Intel Sensor Fusion Development Kit

Design and Development Reference

April 2023

Document Number: 771026-1.0

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or visit www.intel.com/design/literature.htm.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No product or component can be absolutely secure. Check with your system manufacturer or retailer or learn more at intel.com.

No product or component can be absolutely secure.

Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Contents

Figures

Tables

Revision History

Date	Revision	Description
April, 2023	1.0	Initial release

§

1.0 Introduction

This documentation contains information about the system environment in which the Intel® Sensor Fusion Development Kit software runs, and provides an overview of various functional interfaces. Please refer to the *Intel Sensor Fusion Development Kit User Guide* (RDC #774272) for more support of software installation, defect description, and constraints.

1.1 Overview

This project aims to provide a leading platform for the fusion of industrial sensors in mobile robots. This software was developed for the Linux* operating system (kernel 5.10), and thus relies upon the use of Linux drivers for sensor fusion, ROS2 software packages to process Field Programmable Gate Arrays (FPGA) timestamp and synchronization, and the development of applications to demonstrate the performance of sensor fusion. During project development, test plans and test cases were designed based on functional interfaces. After completing the programming work, the software test report and recorded troubleshooting steps were delivered.

1.2 Terminology

Table 1. Terminology

Term	Description
FPGA	Field Programmable Gate Arrays
ROS	Robot Operating System
V4L2	Video device driver on Linux systems
I2C	Inter-Integrated Circuit
MSOP	Main data Stream Output Protocol
UCWP	User Configuration Write Protocol
DIFOP	Device Information Output Protocol
DMA	Direct Memory Access
DBSCAN	Clustering Algorithm
PCL	Point Cloud Library

Term	Description
UI	User Interface

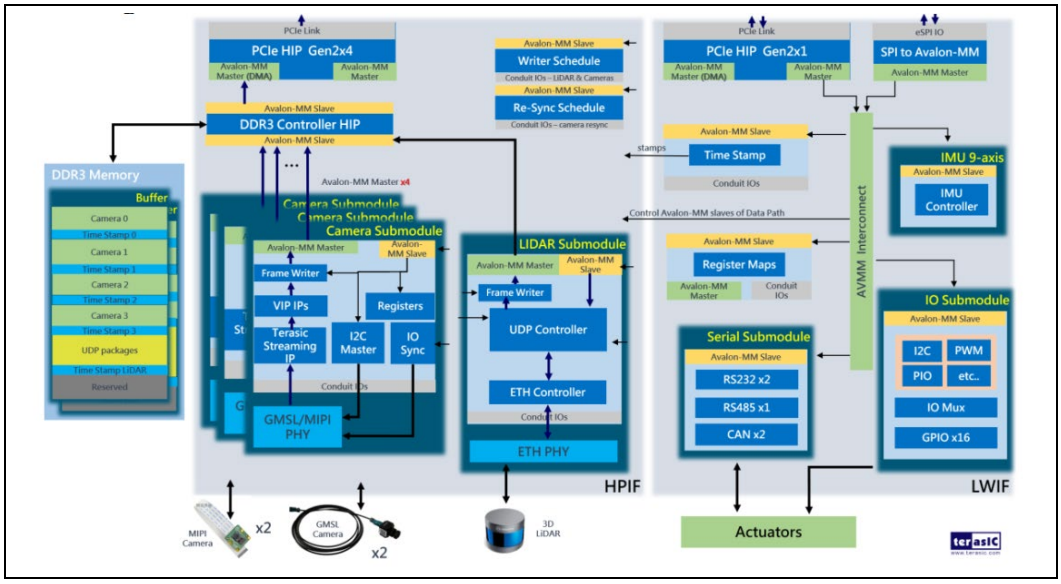
§

2.0 Software Design Description

2.1 Overall Design

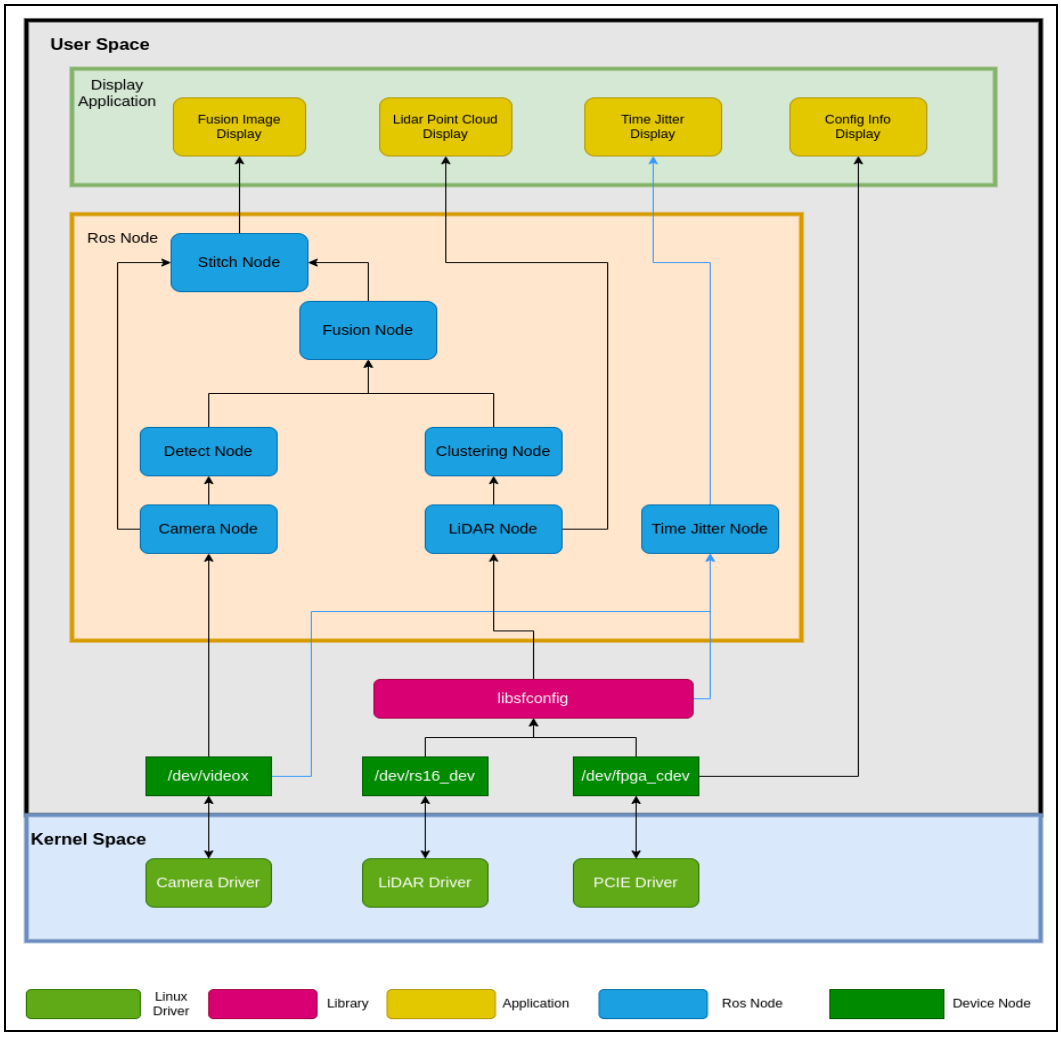
In accordance with hardware functionality requirements (as shown in Figure 1), the first priority for software development is driver development for a LiDAR device, the camera device, and FPGA time synchronization processing functionality of the Linux system. The utility layer will then implement data processing via Robot Operating System (ROS) nodes and display the result in the User Interface (UI).

Figure 1. Hardware Functionality



From the system perspective, the software design can be separated into three layers from the bottom to the top: the driver layer, the middle layer, and the application layer. Data communication between layers is demonstrated in Figure 2.

Figure 2. Software System Data Flow Diagram



Based on hardware functionality, three device driver nodes are created for user space applications to handle hardware operations. `/dev/fpga_cdev` is used for FPGA device configuration; `/dev/rs16_dev` is used for LiDAR sensor configuration and sensor data retrieval; `/dev/videox (x = 0~3)` is used for camera sensor configuration and video data retrieval. A user space library is provided to encapsulate the device driver interface for application access. A display application is created to display device information (Config Information Display), sensor data latency graph (Time Jitter Display), original point cloud data (LiDAR Point Cloud Display) and camera fusion data (Fusion Image Display). Config Information Display retrieves data from the `/dev/fpga_cdev` device node to get camera position/angle, time offset, and fps (frame per second). Time Jitter Display subscribes data from Time Jitter Node to handle time stamping information of LiDAR and camera sensor data. LiDAR Point Cloud Display subscribes LiDAR sensor data from LiDAR node, which mainly handles the main data stream output protocol (MSOP) data and device information output protocol (DIFOP) data from LiDAR sensor.

Fusion Image Display subscribes data from the stitch node, which gets video data from four camera sensors and stitches them into a single picture. The fusion node also integrates object detect and point cloud clustering functionality.

Note: Driver code will be open published at GitHub:
https://github.com/TianjinSiasun/Sensor_Fusion_drv

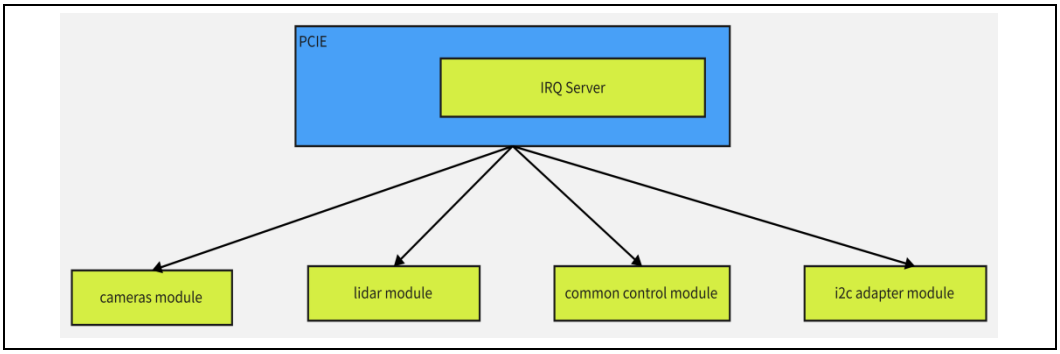
GUI demo application and ROS packages will be provided by Intel separately. Please contact FAE or PAE to get them.

2.2 Detailed Design

2.2.1 Driver Design

The driver design adopts a master-division structure (as shown in Figure 3), where the resource request and configuration of the two PCIE lanes are performed separately in the main driver portal. The requested resources are then made available to each sub-module for use.

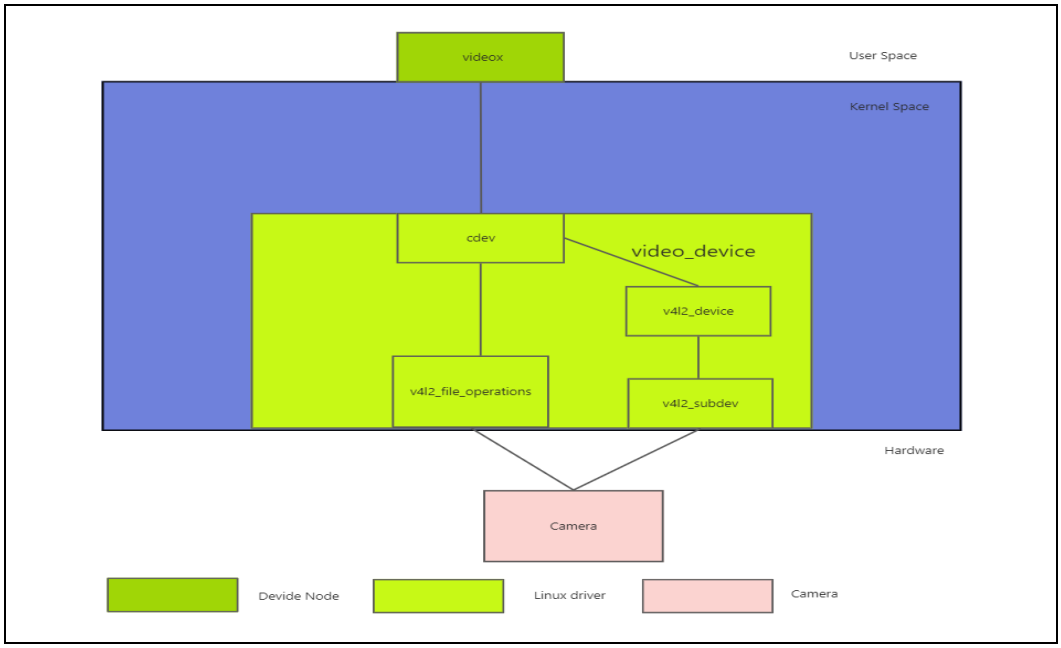
Figure 3. PCIe Driver Structure



Cameras Module Driver Design

The driver for the camera module is developed based on the V4L2 framework, which is a common driver framework of video capturing devices on Linux. The video capturing device driver registers a character device file, named *videox* in the `/dev` directory, and implements image data acquisition through the system calling interface: read, write, ioctl and mmap, etc. A basic block diagram of the v4l2 device driver system is shown in Figure 4.

Figure 4. Block Diagram of V4L2 Device



The sensor fusion project contains four camera module devices (two GMSL interfaces and two MIPI interfaces). Each module is registered as a video_device and the file_operations and essential ioctl functions are implemented in the driver program to initialize the device, set the camera configuration, and get data. There are two main functions:

- Initial configuration of camera module

  The module has an initial configuration of a standard inter-integrated circuit (I2C) configuration. The implementation is that: register 4 I2C_BUS, for camera module devices, in the related file_operations interface of the corresponding video_device, receive and send I2C data using I2C common interface to achieve initialization of camera module configuration.

- Image data acquisition

  Data acquisition is achieved by a combination of interruption and mmap mapping space. A memory space will be mapped to user layer from the system kernel space by mmap function and these spaces are capable to support Direct Memory Access (DMA). After FPGA reports interruption, user layer can call functions to have direct access to memory space data of kernel space.

Functional Interface is expressed in Table 2.

Table 2. Functional Interface of Camera Driver

Functional Interface	Description
open(FILE_VIDEO, O_RDWR)	Open device file
open(FILE_VIDEO, O_RDWR)	Open device
ioctl(fd,VIDIOC_QUERYCAP,&cap)	Query properties of device
ioctl(fd,VIDIOC_S_FMT,&fmt)	Set frame format
ioctl(fd,VIDIOC_S_PAPM,&stream_para)	Set frame rate
ioctl(fd,VIDIOC_REQBUFS,&req)	Apply for frame buffer
ioctl(fd,VIDIOC_QBUF,&buf)	Enqueue
ioctl(fd,VIDIOC_STREAMON,&type)	Start video streaming
ioctl(fd,VIDIOC_DQBUF,&buf)	Dequeue
ioctl(fd,VIDIOC_STREAMOFF,&req)	Stop video streaming
close(FILE_VIDEO)	Close device

LiDAR Driver Design

In agreement with data transmission protocol of LiDAR devices and implementation of FPGA, the normal character device driver design is applied, and makes it easy for facilitating utility layer to encapsule the driver interface as a library file.

- LiDAR driver uses DMA to the MSOP package and transmits the data of FPGA to specific memory space. The transmission status is reported in the form of interruptions. Mapping to the user space by mmap can accelerate the transmission.
- For user configuration write protocol (UCWP) and DIFOP data, ioctl command is applied to receive and deliver data.
- Initialization and synchronization settings operations, etc. of LiDAR device.

Functional interface of LiDAR device is defined as per Table 3.

Table 3. Functional Interface of LiDAR Driver

Functional Interface	Description
int Lidar_Open(void)	Open LiDAR device
void Lidar_Init(int fd)	Initialize LiDAR device
int Lidar_Get_DIFOP(int fd, RS16DifopPkt *difop)	Get LiDAR DIFOP data
int Lidar_Set_TimeStamp(int fd, RSTimestampYMD time)	Set LiDAR time stamp
int Lidar_Get_MSOP(int fd, unsigned char *mem, unsigned char *pkgnum, RS16MsopPkt *msop)	Get LiDAR MSOP data
int Lidar_Set_fps(int fd, unsigned int fps)	Set LiDAR fps
int Lidar_Set_Ethnet(int fd, RSEthNet eth)	Set LiDAR Ethernet

int Lidar_Set_FOV(int fd, RSFOV fov)	Set LiDAR FOV
int Lidar_Set_Target_Angle(int fd, unsigned int angle)	Set LiDAR target angle range
int Lidar_Set_Motor_Phase(int fd, int angle)	Set LiDAR motor phase
int Lidar_mmap_Addr(int fd, unsigned int *phy_addr)	Get LiDAR MSOP data address for mmap mapping in application
int Lidar_Close(int fd)	Close LiDAR device

PCIe Configuration

PCIe configuration implements functions of operational interfaces, such as configuration of internal buffer of FPGA, configuration of synchronization and trigger mode of cameras, etc.

Functional interface of PCIe configuration defined per Table 4.

Table 4. Functional Interface of PCIe Configuration

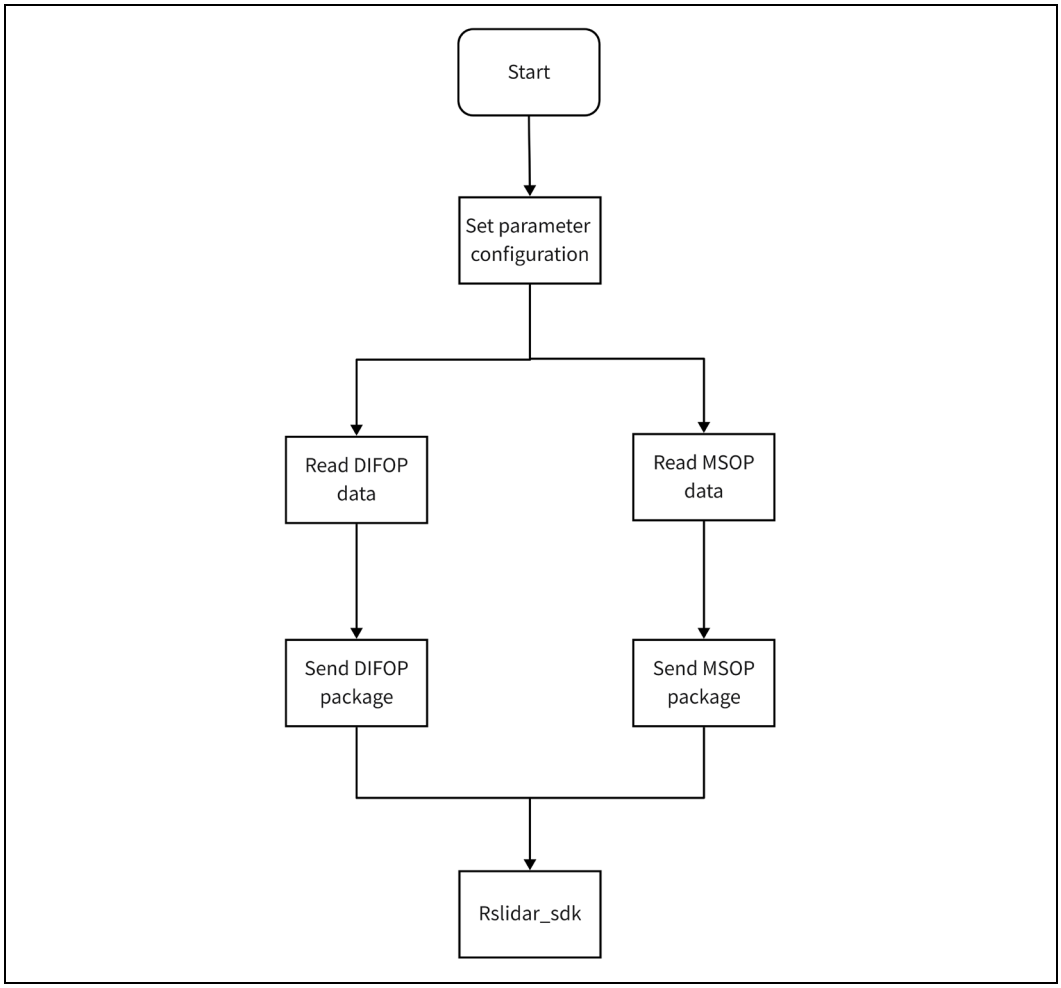
Functional Interface	Description
void Buf_Sys_Enable(int fd, bool enable)	Enabling BUF & SYNC management module.
void Buf_Config(int fd,bufcfg cfg)	Set the size and the number of buffers of FPGA for storing camera and LiDAR data
void FPGA_TimeStamp_SET(int fd, TimeStampCfg time)	Set FPGA timestamp.
TimeStamp FPGA_TimeStamp_GET（int fd）	Get FPGA timestamp.
void Trigger_Mode_Set(int fd,bool mode)	Set trigger mode of camera.
void Camera_Angle_FPS_Set(int fd, SyncCfg sync)	Set configurations for synchronization, including camera mounting angles, and time offset, etc.
SyncCfg Camera_Angle_FPS_Get(int fd)	Get FPS, camera mounting angles, offset value.

2.2.2 Middleware Layer Design

LiDAR Node

LiDAR node is mainly used to read the MSOP data and DIFOP data from LiDAR device driver; and forward them to rslidar node.

Figure 5. Flowchart of LiDAR Node



When starting the node, begin by setting the parameter configuration based on the LiDAR driver interface. The LiDAR MSOP acquisition thread will start to read MSOP data and extract the first block angle of the first MSOP package from each data frame. Then, based on the angle of the first block, it will calculate the time stamp of position 0 degree. This time stamp will be sent to Time Jitter node, and the MSOP package will be sent to rslidar_sdk as UDP packages via the ROS topic /points2. In the meantime, DIFOP acquisition thread will begin to read DIFOP data and forward it to rslidar_sdk in the form of UDP packages.

The functional interface of LiDAR node is defined as per Table 5.

Table 5. Functional Interface of LiDAR Node

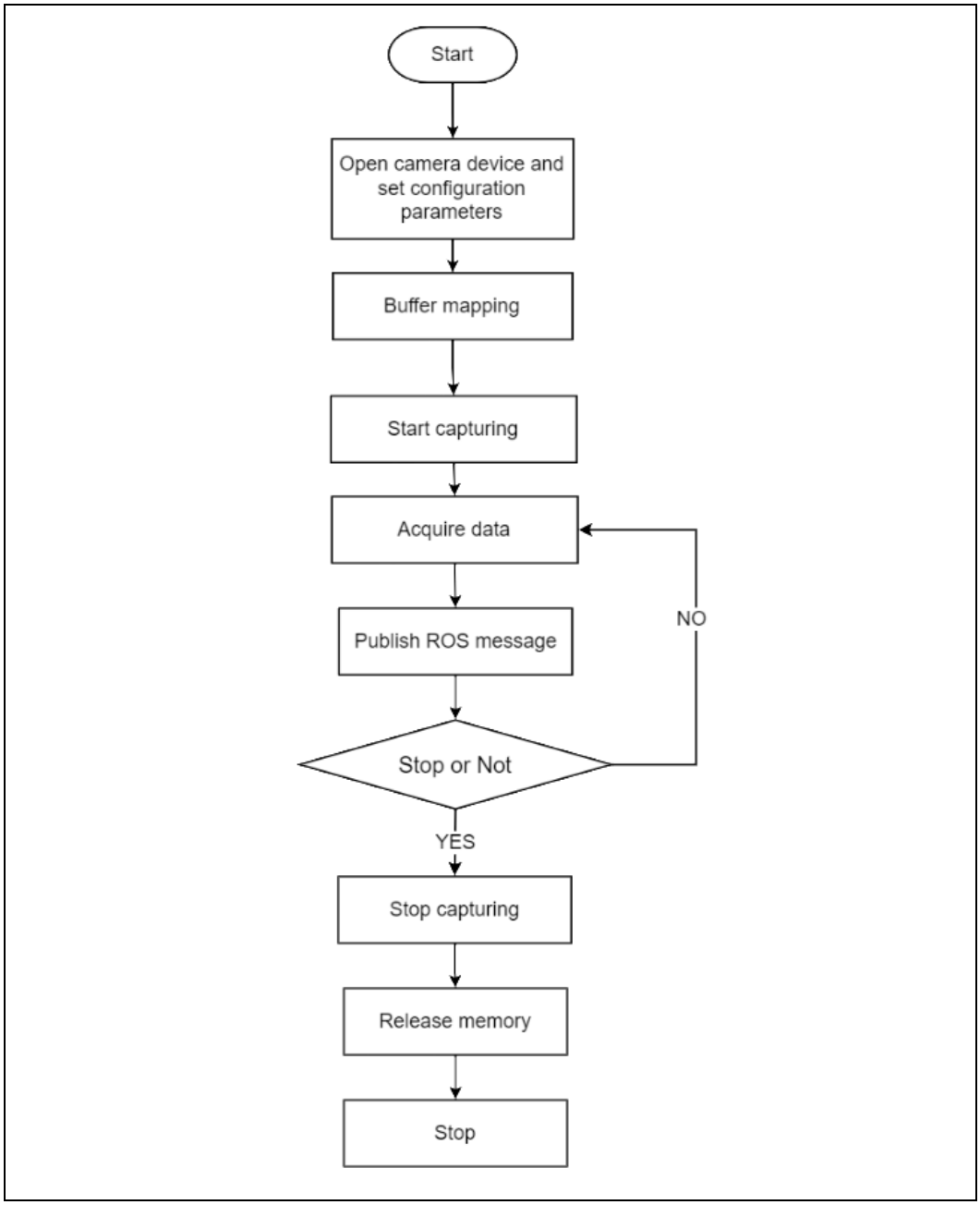
Functional Interface	Description
int init_lidar();	Initialize LiDAR device
void DifGet();	Get LiDAR device configuration

Functional Interface	Description
void MosGet();	Get MSOP data
void difop_start();	Start DIFOP
void msop_start();	Start MSOP
int LiDAR_UDP_Difop();	LiDAR difop client
int LiDAR_UDP_MSOP();	LiDAR msop client
void msop_exit()	Close MSOP thread
void difop_exit();	Close DIFOP thread

Camera Node

Camera node implements data capturing of video stream, getting, and setting the device parameters based on the V4L2 driver. To reduce low-level software dependencies, the camera node is designed as an ROS node. Camera data is published in the form of ROS messages.

Figure 6. Flowchart of Camera Node



Four threads are used to process data from four camera devices. According to the driver interface, the device is started, and configuration the parameters are delivered. Open the memory mapping space in the camera's main data area and capture data signals. When the signal has been captured, analyze the image data with OpenCV, compress it to ROS image message by cv_bridge and publish it.

Functional interface of camera node is defined as per Table 6.

Table 6. Functional Interface of Camera Node

Functional Interface	Description
int fun_open_dev(const char * path);	Open camera device
int fun_close_dev();	Close camera device
int fun_mmaping(void *buff , int size);	Buffer mapping
int fun_unmmaping(void *buff)	Buffer unmapping
int fun_start_capture();	Start capturing
int fun_stop_capture();	Stop capturing
int fun_pull_data();	Pull data
int fun_publish_data();	Publish ROS message

Clustering Node

Clustering node implements functions as point cloud pre-processing, point cloud clustering, and point cloud area picking, etc. The overall system design is separated into layers for data reading, extraction of valid region, ground plane fitting of point cloud data, point cloud clustering, implementation clustering algorithm (DBSCAN), publishing of clustering results, transferring, and testing of joint calibration. The software design is developed and transferred according to the different layers of functions. The detailed functions are shown in Figure 7.

Figure 7. Basic Block Diagram of Clustering Node

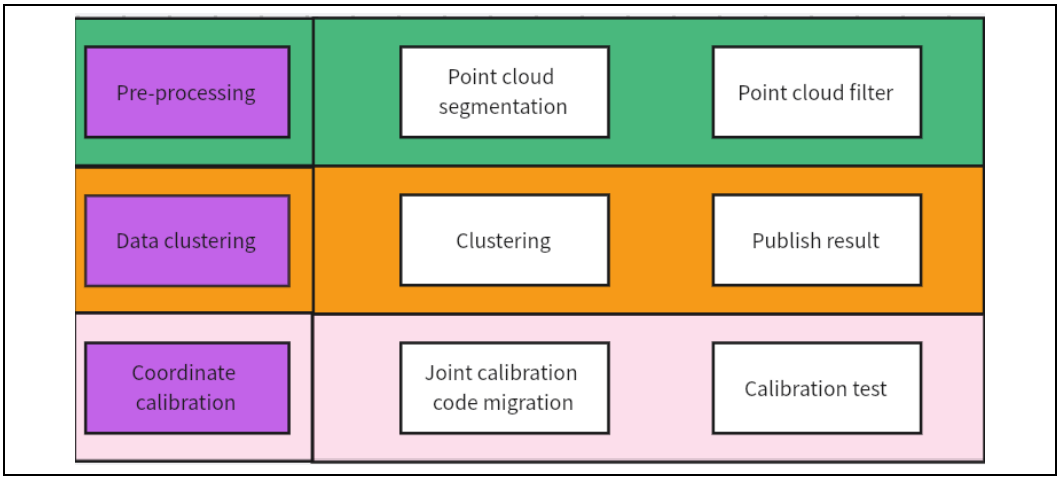
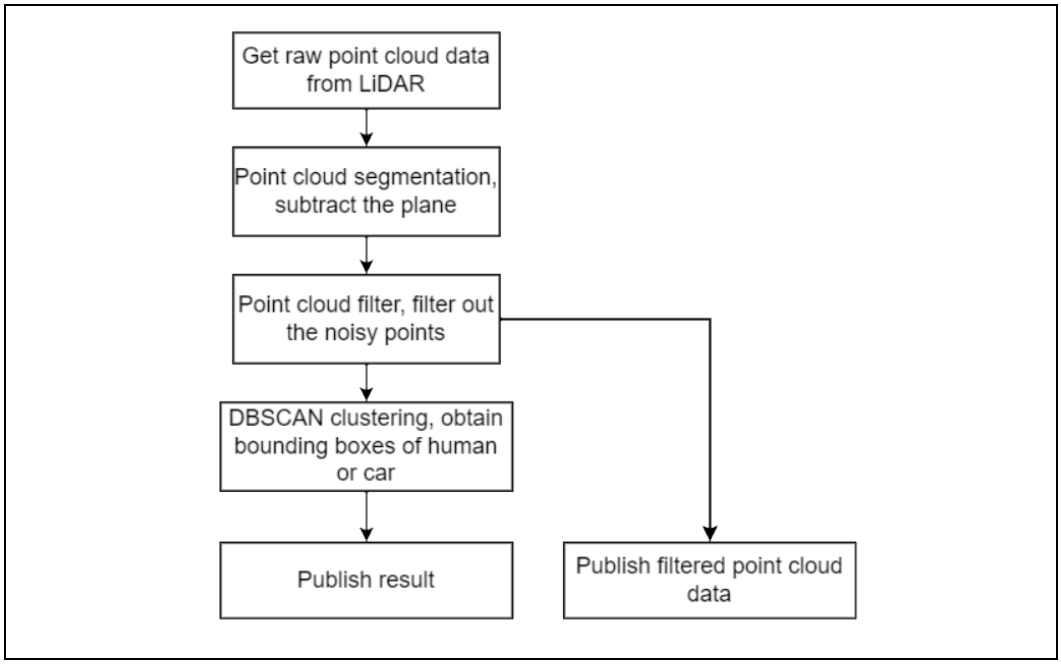


Figure 8. Flowchart of Clustering Node



Functional interface of filter module is defined as per Table 7.

Table 7. Functional Interface of Filter Module

Functional Interface	Description
int remove_nan(pcl::PointCloud<pcl::PointXYZI>::Ptr pcl_pc_in);	Remove NaN points
int statistical_filtered(pcl::PointCloud<pcl::PointXYZI>::Ptr initial_pointCloud);	Statistical filter
int radius_filtered(pcl::PointCloud<pcl::PointXYZI>::Ptr initial_pointCloud);	Radius filter
int cropbox_filtered(pcl::PointCloud<pcl::PointXYZI>::Ptr initial_pointCloud);	Cropbox filter

Functional interface of segmentation module is defined as per Table 8.

Table 8. Functional Interface of Segmentation Module

Functional Interface	Description
int init_sacSeg() ;	Module initialization
int sacSegmentation(pcl::PointCloud<pcl::PointXYZI>::Ptr filtering_pointCloud);	External points extraction

Functional interface of clustering module defined as below:

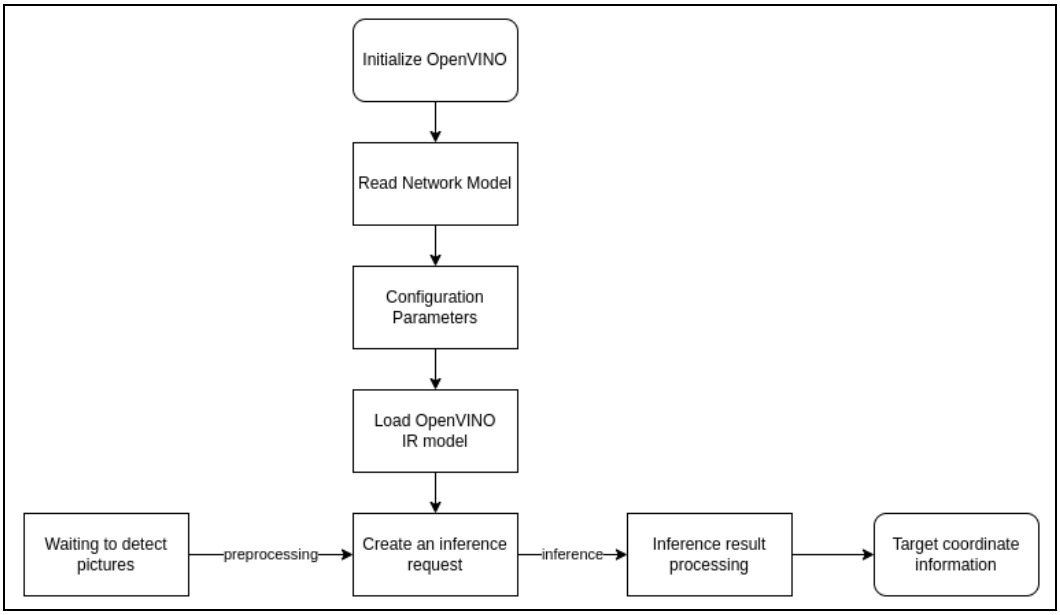
Table 9. Functional Interface of Clustering Module

Functional Interface	Description
int dbscan(pcl::PointCloud<pcl::PointXYZI>::Ptr pcl_pc_in);	DBSCAN clustering
int marker_pose_pub(std::vector <pcl::PointCloud <pcl::PointXYZI> ::Ptr, Eigen:: aligned_allocator <pcl::PointCloud <pcl::PointXYZI>::Ptr>> clusters) ;	Publish clustering result

Detect Node

Detect node retrieves data from video camera and then detect objects. It runs OpenVINO™ inference engine, with YOLOv5-nano IR model optimized on OpenVINO™. Its workflow is described as per Figure 9.

Figure 9. Detect Node Workflow



Functional interface of detect node is defined as per Table 10.

Table 10. Functional Interface of Detect Node

Functional Interface	Description
int load_ov(const std::string &xml_file, const std::string &label_file, const std::string &device = "CPU");	Load OpenVINO™ IR model
void post_process(cv::Mat &input_image, std::vector<ObjInfo_t> &out_vec, int camera_id);	Get inference result
void draw_rect(cv::Mat &input_img, std::vector<ObjInfo_t> &input_vec);	Mark object rectangles

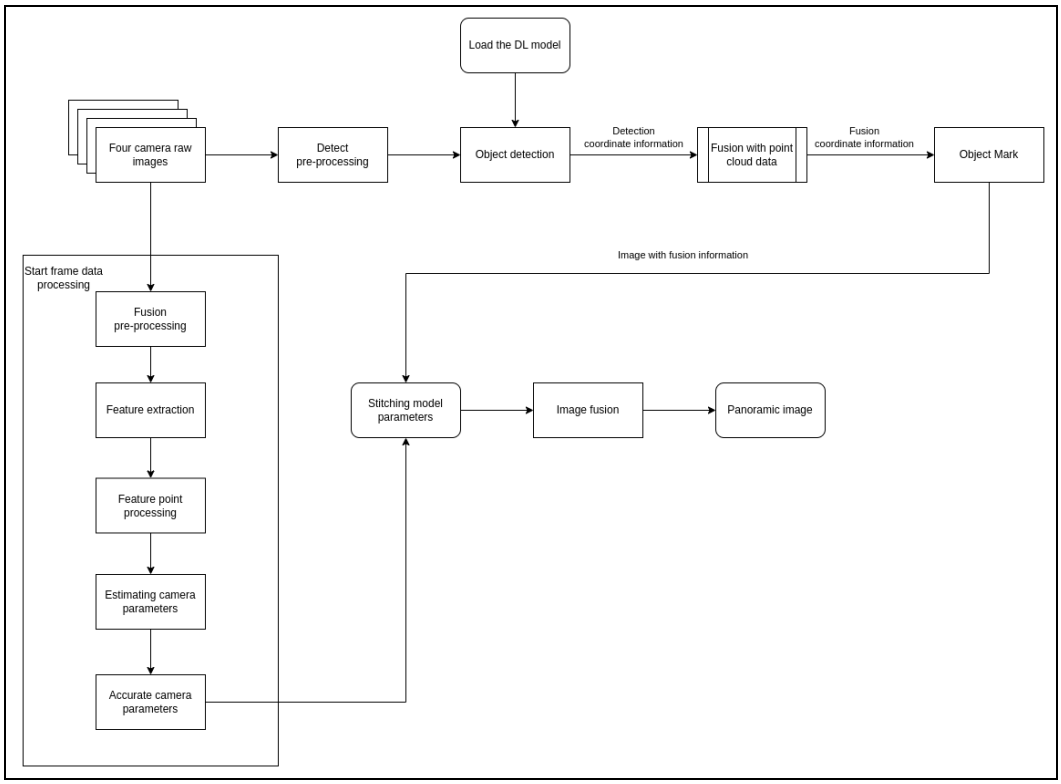
Detect result data structure defined as below:

```
typedef struct ObjInfo {
        char name[16]; // detect obj name
        double x; // rect x
        double y; // rect y
        double width; // rect w
        double height; // rect h
        double confidences; // confidences
        bool is_match; // [use] is match
        double d; // [use]distance
        double angle;
} ObjInfo_t;
```

Stitch Node

Stitch node gets image data from video cameras and fusion data of image and point cloud, and stitch data from four cameras to one single image. Its workflow is described in Figure 10.

Figure 10. Stitch Node Workflow



- Start frame data processing: Get stitching parameters.
- Fusion pre-processing: Downscaling image size, improve feature extraction speed.
- Feature extraction: Extract features of images from four cameras with non-linear AKAZE algorithm.

- Feature point processing: Apply Nearest Neighbor for feature extraction, determine if a full scene can be generated from the feature to estimate the camera parameters: according to the feature and feature points, estimate the intrinsic matrix and orthogonal rotation matrix of four images.
- Accurate camera parameters: Get optimized 3D model and camera parameters from visual reconstruction with bundle adjustment.
- Detect pre-processing: Convert fusion images to 4-dimension blob of neural network model input.
- Object detection: Load OpenVINO™ optimized YOLOv5-nano IR model and run inference.
- Object mark: Mark the result of image and point cloud data fusion in the original images.
- Image fusion: Project four images onto a sphere with model parameters to obtain a consistent field of view and make a panorama. Since the exposure is not fixed, the overall brightness of the pictures taken at different times will be different, so direct stitching will result in significant variations in light and darkness. Therefore, the exposure compensation should be set so that overall brightness of the different photos is the same, use the graph cut algorithm to find the stitching and use the multiband blending algorithm for image fusion

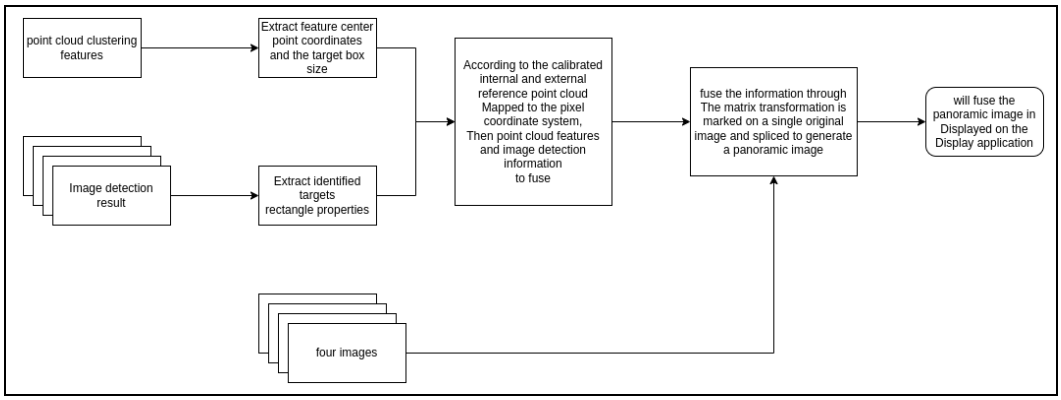
Functional interface of stitch node is defined as per Table 11:

Table 11. Functional Interface of Stitch Node

Functional Interface	Description
int picture_stitch_with_fusion_info(std::vector<cv::Mat> &in_vec_pic_mat, std::vector <std::vector<bys::ObjInfo_t> > &in_vec_info, size_t pic_nums, cv::Mat &out_pic);	Stitch images

Fusion Node

Figure 11. Fusion Node Workflow



The fusion node currently processes a feature-based fusion of LiDAR and camera data. First, the coordinate systems of multiple-sensor system should be clarified in space dimension, including LiDAR coordinate system and visual coordinate system (cameral coordinate system, image physical coordinate system, image pixel coordinate system). The LiDAR coordinate system works as a world coordinate system. As hard synchronization is implemented on FPGA hardware layer, time fusion is not implemented in the utility layer so far. The time of every frame of data is assumed to be synchronized by FPGA and only fusion of space is implemented in utility layer.

- Firstly, accept the clustering features of point cloud and detection feature of four camera images, and both data type is MarkerArray message in ROS.
- Extract the Cartesian coordinate of centroid of the point cloud feature and the size of the bounding box. In the meantime, extract the pixel coordinate of the centroid of the image feature, the size of the bounding box and the detected result.
- The extrinsic and intrinsic can be jointly calibrated with four cameras and LiDAR. The centroid coordinate of point cloud feature will be mapped to the corresponding camera image coordinate system and fused with image detection information. The information of bounding box after fusion contains distance and detection information of the object and it will be attached to four camera images correspondingly.
- Transmit four images with fused feature and information to the image stitching module.

Functional interface of fusion node is defined as per Table 12.

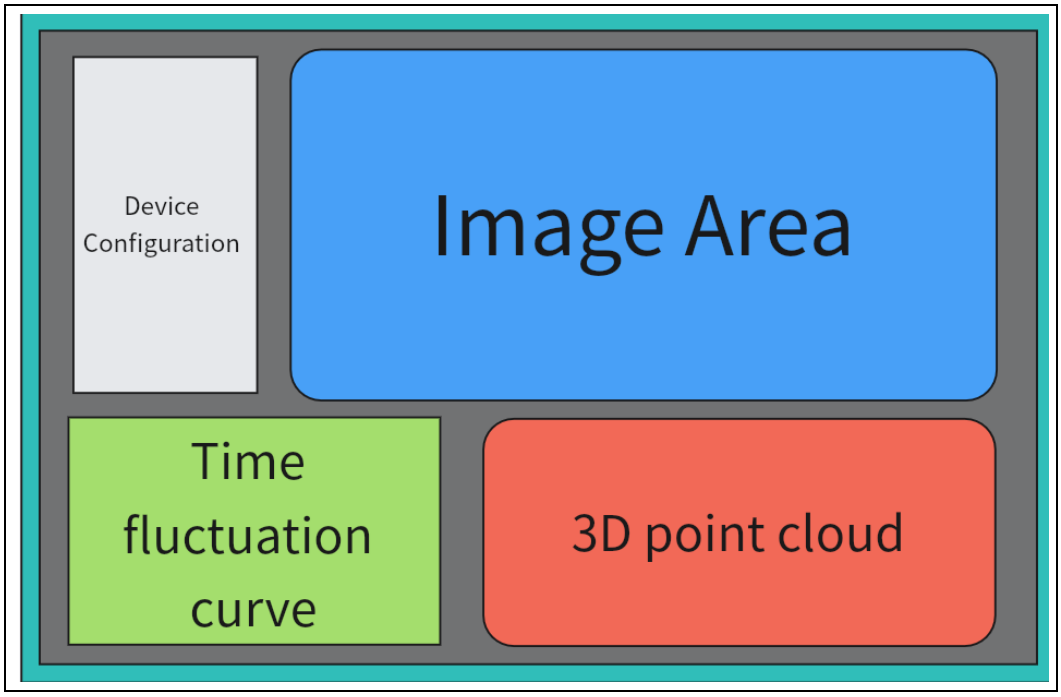
Table 12. Functional Interface of Fusion Node

Functional Interface	Description
int set_calib_files(const cv::String &calib_files);	Read calibration file
int fusionPoint(const geometry_msgs::msg::PoseArray &posesmsg, std::vector<bys::ObjInfo_t> &input_objs);	Point fusion

2.2.3 Display Application

Display application demonstrates the point cloud, the camera views and fluctuation of time difference between each camera. All data and functions will be displayed in one screen and the layout of each function block should be matched for corresponding data analysis and it has been separated to four areas:

Figure 12. Display Application Function Blocks



Point Cloud Display

- Get point cloud data via ROS2 communication protocol
- Get point cloud data: pcl::PointCloud<pcl::PointXYZ>
- Display the point cloud via QVTKWidget interface of Qt by function: pcl::visualization::PCLVisualizer().
- Modify point cloud size and coordinate viastd::shared_ptr<pcl::visualization::PCLVisualizer>
- Modify points size and color via pcl::visualization::PointCloudColorHandlerCustomPointCloud<pcl::PointXYZ>

Camera Data Display

Image data from four cameras should be merged and displayed in the application. Post-processing will be applied for better visual effect, including shadow of the images, etc.

Time Fluctuation Display

Camera 0°time stamp (T1) is retrieved based on time stamp of camera data packages. LiDAR 0°time stamp (T2) is retrieved based on time stamp of the first data package from LiDAR sensor. Latency (ΔT) is calculated by the difference of T1 and T2: ΔT = T1 - T2. In the application, time fluctuation curve is display with time as X-axis and latency as Y-axis. Max latency is also displayed.

Approach to calculate 0°time stamp of cameras and LiDAR is described below:

- 0°of camera data packages:

  Take 20fps as example, each cycle takes 50000us (t). Based on camera's hardware characteristics (exposure time, latency of data transmission, etc.), camera time offset (C) can be calculated. Then based on angle (A) of each camera's position and actual time stamp (T) of each camera, 0°time stamp (T1) can be calculated using the formula:

  $$T1 = T - A/360*t + C$$

- 0°of LiDAR data packages:

  Take 20fps as example, each cycle takes 50000us (t). Depacketize the first MSOP package to get time stamp of the first MSOP package (T), and horizontal angel (A) of the first block of MSOP UDP package, then calculate 0°time stamp (T2) according to the formula:

  $$T2 = T + (360 - A)/360*t$$

Device Configuration Display

Device configuration displays sensor configuration (LiDAR fps, camera fps, camera position, etc) retrieved from FPGA PCIe driver.

§

3.0 Reference

Reference documentation:

- Phoenix Hill FPGA RTL Design Specification
- RS-Lidar-16_User_Guide_v4.3.3_CN
- Linux Media Subsystem Documentation

§

4.0 Supported Platforms

4.1 Hardware Platform

- Mainboard: Intel Phoenix Hill Reference Board
- LiDAR sensor: RoboSense RS-LiDAR-16
- Camera sensor: OmniVision OV8865

4.2 Software Environment

- Operating System: Ubuntu 20.04
- Middleware: ROS2 (Foxy Fitzroy)
- Linux Kernel version: Kernel 5.15
- Dependency Libraries
  - OpenVINO™ 2022.1
  - OpenCV 4.5.5
  - PCL 1.12
  - VTK 8.2.0

§