Velodyne 3D Lidar(VLP-16) Driver

함수 구조

```
File: velodyne_driver
└── Namespace: velodyne
```

```
    ├── Type Function
    │   └── Struct: point_cloud
    └── Class: driver
        ├── Public
        │   ├── Constructor: driver
        │   └── Destructor: ~driver
        └── Private
            ├── Type Function
            │   ├── tuple: data_tuple
            │   ├── Struct: point
            │   ├── Struct: data_block
            │   └── Struct: data_packet
            └── Function
                ├── calculate_xyz
                └── data_processing
```

헤더파일

```
// 기본 헤더
#include <iostream>
#include <functional>
#include <ctime>
#include <cassert>
#define _USE_MATH_DEFINES
#include <math.h>

// Socket 연결 헤더
#ifdef _WIN64
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")
#else
#include <sys/shm.h>
#include <sys/ipc.h>
// Linux socket 관련 헤더
#include <sys/types.h>
#include <arpa/inet.h>
#include <sys/socket.h>
// Linux processer 실행 헤더
#include <unistd.h>
#endif // _WIN64
#include <fcntl.h>

// 멀티 쓰레드 접근 헤더
#include <thread>
#include <mutex>

// Queue 통신
#include <queue>

// Tuple 데이터 타입
#include <tuple>
```

사용 방법

1. **드라이버 인스턴스 생성**

```
#include "include/velodyne_driver.hpp"

namespace vd = velodyne;

int main(int argc, char *argv[])
{
  vd::driver driver = vd::driver();
```

2. **패키지 추출**

인스턴스를 통해서 접근 가능하다. 패키지는 실시간으로 업데이트 된다.

```
  for (int i = 0; driver.point_clouds[i].x != NULL; i++)
  {
      std::cout << "x: " << driver.point_clouds[i].x << " | y: " << driver.point_cloud
s[i].y << " | z: " << driver.point_clouds[i].z << std::endl;
  }
```

3. **드라이버 종료**

메인 프로세스가 끝나기 전에 소멸자를 호출해서 종료시켜야 한다.

```
  driver.~driver();

  return 0
}
```

VLP-16가 수신하는 데이터의 구조

용어

1. **Firing Sequence**

   패킷 데이터 수집 시작 시간을 기준으로 VLP-16이 레이저를 발사한 횟수다.

2. **Laser Channel**

   라이다의 위 또는 아래로 최대한 볼 수 있는 각도(부앙각)를 몇 등분 했는지 나타낸 지표.

VLP-16은 16채널이다.

3. **Data Point (3 Bytes)**

> **변위(Distance)**: 2 bytes
> **반사율(Reflectivity)**: 1 byte

반사율을 이용하진 않으므로 해당 코드에서는 변위만 추출한다.

4. **Data Block(100 bytes)**

> **플래그(Flag)**: 2 bytes
> **방위각(Azimuth)**: 2 bytes
> **Data Point 32개**: 96 bytes

플래그를 통해서 해당 블럭의 데이터가 정상값을 초과했는지 정상값보다 적은지 확인할 수 있다.

5. **Data Packet**

> **해더(Header)**: 42 bytes ⇒ 버퍼에서 수집 X
> **Data Block 12개**: 1200 bytes
> **timestamp**: 4 bytes
> **factory field**: 2 bytes

factory field를 통해 Lidar의 return mode를 확인할 수 있다.

6. **Return Modes**

> **Strongest (Default)**: 가장 강한 값을 반환
> **Last**: 가장 마지막에 감지된 값을 반환
> **Dual**: 위의 두 모드의 값을 모두 반환

데이터 구조

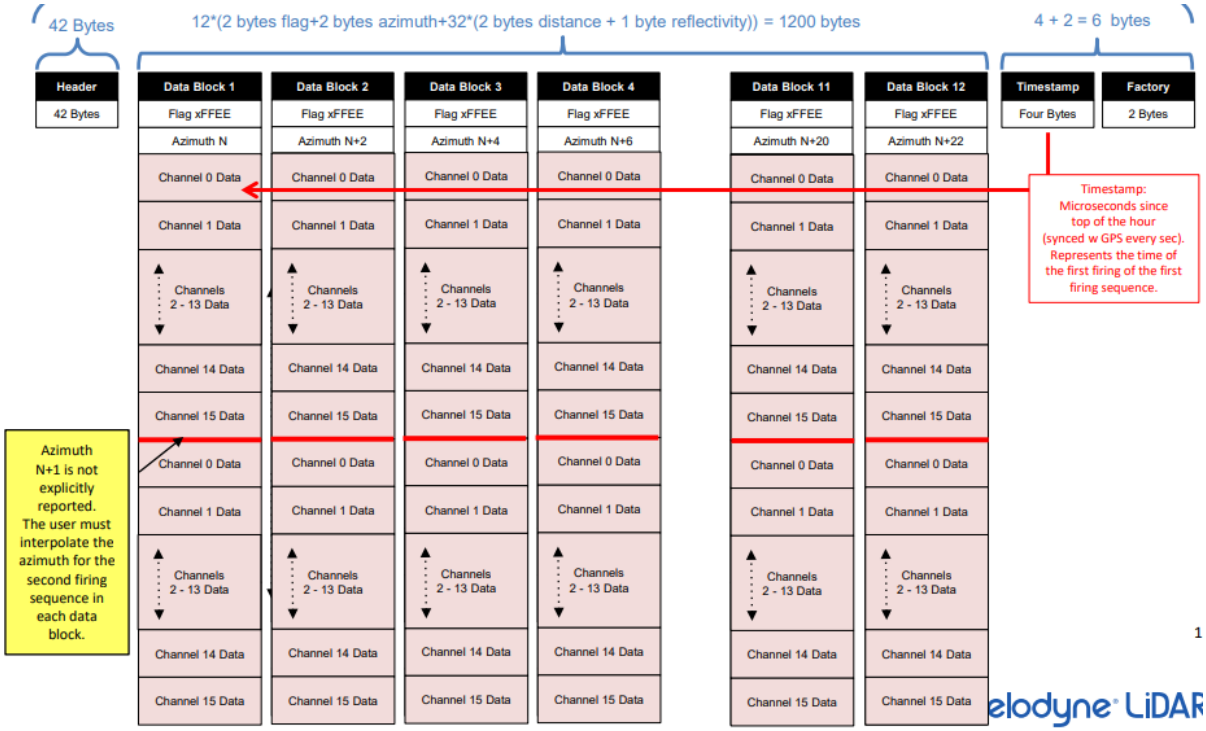
부앙각

데이터 블럭에 저장되어 있는 데이터 포인트는 16개씩 묶을 수 있다. 16개씩 묶은 데이터들은 가장 앞부터 순서대로 0~15채널의 값을 가지며, 각 채널은 아래의 표에 대응된다.

VLP-16 Channel #	VLP-16 Vert Angle (°)
0	-15°
1	1°
2	-13°
3	-3°
4	-11°
5	5°
6	-9°
7	7°
8	-7°
9	9°
10	-5°
11	11°
12	-3°
13	13°
14	-1°
15	15°

채널에 대응하는 부앙각

패킷 구조(Strongest & Last)



패킷은 12개의 블럭으로 구성되고, 각 블럭은 데이터 32개로 구성된다. 블럭마다 고유의 방위각을 가지고 있으며, 가장 처음 데이터 16개는 해당 방위각과 일치하며, 뒷쪽 16개는 다음 블럭의 값을 이용하여 보간해서 구해야한다.

해당 코드에서 패킷을 받았을 때, **(1)**패킷을 블럭으로 나누고, **(2-1)**블럭을 2등분 한뒤, **(3)**그 내부의 데이터 16개의 데이터에서 부앙각과 변위를 추출한다. **(2-2)**블럭을 나누는 과정에서 방위각을 추출하고, **(4)**추출한 3종류의 데이터로 Lidar를 원점으로한 상대 좌표를 실시간으로 연산한다.

참조

> https://s3-us-west-2.amazonaws.com/secure.notion-static.com/0d8445fd-b639-4d57-9f31-527f8d6b2bd4/63-9276-Rev-C-VLP-16-Application-Note-Packet-Structure-Timing-Definition.pdf

---

구조체(Structure Type)

point_cloud

```
typedef struct
  {
    double x;
    double y;
    double z;
  } point_cloud;
```

x, y, z 값을 가진 데이터 묶음

data_block

```
typedef struct
    {
      uint16_t flag;          // 데이터 플래그
      uint16_t azimuth;       // 방위각
      uint8_t points[POINT_SIZE];   // 거리 데이터 및 반사율
    } data_block;
```

VLP-16의 Data Block의 형태를 구조체로 구현

data_packet

```
typedef struct
    {
      data_block blocks[BLOCK_SIZE];  // 포인트 블럭
      uint32_t timestamp;       // 송신한 시간
      uint16_t factory_field;     // 모델명, 실질적으로 사용 X
    } data_packet;
```

VLP-16의 Data Packet을 구조체로 구현

공유체(Union Type)

point

```
typedef union
    {
      uint16_t distance;    // 물체까지 변위
      uint8_t bytes[2];   // 바이트별 숫자 크기 (16진법)
    } point;
```

Data Block의 points 내에 흩어져 있는 변위 데이터 2개를 bytes에 저장하고 distance를 호출하면 두 데이터가 합쳐진 형태의 데이터를 얻을 수 있다.

함수(Fucntion)

생성자

▼ 소스 코드

```
driver::driver()
    {
#ifdef _WIN64
      /***************** 윈도우일 경우 라이브러리 초기화 ***************/

      WSADATA wsa_data;
      if (WSAStartup(MAKEWORD(2, 2), &wsa_data) != 0) {
        cerr << "Failed to initialize Winsock." << endl;
        return;
      }
      cout << "Succese : Initalisze Winsock package" << endl;
#endif  //_WIN64


      /***************** UDP 소켓 생성 ***************/

      socket_ = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
#ifdef _WIN64
      if (socket_ == INVALID_SOCKET)
      {
        cerr << "Failed to create socket." << endl;
        closesocket(socket_);
        WSACleanup();

        return;
      }
#elif __linux__
      if (socket_ == -1)
      {
        std::cerr << "Failed to create socket." << std::endl;
        close(socket_);

        return;
      }
#endif // _WIN64
      cout << "Succese : Create socket" << endl;


      /***************** 소켓 로컬 주소 구성 ***************/

      sockaddr_in local_address;
      local_address.sin_family = AF_INET;       // IPv4로 패밀리 설정
```

```
        local_address.sin_addr.s_addr = INADDR_ANY;   // 로컬 주소 자동 탐색
        local_address.sin_port = htons(PORT);     // 센서 포트 번호

        if (bind(socket_, (sockaddr*)&local_address, sizeof(local_address)) == SOCKE
    T_ERROR) {
            cerr << "Failed to bind socket." << endl;
            closesocket(socket_);
            WSACleanup();

            return;
        }

        cout << "Succese : Bind socket" << endl;

        std::thread receive(&driver::data_receive, this);

        receive.detach();   // 데이터 송신 및 처리 시작

        return;
    }
```

분석

```
WSADATA wsa_data;
if (WSAStartup(MAKEWORD(2, 2), &wsa_data) != 0) {
    cerr << "Failed to initialize Winsock." << endl;
    return;
}
cout << "Succese : Initalisze Winsock package" << endl;
```

Windows에서 소켓을 구현하는 경우 소켓 구현과 관련된 정보를 `WSADATA` 에 저장해야 한다.

첫번째 인수에는 버전 정보가 기입되어야 하며, 메크로 `MAKEWORD` 를 이용한다. (Ex. `MAKEWORD( 2 , 2 )` ⇒ 소켓 2.2버전 사용)

두번째 인수에는 데이터를 저장할 `WSADATA` 타입의 변수가 필요하다.

정상적으로 정보를 받아왔다면 **0을 반환**하며 **에러**가 생겼다면 **0이 아닌 숫자**를 반환한다.

```
socket_ = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

if (socket_ == INVALID_SOCKET)
{
    cerr << "Failed to create socket." << endl;
    closesocket(socket_);
    WSACleanup();

    return;
}
```

```
cout << "Succese : Create socket" << endl;
```

`socket` 함수를 통해 소켓을 생성한다. 생성에 실패한 경우 Socket과 패키지를 닫고 함수를 종료한다. 인수들의 의미는 아래와 같다

> **AF_INET**: IPv4를 이용
> **SOCK_DGRAM**: 데이터 손실이 있어도 최대한 빠르게 정보를 수신
> I**PPROTO_UDP**: 통신에 UDP 통신을 이용

```
sockaddr_in local_address;
local_address.sin_family = AF_INET;
local_address.sin_addr.s_addr = INADDR_ANY;
local_address.sin_port = htons(PORT);

if (bind(socket_, (sockaddr*)&local_address, sizeof(local_address)) == SOCKET_ERROR)
{
    cerr << "Failed to bind socket." << endl;
    closesocket(socket_);
    WSACleanup();

    return;
}

cout << "Succese : Bind socket" << endl;
```

소켓통신에 필요한 로컬 주소를 구성하고 소켓과 연결한다. VLP-16의 포트는 **2368**이며 헤더 파일에 상수값으로 저장되어 있다.

> **AF_INET**: IPv4를 이용
> **INADDR_ANY**: 연결 주소 자동 탐색

```
std::thread receive(&driver::data_processing, this);

receive.detach();
```

멀티쓰레드 작동 방식

소멸자

소스 코드

```
driver::~driver()
{
  // 소켓 종료
#ifdef _WIN64
  WSACleanup();
  closesocket(socket_);
#elif __linux__
  close(socket_);
#endif // _WIN64

  return;
}
```

소켓과 관련된 패키지를 모두 닫고 인스턴스를 종료한다.

---

calculate_xyz

소스 코드

```
point_cloud driver::calculate_xyz(int azimuth, double distance, int angle)
{
  point_cloud point;
```

```
  double omega = angle * M_PI / 180.;
  double alpha = (double)azimuth * ROTATION_RESOLUTION * M_PI / 180.;

  point.x = DISTANCE_RESOLUTION * distance * cos(omega) * sin(alpha);
  point.y = DISTANCE_RESOLUTION * distance * cos(omega) * cos(alpha);
  point.z = DISTANCE_RESOLUTION * distance * sin(omega);

  return point;
}
```

ㅇ방위각, 거리, 부앙각(구면좌표계)을 입력받고 입력받은 데이터를 바탕으로 x, y, z 데이터(직교좌표계) 산출한다.

---

data_processing

▼ 소스 코드

```
void driver::data_processing()
  {
    /***************** 변수 선언 ****************/
    int sequence;       // 시퀀스 반복 횟수
    int azimuth;        // 방위각
    int azimuth_diff;     // 블록당 방위각 변화
    int last_azimuth_diff;    // 마지막으로 측정된 방위각 변화
    int point_cloud_idx = 0;  // Point Cloud 저장 주소

    double distance;      // 물체까지의 변위
    double time;        // 데이터가 추출된 시간
    double start_time = 0;    // 패키징 시작 시간

    char buffer[BUFFER_SIZE]; //데이터 버퍼

    data_packet* packet;    // 데이터 패킷
    data_block* block;      // 데이터 블록

    point point;        // 지점의 데이터 공용체

    uint32_t timestamp;     // 패킷을 전송 받은 시간

    point_cloud temp;     // 데이터 임시값


    /***************** 드라이버 코드 ****************/

    sockaddr_in remote_address;             // 연결된 주소 정보
    int remote_address_size = sizeof(remote_address); //주소 크기

    while (true)
    {
      /***************** 데이터 수신 ****************/
      int receive_data_size = recvfrom(socket_, buffer, BUFFER_SIZE, 0, (sockaddr
*)&remote_address, &remote_address_size);
```

```
      /***************** 데이터 처리(버퍼 --> 패킷) ****************/

      azimuth_diff = 0;   // 방위각 변화율 초기화
      last_azimuth_diff = 0;  // 마지막으로 측정된 바위각 변화율 초기화

      if (receive_data_size == 0) continue; // 데이터가 정상적으로 들어올 때까지 수신 반복

      // 받아온 데이터를 패킷으로 이동
      packet = (data_packet*)buffer;
      assert(packet->factory_field == 0x2237);  // 예외 : 라이다의 값이 지정한 모드와 다른
경우

      timestamp = packet->timestamp;        // 데이터 수신 시간 저장


      /***************** 데이터 처리(패킷 --> 블록) ****************/

      for (int block_idx = 0; block_idx < BLOCK_SIZE; block_idx++)
      {
        // 패킷에서 블럭 추출
        block = &packet->blocks[block_idx];
        assert(block->flag == UPPER_BANK);    // 예외 : 블럭이 상한선을 넘은 경우

        azimuth = (int)block->azimuth;      // 해당 블럭의 데이터가 가지고 있는 방위각 저장

        // 다음 블럭의 방위각을 이용하여 방위각 사이 각도 계산
        if (block_idx + 1 < BLOCK_SIZE)
        {
          azimuth_diff = (ROTATION_MAX + packet->blocks[block_idx + 1].azimuth - a
zimuth) % ROTATION_MAX;   // 방위각 간격 계산
          last_azimuth_diff = azimuth_diff; // 최신 데이터 업데이트
        }
        else
          azimuth_diff = last_azimuth_diff; // 새로 산출이 불가능한 경우 가장 최신 데이터 사
용

        // 블럭의 데이터를 Firing마다 분석
        for (int firing = 0, k = 0; firing < BLOCK_PER_FIRING; firing++)
        {
          sequence = BLOCK_PER_FIRING * block_idx + firing; // Firing 횟수 산출


          /***************** 데이터 처리(블록 --> 채널) ****************/

          for (int chanel = 0; chanel < CHANELS; chanel++, k += LASER_DATA_BYTES)
          {
            point.bytes[0] = block->points[k];    // 위치 데이터의 첫번째 숫자 저장 (HEX)
            point.bytes[1] = block->points[k + 1];  // 위치 데이터의 두번째 숫자 저장 (HE
X)
            distance = point.distance;        // 위에서 입력받은 데이터를 한번에 저장

            if (distance == 0) continue;  // 거리가 0인 경우(= 사각지대 or 범위 밖) 다음
 채널로 통과

            time = (timestamp + (sequence * 55.296 + chanel * 2.304)) / 1000000.0;
// 통신 지연 시간 산출
```

```
            // 한바퀴 돌때마다 패키징 다시 시작
            if (time - start_time >= FRAME_CUT)
            {
              //cout << point_idx << endl;
              point_cloud_idx = 0;  // 패키징 주소 초기화
              start_time = time;    // 패키징 시작 시간 초기화
            }


            /***************** 데이터 처리(데이터(구면) --> 데이터(직교)) ***************
*/

            temp = calculate_xyz(azimuth, distance, LASER_ANGLE[chanel]); // 직교좌
표계 연산

            // 레이더 사용에 필요없는 부분 생략
            if (temp.y >= 0)
            {
              point_clouds[point_cloud_idx] = temp;
              point_cloud_idx++;
            }
          }
        }

        // 다음 방위각 연산
        azimuth += azimuth_diff / BLOCK_PER_FIRING;
        azimuth %= ROTATION_MAX;
      }
    }

    return;
  }
```

분석

```
int receive_data_size = recvfrom(socket_, buffer, BUFFER_SIZE, 0, (sockaddr*)&remote_a
ddress, &remote_address_size);
```

드라이버에서 생성한 소켓을 통해 라이다에서 데이터를 수신한다. **해당 코드는 입력이 들어오기 전까지 대기**하기 때문에 이 부분에서 코드가 멈춰있다면 라이다 연결을 확인하자.

```
if (receive_data_size == 0) continue;

packet = (data_packet*)buffer;
assert(packet->factory_field == 0x2237);

timestamp = packet->timestamp;
```

```
azimuth_diff = 0;
last_azimuth_diff = 0;
```

데이터가 정상적으로 수신될때까지 대기하고, 정상적으로 수신됐다면 버퍼의 데이터를 패킷 단위로 변환한다. 변환한 패킷에서 전송된 시간을 추출하고, 방위각 정보를 초기화한다.

```
block = &packet->blocks[block_idx];
assert(block->flag == UPPER_BANK);

azimuth = (int)block->azimuth;

if (block_idx + 1 < BLOCK_SIZE)
{
  azimuth_diff = (ROTATION_MAX + packet->blocks[block_idx + 1].azimuth - azimuth) % RO
TATION_MAX;
  last_azimuth_diff = azimuth_diff;
}
else
  azimuth_diff = last_azimuth_diff;
```

패킷에서 순차적으로 블럭을 뽑고 블럭에 포함되어 있는 방위각 정보를 이용하여 각 Firing 사이의 방위각 변화를 계산한다.

```
sequence = BLOCK_PER_FIRING * block_idx + firing;
```

각 포인트를 센서가 인식한 시간을 정확히 파악하기 위해 패킷 전송시간 기준으로 시퀀스 횟수를 계산한다. 이 때 firing 변수는 한 블럭에서 몇 세트(16개 기준)의 데이터가 기존에 인식 됐는지 표시한다. VLP-16은 블럭당 두 세트의 데이터가 있으므로 최대 2까지 값을 가지게 된다.

```
point.bytes[0] = block->points[k];
point.bytes[1] = block->points[k + 1];

distance = point.distance;

if (distance == 0) continue;

time = (timestamp + (sequence * 55.296 + chanel * 2.304)) / 1000000.0;
if (time - start_time >= FRAME_CUT)
{
  point_cloud_idx = 0;
  start_time = time;
}
```

위의 시퀀스 횟수를 바탕으로 계산한 시간을 계산하고, 패키징 시작 시간기준으로 한바퀴 도는 시간(FRAME_CUT = 0.1s)에 도달하면 패키징 시간을 초기화 하고, 반환값을 처음 값부터 다시 업데이트한다.

블럭에서 부분적으로 나눠져있는 변위 데이터를 합친다.

```
temp = calculate_xyz(azimuth, distance, LASER_ANGLE[chanel]);

if (temp.y >= 0)
{
  point_clouds[point_cloud_idx] = temp;
  point_cloud_idx++;
}
```

ㅇ위에서 계산한 값들을 이용하여, 데이터를 직교좌표계 형태로 계산한다. 라이다 기준 앞쪽의 데이터만 사용할 예정이므로 y값이 0 이상인 부분만 저장한다.