ROBIT 예비단원 최종 프로젝트 보고서

지능팀 18기 예비단원 주성민

[목차]

1. 임베디드 통신 과정
    a. 주요 개념
    b. Write
    c. Read
2. 스마트 액추에이터, 다이나믹셀
    a. 매니퓰레이터 구동을 위한 패킷 구성
    b. 팬틸트

[본문**]**

1. 임베디드 통신 과정

A. 주요 개념

- **Baud Rate**

보 레이트는 초당 얼만큼의 심볼을 보낼 수 있는지를 나타내는 수치이다. 쉽게 말하면 이론적으로 통신 단위로 초당 신호의 요소의 수를 의미한다. 심볼이란, 의미 있는 데이터 묶음으로 시리얼 통신에서는 data bit가 8bit를 사용하므로 이를 하나의 심볼이라고 할 수 있다. => 1개의 심볼 = 8bit의 정보

ex) baud rate: 57600이 갖는 의미는 초당 57600개의 심볼, 의미 있는 데이터를 전송할 수 있다는 의미이다.

- **loopRate**

ros::Rate는 ROS에서 제공하는 시간 기반 제어 클래스로 이를 이용하면 프로그램 실행 주기를 제어할 수 있다. ros::Rate loopRate(rosrate)에서 rosrate는 일정한 주기로 루프를 실행하기 위한 초당 반복 횟수이다. 예를 들어 rosrate가 80이면 1초에 80번 루프가 실행된다.

- **Parsing**

송신한 패킷을 수신 받을 때 패킷 검사 작업을 진행한다. 패킷의 헤더와 crc의 값을 확인한 후 데이터 손상이 이루어지지 않았다고 판단되면 헤더와 crc 사이에 위치하는

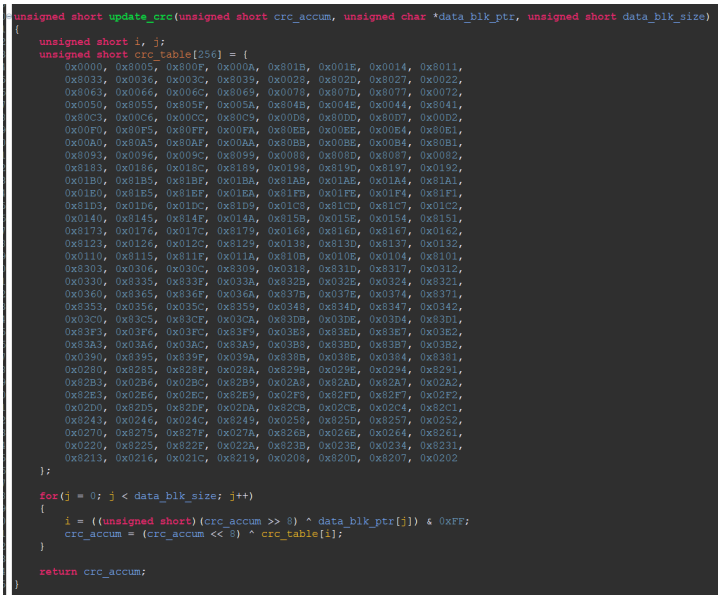
데이터 값만 추출하는 과정이다. 해당 데이터를 가공하거나 알맞은 형태에 맞게 사용할 수 있다.

- **Mutex**

여러 개의 프로세스가 동시에 공유된 자원에 접근하면 문제가 발생할 수 있다. 예를 들어 두 개의 프로세스 A, B가 변수 x에 관여한다고 해보자. A는 x를 0으로, B는 y를 1로 변경한다고 하면 두 과정은 충돌하게 된다. 따라서 하나의 프로세스만이 하나의 변수에 접근할 수 있도록 하는 작업이 필요하다. Mutex란 mutual exclusion의 줄임말로 여러 프로세스가 공유된 자원에 동시에 접근하는 것을 막는 기술이다. 프로세스가 자원에 접근하고 싶으면 'lock'을 얻어야 접근이 가능하고 lock이 반환되기 전까진 다른 프로세스가 자원에 접근하지 못한다. 따라서 하나의 프로세스만이 공유된 자원에 접근할 수 있게 된다.

- **CRC(Cyclic Redundancy Checking)**

CRC는 순환 중복 검사로 에러 유무를 판단할 수 있다. 이는 데이터 링크 계층에서 많이 사용되는 비트 지향적인 에러 검출 기술로 송신측에서 crc 계산법에 의해 계산한 후 결과값을 송신하고 수신측에서는 같은 계산 방식으로 계산을 해서 수신받은 값과 자체적으로 계산한 값이 같으면 올바른 패킷으로 인식을 하고 데이터를 받아들인다.
Crc 계산 과정은 다음과 같다
함수의 인자: 업데이트할 현재 crc 값, crc가 계산되어야하는 데이터 블록 즉 전송하는 패킷을 의미한다. 그리고 데이터 블록의 크기를 끝으로 세 정보를 가지고 crc를 계산하게 된다.



```
unsigned short update_crc(unsigned short crc_accum, unsigned char *data_blk_ptr, unsigned short data_blk_size)
{
    unsigned short i, j;
    unsigned short crc_table[256] = {
        0x0000, 0x8005, 0x800F, 0x000A, 0x801B, 0x001E, 0x0014, 0x8011,
        0x8033, 0x0036, 0x003C, 0x8039, 0x0028, 0x802D, 0x8027, 0x0022,
        0x8063, 0x0066, 0x006C, 0x8069, 0x0078, 0x807D, 0x8077, 0x0072,
        0x0050, 0x8055, 0x805F, 0x005A, 0x804B, 0x004E, 0x0044, 0x8041,
        0x80C3, 0x00C6, 0x00CC, 0x80C9, 0x00D8, 0x80DD, 0x80D7, 0x00D2,
        0x00F0, 0x80F5, 0x80FF, 0x00FA, 0x80EB, 0x00EE, 0x00E4, 0x80E1,
        0x00A0, 0x80A5, 0x80AF, 0x00AA, 0x80BB, 0x00BE, 0x00B4, 0x80B1,
        0x8093, 0x0096, 0x009C, 0x8099, 0x0088, 0x808D, 0x8087, 0x0082,
        0x8183, 0x0186, 0x018C, 0x8189, 0x0198, 0x819D, 0x8197, 0x0192,
        0x01B0, 0x81B5, 0x81BF, 0x01BA, 0x81AB, 0x01AE, 0x01A4, 0x81A1,
        0x01E0, 0x81E5, 0x81EF, 0x01EA, 0x81FB, 0x01FE, 0x01F4, 0x81F1,
        0x81D3, 0x01D6, 0x01DC, 0x81D9, 0x01C8, 0x81CD, 0x81C7, 0x01C2,
        0x0140, 0x8145, 0x814F, 0x014A, 0x815B, 0x015E, 0x0154, 0x8151,
        0x8173, 0x0176, 0x017C, 0x8179, 0x0168, 0x816D, 0x8167, 0x0162,
        0x8123, 0x0126, 0x012C, 0x8129, 0x0138, 0x813D, 0x8137, 0x0132,
        0x0110, 0x8115, 0x811F, 0x011A, 0x810B, 0x010E, 0x0104, 0x8101,
        0x8303, 0x0306, 0x030C, 0x8309, 0x0318, 0x831D, 0x8317, 0x0312,
        0x0330, 0x8335, 0x833F, 0x033A, 0x832B, 0x032E, 0x0324, 0x8321,
        0x0360, 0x8365, 0x836F, 0x036A, 0x837B, 0x037E, 0x0374, 0x8371,
        0x8353, 0x0356, 0x035C, 0x8359, 0x0348, 0x834D, 0x8347, 0x0342,
        0x03C0, 0x83C5, 0x83CF, 0x03CA, 0x83DB, 0x03DE, 0x03D4, 0x83D1,
        0x83F3, 0x03F6, 0x03FC, 0x83F9, 0x03E8, 0x83ED, 0x83E7, 0x03E2,
        0x83A3, 0x03A6, 0x03AC, 0x83A9, 0x03B8, 0x83BD, 0x83B7, 0x03B2,
        0x0390, 0x8395, 0x839F, 0x039A, 0x838B, 0x038E, 0x0384, 0x8381,
        0x0280, 0x8285, 0x828F, 0x028A, 0x829B, 0x029E, 0x0294, 0x8291,
        0x82B3, 0x02B6, 0x02BC, 0x82B9, 0x02A8, 0x82AD, 0x82A7, 0x02A2,
        0x82E3, 0x02E6, 0x02EC, 0x82E9, 0x02F8, 0x82FD, 0x82F7, 0x02F2,
        0x02D0, 0x82D5, 0x82DF, 0x02DA, 0x82CB, 0x02CE, 0x02C4, 0x82C1,
        0x8243, 0x0246, 0x024C, 0x8249, 0x0258, 0x825D, 0x8257, 0x0252,
        0x0270, 0x8275, 0x827F, 0x027A, 0x826B, 0x026E, 0x0264, 0x8261,
        0x0220, 0x8225, 0x822F, 0x022A, 0x823B, 0x023E, 0x0234, 0x8231,
        0x8213, 0x0216, 0x021C, 0x8219, 0x0208, 0x820D, 0x8207, 0x0202
    };

    for(j = 0; j < data_blk_size; j++)
    {
        i = ((unsigned short)(crc_accum >> 8) ^ data_blk_ptr[j]) & 0xFF;
        crc_accum = (crc_accum << 8) ^ crc_table[i];
    }

    return crc_accum;
}
```

함수의 실행
- j를 이용하여 전송하는 패킷의 크기만큼 각 바이트를 반복
- 각 바이트에 대해 현재 crc 값, 전송하는 패킷의 현재 위치에 있는 바이트와 0xFF와 XOR 비트 연산을 사용하여 i를 계산
- 현재 crc 값을 8비트 왼쪽으로 시프트 연산을 하고 crc 테이블에서 계산된 i번째 값과 XOR하여 최종 업데이트된 crc를 반환

이러한 과정을 거쳐 전송하는 패킷에 알맞는 crc 값이 나오고 이 값을 이용하여 오류 여부를 확인한다.

[임베디드 통신**]**
다이나믹셀: 'hello'라는 토픽명으로 4개의 goal position과 1개의 goal velocity의 값을 publish 받는다.

바퀴 모터
1. 자율 주행의 모드인 경우(mode_flag == 0), 'bye'라는 토픽명으로 2개의 값을 publish 받는다.
2. 수동 주행의 모드인 경우(mode_flag == 1), 'hello'라는 토픽명으로 2개의 값을 publish 받는다.

B. Write

Node.cpp에서 sub 선언 + node.cpp와 test.cpp의 연결
우선 조이스틱 값을 다이나믹셀의 goal position으로 변환된 값을 토픽으로 publish를 하게 된다. 시리얼 통신하게 될 패키지는 이를 subscribe하게 되고 sub받은 값을 송신하는 과정으로 write가 진행된다. Test.cpp에 패킷을 보내는 코드가 있고 node.cpp에서는 퍼블리시, 서브스크라이브 선언만을 할 수 있다. 평소 서브스크라이브 선언을 qnode.cpp에서 다음과 같이 했다면

```
sub = n.subscribe("/String_topic", 10, &QNode::myCallBack, this);
```

Node.cpp에서 sub를 선언하지만 callback함수는 Test클래스에서 정의되어있으므로

```
sub = n.subscribe("/hello", 10, &Test::myCallBack, &test);
sub_autorace = n.subscribe("/bye", 10, &Test::myCallBack_autorace, &test);
sub_mode = n.subscribe("/mode", 10, &Test::myCallBack_mode, &test);
```

콜백함수의 주소를 test로 설정을 해준다.

Subscriber sub_mode는 sub와 sub_autorace가 어떤 값을 받는지 정해주는 플래그로 1일 땐 hello로 다이나믹셀과 조이스틱으로부터의 모터 값을 모두 받고 0일 땐 hello로 다이나믹셀 모터 값 5개와 bye로 비전으로부터 얻은 자율주행 모터 값 2개를 받아온다.

- Test.cpp에서 콜백함수로 값을 sub, stm_write.cpp에서 test의 변수 사용

콜백함수는 test노드에서 실행이 되고 sub를 받은 값은 결국 write 노드에서 송신을 하게 된다. 다른 노드에서 같은 변수를 사용하는 방법에는 여러 가지가 있지만 그 중 한가지는 다른 노드에서 클래스 형 변수를 선언하여 그 클래스를 참조하는 것이다. 각 노드에서 선언된 헤더파일들을 살펴보면 write 노드에서는 header.h가 선언되어있고, test 노드에서는 header.h와 stm_write.h가 선언되어있다. Test에서 write 노드의 헤더파일을 선언해서 해당 클래스 안의 변수와 함수를 이용할 수 있는 환경이 갖춰져 있다. 따라서 test노드에서 write 노드의 클래스인 StmPacketGenerator형 StmPacketGenerator를 선언하고 StmPacketGenerator.(write 노드의 변수)의 형태로 두 노드간의 변수를 연결할 수 있다.

아래의 사진은 write노드에서 선언한 변수를 test 노드에서 write의 클래스형 변수를 선언하여 값을 받는 코드이다.

```
class Test
{
private:
  StmPacketGenerator stmPacketGenerator;
  StmPacketTranslator stmPacketTranslator;
```

```
void Test::myCallBack(const tutorial_msgs::mydmxelConstPtr& msg)
{
  if(mode_flag == 1)
  {
    stmPacketGenerator.ID_1 = msg->motor1;
    stmPacketGenerator.ID_2 = msg->motor2;
    stmPacketGenerator.ID_3 = msg->motor3;
    stmPacketGenerator.ID_5 = msg->motor4;
    stmPacketGenerator.ID_6 = msg->motor5;

    stmPacketGenerator.left_wheel = msg->L_wheel;
    stmPacketGenerator.right_wheel = msg->R_wheel;
  }
  else if(mode_flag == 0)
  {
    stmPacketGenerator.ID_1 = msg->motor1;
    stmPacketGenerator.ID_2 = msg->motor2;
    stmPacketGenerator.ID_3 = msg->motor3;
    stmPacketGenerator.ID_5 = msg->motor4;
    stmPacketGenerator.ID_6 = msg->motor5;
  }

}
```

```
void Test::myCallBack_autorace(const tutorial_msgs::mydmxelConstPtr& msg)
{
  if(mode_flag == 1)
  {

  }
  else if (mode_flag == 0)
  {
    stmPacketGenerator.left_wheel = msg->L_wheel;
    stmPacketGenerator.right_wheel = msg->R_wheel;
  }

}
```

- 패킷의 구성

Write 노드는 패킷을 구성하는 노드이다. 패킷은 기본적으로 헤더와 의미 있는 데이터, 체크섬으로 이루어져있다. 이 패키지에서는 헤더 네 개와 다이나믹셀 5개를 작동시키기 위한 goal position 값 5개, 모터값 2개 , crc 값 1개로 구성되어있다.
헤더는 직접 정할 수 있고, 데이터 7개도 서브스크라이브로 받은 값을 변수에 저장하고 해당 변수를 송신하면 된다. 또한 crc도 1번에서 설명한 계산 방식으로 값을 특정할 수 있다. 이렇게 해서 패킷이 구성되었지만 주의해야할 점이 있다. 송신을 할 수 있는 바이트는 최대 1 바이트로 8bit가 최대이다. 다이나믹셀의 goal position의 최댓값은 4095로 8 bit를 넘기고 crc 값도 패킷에 따라서 8bit를 넘길 수 있는 값이다. 따라서 이를 두 개의 바이트로 나누어서 보낼 필요가 있다.

```
void StmPacketGenerator::divideByte(vector<uint8_t> &packet, int value, int length)
{
  for(int i = 0; i < length; i++)
  {
    packet.push_back((value >> (i*8)) & 0xFF);
  }
}
```

위와 같은 함수를 사용하여 어떤 패킷의 어떤 값을 몇 개의 바이트로 나누어서 보낼 것인지를 적으면 해당 패킷에 바이트로 나뉜 값이 저장이 되고 송신한다. 따라서 subscribe로 받은 데이터 값은 7개지만 패킷에 5개는 8bit가 넘어서 위 함수를 거쳐 송신하고 모터 rpm 2개는 함수를 거치지 않고 송신하여 총 12개의 값을 송신한다..

- 패킷이 작성 되었으면 패킷 송신의 과정

```
bool Node::retry()
{
  if (!s.isOpen())
  {
    try
    {
      s.open();
    }
    catch (exception e)
    {
      return false;
    }
  }

  return true;
}

void Node::timerCallback()
{
  if(!retry())
  {
    return;
  }

  test.Algorithm_Test();
}
```

Try & catch 구문을 이용하여 예외처리를 하는 동시에 serial 패키지에서 시리얼 포트가 열린 상태가 되면 test 클래스에 있는 Algorithm_Test()라는 함수가 실행이 된다.

```
void Test::Algorithm_Test()
{
  switch (conversationStatus)
  {
  case CONVERSATION_REQUEST:
  {
    try
    {
      vector<uint8_t> packet;

      stmPacketGenerator.writePacket(packet);

      s->write(packet);
    }
    catch (serial::IOException e)
    {
      cerr << "Port open failed." << e.what() << endl;
      s->close();
    }

    conversationStatus = CONVERSATION_RESPONSE;

    break;
  }
```

Algorithm_Test()라는 함수는 열거형 변수 conversationStatus에 따라 수행하는 동작이 다르다. CONVERSATION_REQUEST 상태에서는 stmPacketGenerator.writePacket(packet)을 호출하여 패킷을 구성하고 시리얼 통신을 사용해서 데이터를 송신하고 CONVERSTATION_RESPONSE 상태로 전환된다. 예외가 발생하는 경우, 포트를 닫는 일을 한다.
이러한 과정을 거쳐 pc에서 stm으로 원하는 값을 패킷에 넣어 송신을 할 수 있다.

- 통신을 하기 전에 지켜야할 주의 사항

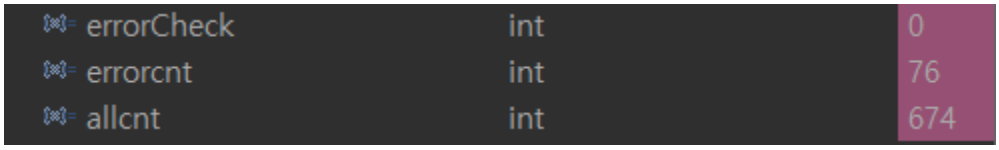
- User permission 얻기
  시리얼 포트에 대한 접근 권한을 얻기 위해서 user permission을 얻어야한다.
  [유저 퍼미션 얻는 법]
  Sudo usermod -aG dialout [username]
  Su - [username]
  Sudo gpasswd –add [username] dialout
- 터미널 창을 열어 ls /dev/ttyUSB*를 검색해서 포트 번호를 일치시켜주기
- 통신 속도인 보드레이트를 pc와 stm을 일치시켜주기

다이나믹셀 위자드 2.0에서 다이나믹셀의 통신 보드레이트를 1000000으로 설정해주었다. 최종 목표가 다이나믹셀을 작동시키는 것이라면 stm, pc 모두 다이나믹셀에 맞는 보드레이트로 설정해주어야한다.

[주의사항**]**
코드 상에서 port name, baud rate이 같아야한다.
[User dialout에 추가하는 방법]
Sudo usermod -aG dialout [username]
Su - [user name]
groups : user dialout에 추가 되었는지 확인 가능
Sudo gpasswd –add [user name] dialout

C. Read

Pc에서 보낸 패킷을 stm에서 수신을 받는다. 수신 받은 raw data를 일정한 크기를 가진 배열에 담고 패킷 검사를 진행한다. 패킷 검사를 진행하는 이유는 데이터의 무결성을 보장하고 신뢰성을 확보하기 위해서이다. 데이터 전송 도중 발생하는 노이즈, 간섭 등의 요인으로 데이터가 손상이 될 수 있다. 그러한 상태에서 패킷 검사를 진행하지 않고 그대로 하드웨어에 값을 입력하면 원하는 대로 모터가 돌지 않거나 하드웨어에 손상이 갈 수도 있다. 이러한 이유로 패킷을 검사하고 검사하는 방식에는 crc 순환 중복 검사와 체크섬 검사방식이 있다. 체크섬 방식은 패킷의 모든 요소를 다 더한 후 비트 반전 시키므로써 얻어지는 값을 비교하면서 진행된다. 간단한 오류 검출 방법이고 우연의 일치로 잘못된 값이 들어왔는데 체크섬이 일치할 수도 있어 주로 crc를 사용한다.
다음 사진은 항상 증가하는 allcnt와 잘못된 패킷이 들어왔을 때 증가하는 errorcnt의 값이다. 실제로 올바르지 않은 패킷이 들어오며 헤더와 crc를 이용한 검사작업이 필요한 이유이다.



1번에서 crc가 어떤 것이고 값을 어떤 방식으로 얻었는지에 대한 설명이 있었다면 crc검사를 이용하여 패킷을 검사하는 방법을 설명하고자 한다. 패킷은 기본적으로 헤더 +

데이터 + crc로 이루어져있다. 받고자하는 패킷의 헤더와 crc 값을 비교해 일치하면 데이터를 파싱하는 과정으로 이루어진다.
코드를 예시로 들어본다면 2. Write에서 보내고자하는 패킷의 헤더의 개수는 4개이고 원하는 데이터 값은 7개이지만 8bit가 넘는 경우를 대비해서 모든 값을 2바이트로 나누어서 보냈으므로 12개의 데이터가 패킷 속에 담겨있다. Crc 값은 헤더 4개 + 데이터 12개로 16 byte와 패킷을 이용해서 값이 나왔을 것이고 2바이트로 나뉘어 송신했을 것이다. 똑같이 readPacket이라는 함수를 만들어서

```
if(buffer[0] == 0xFF);
else errorCheck = 1;
if(buffer[1] == 0xFF);
else errorCheck = 1;
if(buffer[2] == 0xFD);
else errorCheck = 1;
if(buffer[3] == 0x00);
else errorCheck = 1;
```

다음과 같이 헤더 4개를 검사,

```
received_crc = (buffer[17] << 8) | buffer[16];//crc index
packet_crc = update_crc(0, buffer, 16);
```

다음과 같이 2바이트로 나뉘어서 온 crc 값을 한 바이트로 합치고 crc값을 수신측에서 자체적으로 계산을 한 뒤,

```
if((received_crc == packet_crc)&&(errorCheck == 0))
{
   for(int i = 0; i < 12; i++)
   {
      data[i] = buffer[4 + i];
   }
}
```

헤더가 하나라도 다르면 errorCheck는 1의 값을 가질 것이므로 errorCheck의 값이 0이고 수신받은 crc값과 자체적으로 계산한 crc 값이 같으면 다음과 같이 for문을 이용하여 원하는 데이터만 data 배열에 저장을 한다. 아직 원하는 값을 다 얻진 못했다. 송신을 할

때 2바이트로 나눠서 왔기에 data[0]과 data[1]이 하나의 값, data[2], data[3]이 하나의 값을 의미한다. 따라서

```
for(int i = 0; i < 5; i++)
      {
         restoredDXL[i] = (data[1 + (2 * i)] << 8) | data[2 * i];
      }
```

두 바이트의 값을 한 바이트로 만들어주는 수식을 이용해 restoredDXL에는 우리가 하고자 했던 다이나믹셀을 작동시키기 위한 5개의 값을 모두 받은 상태가 되고 파싱 작업에서 바이트로 쪼개서 온 데이터들을 하나의 바이트로 합쳐 goal position 값에 넣으면 된다. 그리고 data[10], data[11]는 각각 왼쪽 바퀴의 rpm, 오른쪽 바퀴의 rpm의 범위에 맞게 변환되어 모터에 입력된다.

```
//right
if (data[11] >= 10 && data[11] <= 20)
{
   target1 = (2) * (data[11] -10);//b
   target2 = (2.8) * (data[11] -10);//f
}
else if (data[11] >= 0 && data[11] <=10)
{
   target1 = (-1)* (2) * (10 - data[11]);
   target2 = (-1)* (2.8) * (10 - data[11]);
}
```

```
//left
if (data[10] >= 10 && data[10] <= 20)
{
   target1 = (2) * (data[10] -10);//b
   target2 = (2.8) * (data[10] -10);//f
}
else if (data[10] >= 0 && data[10] <=10)
{
   target1 = (-1)* (2) * (10 - data[10]);
   target2 = (-1)* (2.8) * (10 - data[10]);
}
```

3. 스마트 액추에이터**,** 다이나믹셀

a. 매니퓰레이터 구동을 위한 패킷 구성

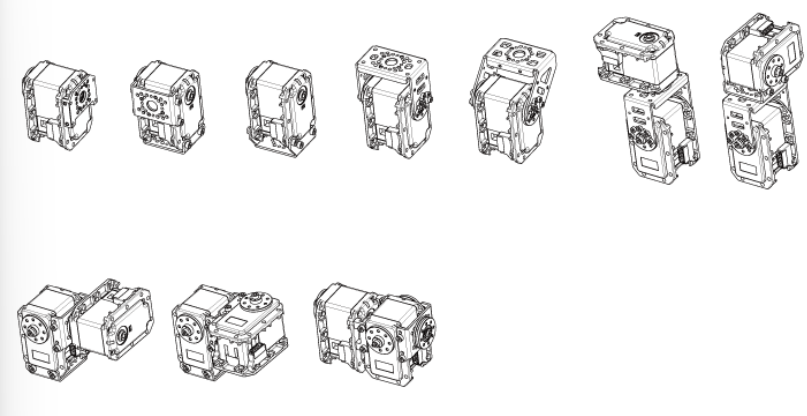
i. 다이나믹셀이란**?**

액추에이터란, 로봇의 관절에 사용되는 전동 모터로 로봇을 구동시키는 중요한 장치 중 하나이다. 로봇의 움직임이 자연스럽게 하기 위해서는 많은 관절이 필요하며 관절에 사용되는 모터는 기능이 좋아야한다. 정밀 제어, 네트워크 연결 등의 기능을 갖추는 것이 좋다. 다이나믹셀은 로보티즈라는 화사가 개발한 스마트 액추에이터로 '역동적'이라는 뜻을 가진 DYNAMIC과 '가장 작은 단위'를 일컫는 말인 CELL의 합성어이다. 토크, 속도, 크기, 통신 방식에 따라 여러 모델이 존재하며 이들은 각각 로봇팔, 의료용 장비, 매니퓰레이터 로봇에 이용된다.

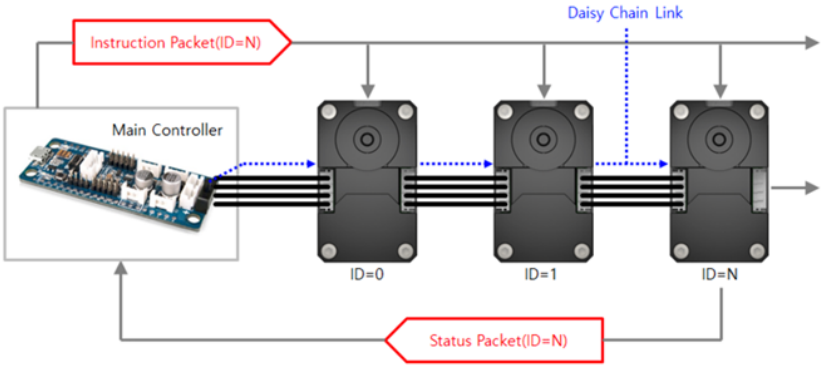
다이나믹셀의 특징은 다음과 같다.

1. 로봇 관절에 필요한 DC모터, 제어기, 드라이버, 센서, 감속기, 네트워크 기능을 모두 담은 일체형 구조이다.

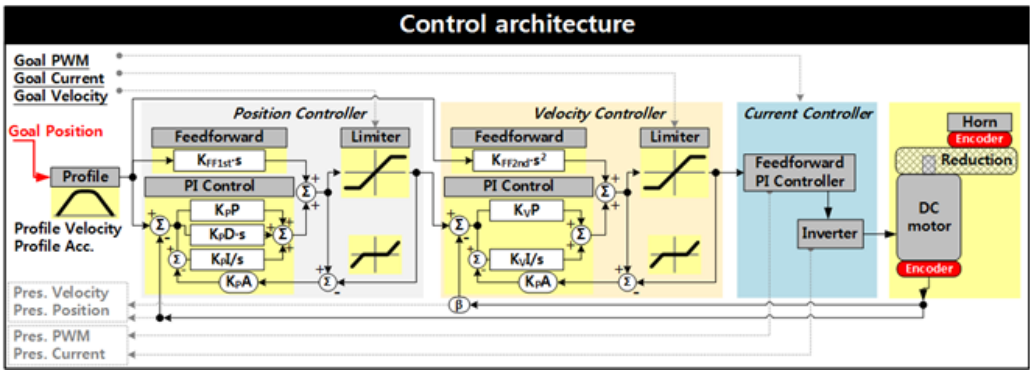
2. 다양한 방법으로 결합 가능한 구조를 갖추고 있다. 또한 공개된 설계 파일을 이용하여 다양한 형태로 로봇을 설계할 수 있다.



3. 여러 개의 다이나믹셀을 데이지 체인(연속적으로 연결되어 있는 장치들의 구성)의 형태를 이용하여 각각 다른 ID를 가진 모듈을 패킷 통신을 이용하여 제어할 수 있다. 또한 RS-485, TTL 등 다양한 통신을 지원한다.



4. PID, 위치, 속도, 토크 제어가 가능하다.



ii. 프로토콜 2.0 패킷의 구성

다이나믹셀은 메인컨트롤러와 패킷이라는 데이터를 주고 받으며 통신한다. 패킷에는 메인 컨트롤러에서 다이나믹셀을 제어하기 위해서 주는 데이터인 instruction packet과 다이나믹셀의 상태를 메인컨트롤러에서 알기 위해 받는 데이터인 status packet이 있다. Pc에서 MCU로 다이나믹셀의 goal position을 다이나믹셀에게 송신하기 위해선 프로토콜에 맞는 instruction 패킷을 구성하는 것이 우선이다. 프로토콜 2.0의 패킷은 Header, ID, Length, Instruction, Param, CRC로 구성되며 각각의 데이터는 알맞은 크기(바이트)로 보내야한다. 각각의 데이터마다의 전송 크기는 '로보티즈의 e-manual에서 control table'에서 참고 가능하다.

Header: 패킷의 시작을 알리는 데이터로 4바이트로 구성되며 이는 각각 0xFF,0xFF, 0xFD, 0x00이다.

ID: 한 개의 버스에서 여러 개의 다이나믹셀을 명령하기 위해선 각각의 다이나믹셀에 고유 번호인 ID를 부여해야한다. 패킷에 ID를 추가하므로써 원하는 다이나믹셀만을 메인컨트롤러에서 제어할 수 있다. 다이나믹셀은 공장출하시 초기 ID는 1번으로 설정되어있고 ID는 다이나믹셀 위자드에서 변경 가능하다.

Length: Length는 패킷의 길이를 뜻하며 Param의 개수와 instruction, CRC 바이트 개수를 의미한다. 쉽게 말해 Length = Param 개수 + 3 이며 2바이트 크기로 전송한다.

Instruction: 패킷의 용도를 나타내는 데이터로 어떤 형태의 패킷인지를 알 수 있다. 해당 데이터는 1바이트로 전송하며 이번 프로젝트에서 사용한 패킷으로는 다이나믹셀 하나만을 제어하는 write, 여러 다이나믹셀에 대해서 같은 주소에서 같은 길이의 데이터를 하나의 패킷으로 제어하는 sync write가 있고 각각의 값은 0x03, 0x83이다. 그 밖에도 장치의 데이터 값을 읽어오는 read, 여러 다이나믹셀에 대해서 동일한 주소에서 동일한 길이의 데이터를 한 번에 읽어오는 sync read 등이 있다.

Param: Instruction의 보조 데이터로 다이나믹셀에 보내는 실질적인 값으로 goal position, goal velocity, profile velocity등이 있다. 해당 정보들은 로보티즈 e-manual의 컨트롤 테이블을 통해 참고할 수 있다. 컨트롤 테이블은 장치의 상태와 제어를 위한 데이터로 구성된 집합체이다. 장치의 데이터를 읽거나 쓸 때 패킷에 해당 데이터 주소를 지정해주어야 한다. 그러므로써 장치는 param이 무슨 값인지 알게 된다.

104	4	Goal Velocity	목표 속도 값	RW	-
108	4	Profile Acceleration	프로파일 가속도 값	RW	0
112	4	Profile Velocity	프로파일 속도 값	RW	0
116	4	Goal Position	목표 위치 값	RW	-

그림과 같이 컨트롤 테이블에는 주소, 크기, 명칭, 의미 등이 기재되어있다. Write의 기준 '시작 주소의 하위 바이트, 시작 주소의 상위 바이트, 첫 번째 바이트, 두 번째 바이트 ..n번째 바이트'로 구성된다. Instruction마다 param의 구성이 다르므로 e-manual을 잘 보는 것이 중요하다.

CRC: 패킷이 통신 중에 파손되었는지 점검하기 위한 데이터로 하위 바이트와 상위 바이트로 나누어 보내고, 계산 범위는 instruction 패킷의 헤더를 포함하여 CRC필드 이전까지의 데이터가 된다.

예시로 sync write packet의 구성 방법은 다음과 같다.

[sync write]

Header 1	Header 2	Header 3	Reserved	Packet ID	Length 1	Length 2	Instruction	Param	Param	Param	CRC 1	CRC 2
0xFF	0xFF	0xFD	0x00	Packet ID	Len_L	Len_H	Instruction	Param 1	···	Param N	CRC_L	CRC_H

기본적인 패킷의 형태는 위와 같다. Sync write 같은 경우, 패킷 안에 ID와 데이터 주소, 데이터 값을 각각 넣기 때문에 Packet ID는 broadcast id인 0xFD가 된다. 다이나믹셀의 고유 ID가 아닌 broadcast id를 쓰는 이유는 특정 장치로 패킷을 전송하는 것이 아니라 특정 네트워크 전체에 패킷을 전송하는 경우이므로 ip 주소가 broadcast id가 된다.

패킷의 길이인 length는 상위 바이트, 하위 바이트로 나뉘어 들어가고 param 개수 + 3을 통해 알 수 있고 sync write의 instruction은 0x83이다.

Sync write의 param 구성은 다음과 같다.

Instruction Packet	설명
Parameter 1	시작 주소의 하위 바이트
Parameter 2	시작 주소의 상위 바이트
Parameter 3	데이터 길이(X)의 하위 바이트
Parameter 4	데이터 길이(X)의 상위 바이트
Parameter 5	첫번째 장치 ID
Parameter 5+1	첫번째 장치 첫번째 바이트
Parameter 5+2	첫번째 장치 두번째 바이트
…	첫번째 장치 …
Parameter 5+X	첫번째 장치 X번째 바이트
Parameter 6	두번째 장치 ID
Parameter 6+1	두번째 장치 첫번째 바이트
Parameter 6+2	두번째 장치 두번째 바이트
…	두번째 장치 …
Parameter 6+X	두번째 장치 X번째 바이트
…	…

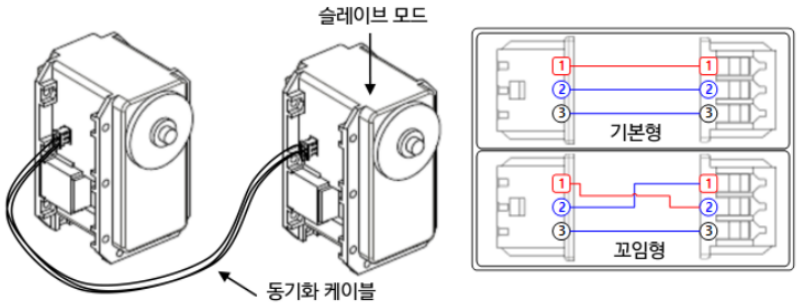
Header부터 crc 이전 필드까지의 크기를 이용하여 crc 값을 구하고 이를 2바이트로 나누면 sync write의 패킷은 완성이 된다.
패킷을 완성했으면 RS-485 통신을 이용해 패킷을 다이나믹셀에 전송하면 해당 주소의 해당 값만큼 작동한다.

```
for (int i = 0; i < 50; i++) {
    while (!LL_USART_IsActiveFlag_TXE(USART3));
    LL_USART_TransmitData8(USART3, txBuf[i]);
}
```

*** Dual mode**

매니퓰레이터 2축에 다이나믹셀 2개를 이용하여 2개의 관절을 하나처럼 구동하기 위해 dual mode를 이용하였다. 이는 다이나믹셀의 Drive mode 중 하나로 마스터 장치와 슬레이브 장치로 구성된다. 마스터 장치의 PWM 신호에 의해서 슬레이브 장치가 제어되므로 슬레이브 장치의 goal position, goal velocity, goal current, goal PWM은 무시된다.

듀얼모드로 설정하려면 마스터 장치와 슬레이브 장치를 동기화 케이블로 연결해주어야한다. 동기화 케이블 중에는 기본형과 꼬임형이 있는데 마스터 장치와 슬레이브 장치의 움직임 방향이 같도록 하고 싶으면 기본형, 반대 방향으로 하고 싶으면 꼬임형으로 연결한다.



동기화 설정은 다이나믹셀 위자드에서 할 수도 있고 코드로 설정해줄 수 있다. Drive mode의 address는 10이고 크기는 1byte, bit는 0x02이므로 다음과 같이 패킷을 구성할 수 있다.

```
void SET_DualMode_MX_Write()
{
   unsigned char txBuf[13];
   unsigned short crc, CRC_L, CRC_H;
   // header 1~4
   txBuf[0] = 0xFF;
   txBuf[1] = 0xFF;
   txBuf[2] = 0xFD;
   txBuf[3] = 0x00;
   // ID
   txBuf[4] = 0x03;
   // length
   txBuf[5] = 0x06;
   txBuf[6] = 0x00;
   // 0x03: write
   txBuf[7] = 0x03;
   // address
   txBuf[8] = 0x0A;//driving mode: 0x0A
   txBuf[9] = 0x00;
   // set dual mode
   txBuf[10] = 0x02;

   // crc
   crc = update_crc(0, txBuf, 11);
   CRC_L = (crc & 0xFF);
   CRC_H = (crc >> 8) & 0xFF;
   txBuf[11] = CRC_L;
   txBuf[12] = CRC_H;
   //transmit to DXL
   for (int i = 0; i < 13; i++) {
      while (!LL_USART_IsActiveFlag_TXE(USART3));
      LL_USART_TransmitData8(USART3, txBuf[i]);
   }
}
```

*** Wheel mode**

그리퍼에 결합된 다이나믹셀은 이번 프로젝트를 진행하면서 위치제어가 아닌 속도 제어를 주로 하기로 했다. 구조상 한 바퀴 이상 돌아가야 완전히 물체를 잡을 수 있었기에 바퀴 모드를 사용하여 속도를 제어하면서 물체를 잡기로 판단한 것이다.

Wheel mode는 operating mode 중 하나로 기본 값은 위치제어모드이므로 위자드나 코드상에서 속도 제어 모드로 변경을 해주어야한다. 주소 11을 의미하는 0x0B와 크기는 1byte이므로 속도 제어모드로 활성화 해주는 0x01로 패킷을 구성할 수 있지만 drive mode에서 normal mode와 operate mode에서 속도 제어모드를 동시에 활성화 하고 싶다면 주소 10인 0x0A와 0x00, 0x01을 연달아 적어준다면 입력한 주소를 시작으로 0x00은 drive mode의 값, 0x01은 operate mode의 값으로 인식한다.

```
void SET_WheelMode_MX_Write()
{
   unsigned char txBuf[14];
   unsigned short crc, CRC_L, CRC_H;
   // header 1~4
   txBuf[0] = 0xFF;
   txBuf[1] = 0xFF;
   txBuf[2] = 0xFD;
   txBuf[3] = 0x00;
   // ID
   txBuf[4] = 0x06;
   // length
   txBuf[5] = 0x07;
   txBuf[6] = 0x00;
   // instr: write
   txBuf[7] = 0x03;
   // address
   txBuf[8] = 0x0A;//driving mode: 0x0A
   txBuf[9] = 0x00;
   // set normal mode
   txBuf[10] = 0x00;
   // operate with goal velocity
   txBuf[11] = 0x01;

   // crc
   crc = update_crc(0, txBuf, 12);
   CRC_L = (crc & 0xFF);
   CRC_H = (crc >> 8) & 0xFF;
   txBuf[12] = CRC_L;
   txBuf[13] = CRC_H;
   //transmit to DXL
   for (int i = 0; i < 14; i++) {
      while (!LL_USART_IsActiveFlag_TXE(USART3));
      LL_USART_TransmitData8(USART3, txBuf[i]);
   }
}
```

바퀴 모드로 세팅을 한 후 속도 제어를 위한 패킷을 이용해 그리퍼를 오므리고 피는 작업을 할 계획이었다. Goal velocity를 이용해서 rpm을 조절할 수 있고 오므리고 피는 작업을 한 번에 하려면 회전 방향을 자유자재로 변경할 수 있어야한다. Drive mode를 보면 기본 값은 Normal mode로 양수값을 넣으면 반시계방향으로 돌고, 음수값을 넣으면 시계방향으로 돈다. Goal velocity의 범위는 0~230이므로 시계방향, 반시계방향 모두 작동하기 위해서는 -230~230의 값이 필요했다. 음수는 2의 보수 변환법을 이용하여 음수를 나타내기 위해 해당 수의 양수 표현을 취한 뒤, 각 비트를 뒤집고 1을 더해주는 것으로 이루어진다. 위자드에서 goal velocity에 -88이라는 값을 넣으면 0xFFFFFFA8로 입력이 된다. 따라서 패킷 안에서 goal velocity의 값은 4바이트로 나뉘어 통신하므로 다음과 같이 패킷을 구성하여 양방향 속도제어를 자유롭게 할 수 있다.

```
void Driving_MX_WheelMode_Write(int velocity) {
    unsigned char txBuf[16];
    unsigned char Vel_1, Vel_2, Vel_3, Vel_4;
    Vel_1 = (unsigned char) (velocity & 0xFF);
    Vel_2 = (unsigned char) ((velocity >> 8) & 0xFF);
    Vel_3 = (unsigned char) ((velocity >> 16) & 0xFF);
    Vel_4 = (unsigned char) ((velocity >> 24) & 0xFF);
    unsigned short crc, CRC_L, CRC_H;
    // header 1~4
    txBuf[0] = 0xFF;
    txBuf[1] = 0xFF;
    txBuf[2] = 0xFD;
    txBuf[3] = 0x00;
    // ID
    txBuf[4] = 0x06;
    // length
    txBuf[5] = 0x09;
    txBuf[6] = 0x00;
    // inst: write
    txBuf[7] = 0x03;
    // address
    txBuf[8] = 0x68; //goal velocity Address: 104 = 0x68
    txBuf[9] = 0x00;
    // goal velocity
    txBuf[10] = Vel_1; //104: goal velocity
    txBuf[11] = Vel_2;
    txBuf[12] = Vel_3;
    txBuf[13] = Vel_4; //4byte
    // crc
    crc = update_crc(0, txBuf, 14);
    CRC_L = (crc & 0xFF);
    CRC_H = (crc >> 8) & 0xFF;
    txBuf[14] = CRC_L;
    txBuf[15] = CRC_H;
    //transmit to DXL
    for (int i = 0; i < 16; i++) {
        while (!LL_USART_IsActiveFlag_TXE(USART3));
        LL_USART_TransmitData8(USART3, txBuf[i]);
    }
}
```

```
if(restoredDXL[4]>=230 && restoredDXL[4] <= 460)
{
    restoredDXL[4] -= 230;
}
else if (restoredDXL[4]>=0 && restoredDXL[4] <= 229)
{
    restoredDXL[4] = restoredDXL[4] - 230;
}
```

*** Torque enable**

다이나믹셀이 구동되기 위해서는 토크가 활성화 되어있어야한다.

2. 3. RAM 영역

주소	크기(Byte)	명칭	의미	접근	기본값
64	1	Torque Enable	토크 On/Off	RW	0

위 사진을 보면 토크의 기본값은 0이므로 어떤 값을 넣어도 작동하지 않을 것이다. 따라서 프로토콜 2.0의 경우, 토크를 수동으로 한 번 활성화를 해주어야한다. 그러기 위해서 다음과 같은 패킷을 구성하였고 goal position 등 매니퓰레이터에 장착된 모든

다이나믹셀을 구동시키기 전에 토크를 먼저 켜주어 다이나믹셀이 돌아갈 수 있도록 하였다.

```
void Torque_Enable_MX_Sync_Write()//index
{
    unsigned char txBuf[26];
    unsigned short crc, CRC_L, CRC_H;
    // header 1~4
    txBuf[0] = 0xFF;
    txBuf[1] = 0xFF;
    txBuf[2] = 0xFD;
    txBuf[3] = 0x00;
    // broadcast id
    txBuf[4] = 0xFE;
    // length
    txBuf[5] = 0x13;
    txBuf[6] = 0x00;
    // inst: sync write instruction packet
    txBuf[7] = 0x83;
    // address
    txBuf[8] = 0x40;
    txBuf[9] = 0x00;
    // length per id
    txBuf[10] = 0x01;
    txBuf[11] = 0x00;
    // id 1 torque enable
    txBuf[12] = 0x01;
    txBuf[13] = 0x01;
    // id 2 torque enable
    txBuf[14] = 0x02;
    txBuf[15] = 0x01;
    // id 3 torque enable
    txBuf[16] = 0x03;
    txBuf[17] = 0x01;
    // id 4 torque enable
    txBuf[18] = 0x04;
    txBuf[19] = 0x01;
    // id 5 torque enable
    txBuf[20] = 0x05;
    txBuf[21] = 0x01;
    // id 6 torque enable
    txBuf[22] = 0x06;
    txBuf[23] = 0x01;
    //crc
    crc = update_crc(0, txBuf, 24);
    CRC_L = (crc & 0xFF);
    CRC_H = (crc >> 8) & 0xFF;
    txBuf[24] = CRC_L;
    txBuf[25] = CRC_H;

    for (int i = 0; i < 26; i++) {
        while (!LL_USART_IsActiveFlag_TXE(USART3));
        LL_USART_TransmitData8(USART3, txBuf[i]);
    }
```

iii. 프로토콜 **1.0** 패킷 구성 방법

프로젝트 진행 중 계획했던 MX-106(2.0), MX-64(2.0), MX-28(2.0)의 사용이 EX-106+, MX-64(2.0), MX-28(2.0)로 바뀌었다. EX-106+의 프로토콜은 1.0이었기에 프로토콜 1.0에 맞는 패킷 또한 필요했다.

전체적인 패킷의 틀은 프로토콜 2.0과 비슷하지만, 프로토콜 2.0에서 패킷이 통신 중에 파손되었는지 점검하기 위해서 CRC를 사용하였다면 프로토콜 1.0에선 Checksum을 계산하여 확인하였다. Instruction Checksum = ~(ID + Length + Instruction + Param1 + …Param N)을 통해 계산할 수 있다. 또한 프로토콜 2.0에서는 토크를 코드상에서 수동으로 활성화를 해주었다면 프로토콜 1.0은 토크가 걸려있지 않은 상태에서 goal position, goal velocity 등 값을 넣어주면 자동으로 토크가 활성화된다. 프로토콜 1.0에 관한 패킷은 다음과 같다.

```
void Driving_EX_Write(unsigned int id, unsigned int position)
{
    unsigned char txBuf[20];
    unsigned char ID = (unsigned char) id;
    unsigned char Pos_L, Pos_H;
    Pos_L = (unsigned char) (position & 0xFF);
    Pos_H = (unsigned char) ((position >> 8) & 0xFF);
    // header 1~2
    txBuf[0] = 0xFF;
    txBuf[1] = 0xFF;
    // ID
    txBuf[2] = ID;
    // length
    txBuf[3] = 0x07;
    // inst: write
    txBuf[4] = 0x03;
    // address 0x1E: moving speed
    txBuf[5] = 0x1E;
    // Goal Position
    txBuf[6] = Pos_L;
    txBuf[7] = Pos_H;
    // moving speed
    txBuf[8] = 0x64;
    txBuf[9] = 0x00;
    // CKSM
    txBuf[10] = 0;
    for(int i=2;i<10;i++)
    {
        txBuf[10] += txBuf[i];
    }
    txBuf[10] = ~txBuf[10];
    //transmit to DXL
    for (int i = 0; i < 11; i++) {
        while (!LL_USART_IsActiveFlag_TXE(USART3));
        LL_USART_TransmitData8(USART3, txBuf[i]);
    }
}
```

iv. 발생했던 문제와 해결 과정

1. 여러 개의 패킷을 보낼 때 패킷의 불완전한 전송으로 인한 delay
2. 통신 문제에 있어서 baud rate 수정
3. 통신 문제에 있어서 예외처리

패킷을 구성하고 매니퓰레이터의 조립이 완성되어 구동을 해보는 과정에서 예상하지 못한 문제들이 발생하였다.

1. 여러 개의 패킷을 보내는 경우, 패킷의 불완전한 전송
   패킷에서 주소를 입력하면 그 주소부터 차례대로 crc 필드 이전까지 데이터를 순차적으로 인식한다. 따라서 주소 간의 간격이 크면 그 사이를 모두 일정한 값으로 채워야하고 패킷의 크기는 커진다. 그것보다 짧은 패킷으로 나누어 보내는 것이 효율적이다. 여러 개의 패킷을 연달아 보내면 이전 패킷이 다 전송되기 전에 다음 패킷이 전송된다. 이런 현상을 해결하기 위해 패킷을 보내고 일정 딜레이를 주어 패킷이 완전하게 전송되도록 하였다.

```
//torque enable, set wheel mode, set dual mode
if (torque_flag == 0) {
    Torque_Enable_MX_Sync_Write();
    LL_mDelay(5);
    SET_WheelMode_MX_Write();
    LL_mDelay(5);
    SET_DualMode_MX_Write();
    LL_mDelay(5);
    torque_flag = 1;
}
```

2. 통신 Baud rate를 57600으로 설정하였으나 연결된 다이나믹셀이 많아 값의 갱신 속도가 현저히 느려지는 현상이 발생하였다. 이를 해결하기 위해 Baud rate를 1000000로 변경해주었더니 해당 현상이 완화되었다.
3. Pc에서 stm으로 송신을 하고 stm에서 수신 받는 그 사이 일정한 텀이 존재했다. 다이나믹셀의 goal position이 2350, 2480, 3070 등등의 값이 입력되어야하는데 통신 지연 문제로 모두 0의 값이 입력되면 구조상 다이나믹셀의 하드웨어에 손상이 간다. 그래서 일정 텀 동안은 다음과 같이 예외처리를 통하여 하드웨어의 손상을 방지하였다.

```
if(DXL1 == 0 && DXL2 == 0 && DXL3 == 0 && DXL4 == 0){
    Driving_MX_Sync_Write(2350, 2480, 3070, 2478);
}
else{
    Driving_MX_Sync_Write(DXL1, DXL2, DXL3, DXL4);
}
LL_mDelay(5);
```

b. 팬틸트**(pan tilt)**

사용하게 된 경위: 차선을 보고 주행하다 보면 표지판을 제대로 인식을 할 수 없었다. 팬틸트를 이용하여 카메라가 차선을 보고 주행하다가 표지판의 일정 크기 이상 인식되면 표지판을 화면 중앙으로 따라가고 다시 차선을 보도록 하였다.

[초기값 세팅**]**

```
#define DEVICENAME "/dev/ttyUSB0"
#define PROTOCOL_VERSION 2.0
#define BAUDRATE 1000000 //여기
#define Hz 80
#define NUMBER_OF_MOTORS 2//

#define OPERATING_MODE 1 //1 = position control, 2 = speed control
```

통신 모듈의 USB 이름과, 장치의 프로토콜 버전, 보레이트, 모터 개수, operate mode 설정을 한다.

```
double pan_input = 0;
double pan_target = 240;
double tilt_input = 0;
double tilt_target = 135;//윈도우의 중앙값
```

Input 변수는 퍼블리시로 받을 표지판의 현 좌표이고 pan_target은 카메라 UI의 x좌표, tilt_target은 카메라 UI의 y좌표로 카메라가 보고자 하는 좌표가 된다.

[절차]

- **Init pub**

카메라가 주행 시작할 때 올바른 각도로 차선을 보기 위해서 init 자세를 취하도록 하는 initCallback 함수를 실행한다. “Init_pub” 라는 토픽으로 불형 value를 받아서 true라면 이닛 자세를 취하도록 한다.

```
void initCallback(const std_msgs::Bool::ConstPtr &msg)
{
    if (msg->data == true)
    {
        goal_Position[0] = 2.5;
        goal_Position[1] = 0;

        controlFSM = INIT_WRITE;
    }
}
```

- **Target**

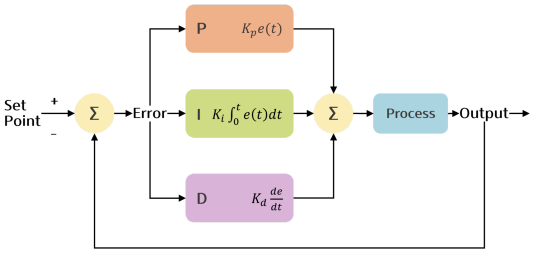
주행하는 도중 초록색 표지판이 일정 비율 이상 커지면 “target” 라는 토픽으로 카메라가 init 자세일 때 초록 표지판의 중앙점을 publish하고 pantiltCallback 함수가 실행된다. 해당 함수는 lock guard를 사용하여 global_mutex에 락을 걸어 하나의 스레드만이 접근 가능하도록 한다. 퍼블리셔를 통해 받은 현 표지판의 좌표가 input 변수에 저장되고 PID_Control 함수를 호출하여 목표값, 현재값, 출력값을 인자로 받아 pid 제어를 수행한다. Goal position에 pid 제어를 통해 나온 output 값을 더해 나가면서 target에 가까워진다.

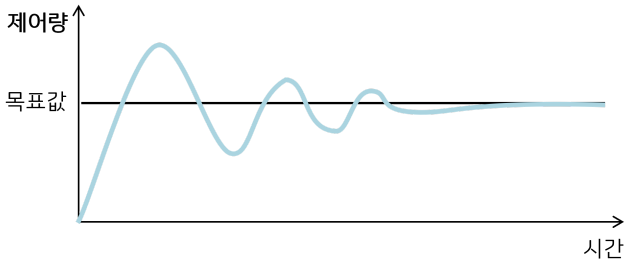
```
void pantiltCallback(const geometry_msgs::Point::ConstPtr &msg)
{
    lock_guard<mutex> lg(global_mutex);

    pan_input = msg->x;
    tilt_input = msg->y;

    PID_Control(&pan, pan_target, pan_input);
    PID_Control(&tilt, tilt_target, tilt_input);

    goal_Position[0] += pan.output;
    goal_Position[1] += tilt.output;

    cout << "pan: " << goal_Position[0] << endl;
    cout << "tilt: " << goal_Position[1] << endl
         << endl;
}
```

[PID]

PID란: 자동제어 방식 중 하나로 p는 비례를 뜻하는 proportional, 적분을 뜻하는 integral, 미분을 뜻하는 differential의 약자로 제어에 유연하다.

피드백 제어란 입력 값을 한 프로세스에 적용을 하고 그 결과 값을 다시 입력값으로 하여 목표값에 도달하는 방식이다. 결과값을 측정한 다음 목표값과 비교하여 오차를 구한 다음, 제어량을 오차값을 이용하여 계산하는 방식이다.





제어량 계산 식은 $$K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de}{dt}$$ (비례항, 적분항, 미분항)이다. 비례항은 편차에 비례하여 현 상태에서 편차의 크기에 비례하여 적용한다. 적분항은 오차값의 적분값에 비례하여 오차를 없앤다. 미분항은 오차값의 미분에 비례하여 목표값보다 오차가 커지는 현상인 오버슈트를 줄인다.

- PID_Control

입력값과 목표값, 목표값에서 현재값을 뺀 오차값을 이용하여 제어량을 구현한다. sumE는 오차인 E를 더하여 적분을 의미하고, 오차값(E)에서 이전 오차값(E_old)을 뺀 diffE는 미분값을 의미한다. 또한 정상범위 내에서 작동하기 위해서 pid의 결과값인 output의 범위를 제한해주었고 오버슈팅 현상을 방지하기 위해서 오차값도 범위 제한을 걸어주었다. 해당 함수 내용은 아래와 같다.

```
void PID_Control(PID *dst, double target, double input)
{
    dst->input = input;
    dst->target = target;

    dst->E = dst->target - dst->input;
    dst->sumE += dst->E;
    dst->diffE = dst->E - dst->E_old;

    if (dst->sumE > dst->sumELimit)
    {
        dst->sumE = dst->sumELimit;
    }
    else if (dst->sumE < -dst->sumELimit)
    {
        dst->sumE = -dst->sumELimit;
    }

    dst->output = (dst->kP * dst->E + dst->kI * dst->sumE + dst->kD * dst->diffE) / division;

    dst->E_old = dst->E;

    if (dst->output > dst->outputLimit)
    {
        dst->output = dst->outputLimit;
    }
    else if (dst->output < -dst->outputLimit)
    {
        dst->output = -dst->outputLimit;
    }
}
```

[PC to DXL]

1. 다이나믹셀 포트 핸들러, 패킷 핸들러, group BulkWrite, group BulkHead를 초기화한다.

2. U2D2 연결과 장착된 모터 개수에 대해서 토크를 활성화한다.

3. 팬 및 틸트 축에 대해서 PID 계수와 함께 pid 초기화한다.

```
serialPortCheck(portHandler);
motorEnable(NUMBER_OF_MOTORS);
motorTorqueOn(portHandler, packetHandler);
storageMovingStatus(&groupSyncRead);

PID_Init(&pan, 25, 0.001, 0, 150, 80);
PID_Init(&tilt, 25, 0.001, 0, 150, 80);
```

4. 포트가 연결됨을 확인하고 나면 operating mode인 위치 제어 모드가 활성화 된다.

```
serialPortCheck(portHandler);

if (serial_result == true)
{
    if (OPERATING_MODE == 1)
    {
        switch (controlFSM)
        {
        case INIT_WRITE:
        {
            acc = 10;
            vel = 60;

            for (int i = 0; i < NUMBER_OF_MOTORS; i++)
            {
                pulse_desired[i] = (init_Position[i] + 180.0) * MX_DEG2PULSE;
            }
```

```
81  void serialPortCheck(dynamixel::PortHandler *portHandler)
82  {
83      if (portHandler->openPort())
84      {
85          if (portHandler->setBaudRate(BAUDRATE))
86          {
87              serial_result = true;
88          }
89          else
90          {
91              serial_result = false;
92          }
93      }
94      else
95      {
96          serial_result = false;
97      }
98  }
```

5. 모터 상태를 말하는 controlFSM 순차적으로 상태를 전환한다.

Init write: init position을 다이나믹셀의 값에 맞게 연결된 모든 장치를 write 한다.

```
case INIT_WRITE:
{
    acc = 10;
    vel = 60;

    for (int i = 0; i < NUMBER_OF_MOTORS; i++)
    {
        pulse_desired[i] = (init_Position[i] + 180.0) * MX_DEG2PULSE;
    }

    setPosition(&groupBulkWrite, packetHandler, pulse_desired);
    groupBulkWrite.clearParam();

    controlFSM = MOVING_STATUS_READ;

    break;
}
```

Moving status read: 움직임의 상태를 읽고 position write로 상태를 전환한다.

```
case MOVING_STATUS_READ:
{
    in_position_cnt = 0;

    readMovingStatus(&groupSyncRead, packetHandler);
    checkData(&groupSyncRead);
    getMovingStatus(&groupSyncRead);

    for (int i = 0; i < NUMBER_OF_MOTORS; i++)
    {
        if (moving_status[i] == 0x01)
        {
            in_position_cnt++;
        }
    }

    if (in_position_cnt == NUMBER_OF_MOTORS)
    {
        acc = 0;
        vel = 0;

        controlFSM = POSITION_WRITE;
    }
    else
    {
        controlFSM = MOVING_STATUS_READ;
    }

    break;
}
```

Position write: 각도를 다이나믹셀의 goal position에 맞게 변환한 후 write 진행한다.

```
case POSITION_WRITE:
{
    global_mutex.lock();
    for (int i = 0; i < NUMBER_OF_MOTORS; i++)
    {
        pulse_desired[i] = (goal_Position[i] + 180.0) * MX_DEG2PULSE;
    }
    global_mutex.unlock();

    setPosition(&groupBulkWrite, packetHandler, pulse_desired);
    groupBulkWrite.clearParam();

    controlFSM = POSITION_WRITE;

    break;
}
default:
    break;
}
```