sonar_Num_1 < 50))
 {
   bt_receive_buf[4] = 4;
 }
 else if
  (Actual_Sonar_Num_2 < 50 && Actual_Sonar_Num_1 < 50)
 {
   bt_receive_buf[4] = 0;
 }
```

EventDispatcher 코드 중

또한 else if 로 끝나고 else가 없는 이유는 한쪽 방향으로 이동중에는 앞차의 회전에 의해서 두 초음파 센서의 값 모두를 못 받는 경우가 존재해서 해당 방향으로 계속 이동하도록 설계하였다.

초음파 센서값의 크기에 따른 속도 조절의 경우, 먼저 두 초음파 센서의 값이 모두 존재하는 경우 모터를 작동시킨다. 앞차와 뒷차의 거리가 21~25사이의 경우 저속으로 주행한다.

두 차 사이의 거리가 25보다 멀리 떨어져 있을 경우 고속으로 변경한다.

```
  //거리가 멀지 않고 적당히 떨어져 있으면 저속으로 주행
   if ((Actual_Sonar_Num_1 + Actual_Sonar_Num_2) / 2 < 26 &&
(Actual_Sonar_Num_1 + Actual_Sonar_Num_2) / 2 > 20)
   {
     Current_speed = Speed_L;
     Motor_Run = 1;
     bt_receive_buf[3] =1;
   }
```

EventDispatcher 코드 중

두 범위에 있지 않은 경우 가까이 있다고 판단하고 정지시킨다.

```
   //그 이외의 경우는 가까운 경우라고 인식하고 정지
   else
   {
     Current_speed = 0;
     Motor_Run = 0;
     nxt_motor_set_speed(NXT_PORT_B, 0, 1);
     nxt_motor_set_speed(NXT_PORT_C, 0, 1);
     bt_receive_buf[3] = 0;
   }
```

EventDispatcher 코드 중

만약 한쪽 센서의 값만 존재한다면 회전하는 경우로 판단하고 고속으로 회전 시 충돌 위험이 있기 때문에 저속으로 회전하도록 하였다.

```
    else if
  (Actual_Sonar_Num_1 < 50 || Actual_Sonar_Num_2 < 50)
 {
   Current_speed = Speed_L;
   Motor_Run = 1;
   bt_receive_buf[3] = 1;
   Terminate_Check = 0;
 }
```

EventDispatcher 코드 중

센서에서 어떠한 값도 새로 들어오지 않는다면 연속적인 700카운트, 즉 7초 동안 값이 들어오는지 확인한다. 700카운트동안 새로운 값을 받지 못하면 앞차를 놓쳤다고 판단하고 속도를 0으로 하고 소리를 발생시킨다.

```
 else
 {
   ++Terminate_Check;
   if (Terminate_Check == 700)
   {
     Current_speed = 0;
     Motor_Run = 0;
     nxt_motor_set_speed(NXT_PORT_B, 0, 1);
     nxt_motor_set_speed(NXT_PORT_C, 0, 1);
     bt_receive_buf[3] =
  0;
     ecrobot_sound_tone(1500, 80, 80);
   }
}
```

EventDispatcher 코드 중

이어서 앞차와 뒷차에 공통으로 들어있는 실질적인 조향장치의 이동과 모터의 구동이다.
먼저 조향장치의 경우 원래 조향장치를 이동하는 부분과 원래 위치로 다시 돌아오는 부분으로 나누어진다.
조향장치가 이동하는 부분부터 설명하자면 조향장치가 이동해야하는 상황이면 LR_Way에 100혹은 -100이 들어온다. 이 상황에서 아직 조향장치는 이동하지 않았기 때문에 get_count는 0이다.

```
 err = (50 * LR_Way) / 100 - nxt_motor_get_count(NXT_PORT_A);
```

err 값에 대해 설명하자면 50에 해당하는 값이 최대 각도에 해당한다. LR_Way에 100 혹은 -100이 들어오면 /100과 약분되고 50만 남는다. 이 err인 50은 이후에 속도로 들어가게 된다. 이는 조향장치가 돌기 시작하는 것을 의미하고 조향장치의 움직임에 따라서 get_count값은 점점 변화한다. 조향장치가 이동하다가 get_count값이 50이 되면 err이 0이 된다. 그럼 모터의 속도를 0으로 바꿔 주게 되어서 조향장치의 이동은 중지한다.
다음으로 조향장치가 원래 위치로 돌아오는 방법은 err와 get_count에 의해서 결정된다. 조향장치에 좌우 돌아가는 입력이 들어오지 않으면 LR_Way가 0이 된다. 그렇게 되면 err는 get_count의 반대부호가 되고 조향장치는 이전에 돌아간 방향의 반대 방향으로 이동하게 된다. 이동하다가 get_count가 0이 되는순간 err도 0이 되고 조향장치는 원래 위치로 돌아온다.

상태	변수 값 변화
직진	err = 0 조향장치 이동 없음
오른쪽으로 이동 시작	LR_Way = 100 get_count = 0 err = 50 err에 의해서 조향장치가 오른쪽으로 이동
오른쪽으로 최대 각도인 50 만큼 이동	LR_Way = 100 get_count = 50 err = 0 더 이상 갈 수 없기 때문에 조향장치의 이동이 중지
조향 장치가 오른쪽으로 이동 후 직진 시작	LR_Way = 0 get_count = 50 err = -50 조향장치가 반대 방향으로 이동

조향 장치가 오른쪽으로 이동하다 원래 위치에 도착	LR_Way = 0 get_count = 0 err = 0 조향장치의 이동이 중지

고속/저속 주행

고속/저속 주행은 앞차와 뒷차 코드에 모두 있는 부분이다. 하지만 블루투스 버튼을 조작할 때 활성화되기 때문에 사용자가 조작할 수 있는 건 앞차뿐이고, 뒷차는 앞차의 속도에 따라 수동적으로 고속과 저속모드 사이에서 변화한다. 천천히 정지 시 완전히 정지하기 때문에 모터를 다시 시작하기 위해 저속에서 고속으로 혹은 반대 경우를 확인해야 한다. 따라서 이러한 경우를 확인하고 모터 회전을 담당하는 변수인 Motor_Run을 1로 만들고 event1을 발생시켜서 속도 변환, 모터 즉시 동작을 실행한다.

즉시/천천히 정지

즉시 정지의 경우 정지를 담당하는 task인 BrakeTask를 event2를 발생시켜서 부른다. Brake 변수를 통해 어떠한 정지를 해야하는 지 저장하고 event2를 발생시켜서 각 상황에 맞는 정지 동작이 일어나도록 한다.

천천히 정지의 경우 BrakeTask에서 Slow_brake에 값을 넣어주면 시간이 흐름에 따라서 현재 속도보다 점점 더 느린 속도를 넣어주는 방식으로 구현했다.이를 구현한 방법은 Slow_brake라는 변수를 현재 속도를 기준으로 만들고 이를 현재 속도에서 조금씩 빼주는 방식으로 구현했다. 중간에 temp_1이라는 변수를 넣었고 Slow_brake라는 변수는 실행되면서 1씩 줄어들게 했다.

```
int temp_1;
 if (Current_speed == Speed_H)
 {
   temp_1 = (Speed_H * 2 + 1 - Slow_brake) / 2;
 }
```

EventDispatcher **코드 중**

이후에 temp_1의 값을 현재 속도에 빼서 속도를 느리게 한다.

```
  nxt_motor_set_speed(NXT_PORT_B, -Current_speed + temp_1, 1);
  nxt_motor_set_speed(NXT_PORT_C, -Current_speed + temp_1, 1);
```

EventDispatcher **코드 중**

직진

직진의 경우 Motor_Run 값을 통해 확인한다. 이 값이 1이면 모터를 실행 함수인 Motor_Run_Fun()을 실행해서 모터를 실행시킨다.

4.1.4. SpeedTask

SpeedTask는 event1을 통해 호출할 수 있다. 이 Task는 후륜 모터의 속도 변경을 관장하는 Task로 고속/저속 변환과 회전 시 ECU가 적용되도록 한다.

조건문을 이용해 현재 속도가 저속일 경우 고속으로, 현재 속도가 고속일 경우 저속으로 토글해준다. 그리고 마지막에 토글을 통해 변경된 속도에 맞춰 ECU가 적용되도록 Motor_Run_Fun()을 호출한다.

```
TASK(SpeedTask)
{
 while (1)
 {
   WaitEvent(event1);
   ClearEvent(event1);
   GetResource(R1);
   if (Current_speed == Speed_L)
   {
     Current_speed = Speed_H;
   }
   else if (Current_speed == Speed_H)
   {
     Current_speed = Speed_L;
   }
   Motor_Run_Fun(Pre_Order);
   ReleaseResource(R1);
 }
 TerminateTask();
}
```

SpeedTask **코드**

4.1.5. BrakeTask

BreakTask는 event2를 통해 호출할 수 있다. 이 Task는 현재 지정된 차량의 정지 방식에 따라 차량이 정지하도록 한다.

```
TASK(BrakeTask)
{
 while (1)
 {
   WaitEvent(event2);
   ClearEvent(event2);
   GetResource(R1);
   if (Brake == 1)
   {
     nxt_motor_set_speed(NXT_PORT_B, 0, 1);
     nxt_motor_set_speed(NXT_PORT_C, 0, 1);
   }
   else if (Brake == 2)
   {
     if (Current_speed == Speed_H && Motor_Run == 1)
     {
       Slow_brake = Speed_H * 2;
     }
     else if (Current_speed == Speed_L && Motor_Run == 1)
     {
       Slow_brake = Speed_L * 2;
     }
     Motor_Run = 0;
   }
   ReleaseResource(R1);
 }
 TerminateTask();
}
```

BrakeTask **코드**

전역 변수인 Brake 변수는 정지 방식에 따라 하드 브레이크일 경우 1, 슬로우 브레이크일 경우 2를 저장하고 있다. event2로 BrakeTask가 호출되면 조건문을 이용해 Brake 변수에 저장된 값을 확인한다. 현재 정지 방식이 하드 브레이크일 경우 nxt_motor_set_speed() 함수를 이용해 하드 브레이크가 작동되도록 한다. 소프트 브레이크일 경우는 현재 주행속도 모드가 고속인지, 저속인지 확인 후 각 모드에 따라 EventDispatcher task에서 temp_1으로 속도를 조절해 준다.

4.2. Functions

4.2.1. Motor_Run_Fun(U8 buf)

이 함수는 모터를 실행시킬 때 사용하는 함수이다. Initialize에서 구한 Steering 변수는 여기서 사용한다. Steering이 의미하는 것은 50%, 60%와 같이 %를 의미한다. 따라서 vel값을 선언하고 이 vel 값에 현재 속도*Steering/100을 해서 현재 속도에 맞는 보정 속도를 구한다.

```
vel = Current_speed * Steering / 100;
```

이후에 LR_Way를 통해 조향장치가 작동중인지를 확인하고 작동중이라면 안쪽 바퀴의 속도는 vel로 하고 바깥쪽 바퀴의 속도는 현재 속도로 한다. 회전시에는 속도가 조금 더 느려지기 때문에 속도를 10% 더 증가시켰다. 마지막으로 LR_Way가 100/-100이 아닌 경우, 다시 말해 else에 해당하는 부분은, 직선 주행에 해당한다. 이 경우는 좌우 바퀴에 모두 현재 속도를 넣고 left에는 속도를 10%더 높게 한다. 이는 여러번의 테스트를 통해 NXT가 직선 주행을 하기 위해 두 바퀴 사이의 속도 차이가 얼마나 있는지 확인하였다. 가장 직선에 가깝게 운행하기 위해서 left에 10%의 속도를 더해주는 것이라는 결론을 얻었다.

```
if (LR_Way == 100)
 {
   left = vel + Current_speed / 10;
   right = Current_speed + Current_speed / 10;
 }
 else if
  (LR_Way == -100)
 {
   left = Current_speed + Current_speed / 10;
   right = vel + Current_speed / 10;
 }
 else
 {
   right = Current_speed;
   left = Current_speed + Current_speed / 10;
 }
```

Motor_Run_Fun(U8 buf) **코드 중**

4.3. Priority 배정

Task 들의 priority는 EventDispatcher가 4, SonarSensing이 5, Initialize가 6, SpeedTask가 7, 마지막으로 BrakeTask가 가장 높은 8이다. AutoStart가 True인 Task들 사이에는 resource를 주어서 이들 Task들 사이에는 동시 실행이 불가능하게 하였고, 이는 BrakeTask의 실행을 보호하는 목적과 Initialize에서 코드가 한번에 실행되는 것을 보호하기 위해서이다.
5ms 마다 EventDispatcher가 실행되고, 12ms마다 SonarSensing이 실행된다. 처음에는 겹치지 않지만 실행 도중에 겹치게 되면 priority가 높은 SonarSensing이 먼저 실행되고 끝난 후에 EventDispatcher가 실행된다.