SDK

LumSDK

Table of Contents

Introduction

The purpose of this document is to provide general, introductory knowledge of LumSDK, from its inner workings to the specific components that comprise this kit.

***Note**: An in-depth documentation of this chapter can be found in the HTML file generated by Doxygen (docs/html.index.html in the LumSDK).*

We will briefly discuss the “eyes” of the sensor, default settings, and discuss the packet types included in the SDK along with the its structure, calibration methods, commands (further details is provided in the HTML file mentioned above), pre-reqs required for installation and finally, the installation instructions including update procedures.

What is LumSDK?

LumSDK is a sample code that demonstrates how to communicate with the Luminar LIDAR in your system.

Model G2 Overview

The Model G2 sensor contains two sides referred to as ***eyes*** that we chose to label as blue and green (*the blue eye is located on the left and scans the right side of the field and the green eye is located on the right side of the sensor and scans the left side of the field*) as shown in Figure 1. These two sides contain separate laser beams and independent scanning mechanisms.

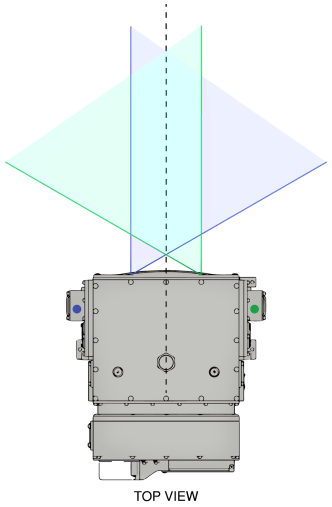

Figure 1: Horizontal Field of View

Each eye has a sixty degrees (60°) Horizontal Field of View (HFOV) which gives the whole system a 120 degrees (120°) HFOV.

A side view of the system showing the vertical field of view (VFOV) is shown in Figure 2. Each individual scanning system can achieve a maximum vertical field of view (VFOV) of thirty degrees (30°) down to a minimum of approximately eleven degrees (11°).

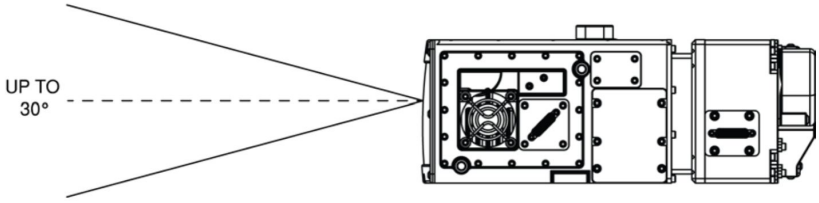

Figure 2: Model G2 System Showing the Vertical Field of View

Default Settings

The sensor has a default IP address of **10.42.0.37**

The sensor has an unchangeable default UDP port of **5517**

Disclaimer: Multicast (set of data sent across a computer network to many users at the same time) is not supported at this time.

Basic Packet Structure

The Luminar G2 system (the latest brainbox of the G model) and client machine communication occur via User Datagram Protocol or UDP (*sends packets of data called datagrams*).

UDP packets (datagrams) contain a 12-byte header preceding the payload (part of the transmitted data that is the actual message).

```
#define LUM_MAGIC_NUMBER \
  (0x1234L << 48 | 0xbeefL << 32 | 0xa1f3L << 16 |
0xfeedL)
struct lum_packet_header {
  uint64_t magic_number_long; // must equal
LUM_MAGIC_NUMBER   uint32_t packet_type;
};
```

Out of the 12 bytes in the header packet, the first 8 bytes are a unique identifier to the Luminar packet defined as the LUM_MAGIC_NUMBER. This is used to minimize the chances of another packet being misrepresented due to the connectionless nature of UDP. Therefore, any packet whose first 8 bytes are not equal to the LUM_MAGIC_Number are ignored.

The remaining 4 bytes refer to the packet *type*. The packet type indicates how to interpret the remaining bytes of the payload. The table below displays the values for the various packet types.

***Note**: "Client" refers to the user who is utilizing and implementing the LumSDK onto their system*

Byte	1	2	3	4	5	6	7	8	9	10
	ed	fe	f3	a1	ef	be	34	12	Type Low	Type Hi (0)

Packet Types

There are several UDP packet types included in the latest version of the Luminar interface. Each packet type has a defined interaction mode.

Disclaimer: Not all packet types receive an acknowledgement packet.

Note*: This document uses C structs to document the packet formats. Some structs end in a char array member of size 1 (e.g., "*`char payload[1]`*"). This denotes a flexible array, since the payload of the packet past that point is not a fixed size. We use a 1-sized array rather than the more conventional 0-sized array because the latter is not C++-compliant.*

In addition to this document, sample source code is provided demonstrating usage of these packets in simple C and C++ programs, including a visualizer using Robot Operating System's (ROS) rviz application. The following table documents the appropriate values for the packet_type field.

Each packet type is described below.

Packet Type	Value in packet_type field of header
Set IP Address Packet	5
Timestamp Packet	6
TLV Packet	7
Heartbeat Packet	8

Note: "*LIDAR", "sensor" and "server" are used interchangeably in this document.*

IP Address Packet

The **Set IP Address Packet** is sent by the client to the LIDAR sensor ***only***.

- Because receiving a packet from the new address provides acknowledgement, no acknowledgment packet is sent to the sender (non-responsive).
- The sensor will immediately try to change the IP Address if the packet is valid.
- The client system should then communicate with the sensor at the new IP address to verify that the IP address was changed.
- The client system must know the sensor's initial IP address in order to set its new IP address.
- Because the sensor always boots up with its default IP address of **10.42.0.37**, packets can be directly sent to the default IP address.

Disclaimer: Setting the IP address via UDP broadcast packets is not supported at this time, but this can be maneuvered around via Linux's virtual network interfaces, in case your network is not a 10.x.x.x subnet.

```
struct address_packet_type
{
     char packetType;
     char reserved[3];
     uint32_t newIPAddress;
};
```

Timestamp Packet

The timestamp packet is used to inform the client of the time difference between the sensor's clock and the client's clock. The timestamp packets are sent by both the client and the LIDAR Sensor.

- When a client sends a Timestamp packet to the sensor, the value of the sent timestamp field is ignored.
- The response packet has the same sequence number as the request packet, but as the packet enters the sensor's *receive queue* during bootup, the timestamp field is set to 32-bit timestamp.

Note*: Since the 32-bit timestamp is a 32-bit unsigned integer, it will reset to zero every 71 minutes.*

Note*: The purpose of the sequence number is to account for the possible reordering of UDP packets.*

```
struct timestamp_packet_struct
```

```
{
    Char packetType;
    Char reserved1[3];
    uint32_t sequenceNum;
    uint32_t timeStamp; // In microseconds
};
```

TLV Packets

Type-length-vector (TLV) packets are the primary method for status and command communication with the G2 system.

Type is an identifier indicating how to interpret the payload, *length* is the size of the payload and *vector* is the payload. The payload of a UDP packet can contain one or more TLV commands.

The client uses TLV to send command packets to the G2 Control Box via the flexible array payload field, while the G2 Control Box uses it to communicate a debug output.

LUM_TLV_PACKET::tlvs contains an array of LUM_TLV objects, each of which represents one TLV command.

Note*: The reason that it is not typed as LUM_TLV_ARRAY is because LUM_TLV is a flexible struct (it doesn't have a fixed size).*

Every LUM_TLV_PACKET received by the LIDAR will cause another LUM_TLV_PACKET to be returned back to the sender, containing the same sequence number for bookkeeping purposes.

```
struct
lum_tlv_packet
{
                    union {
                      uint16_t magic_number_shorts[4];
                      uint64_t magic_number_long;
                    };
```

```
 uint32_t packet_type; /**< Must be equal to
LUM_TLV_PACKET_TYPE */
 uint32_t sequence;
 char tlvs[1];
};


typedef struct lum_tlv LUM_TLV;
struct lum_tlv {
 uint16_t type;
 uint16_t length;
 char vector[1];
};
```

TLV Status Packets

Status alerts displaying bits ranging from 15-9 are informational alerts. (Ex: x0800 informs laser was turned on with a TLV command).

Status alerts displaying bits ranging from 8-0 are hazardous and should be dealt with (ex: a galvo has stopped).

Definition of laser status bit:

```
LASER_STAT_GALVO_AMP_BX_BAD  = 0x0001,
```

```
LASER_STAT_GALVO_AMP_BY_BAD  = 0x0002,

LASER_STAT_GALVO_AMP_GX_BAD  = 0x0004,

LASER_STAT_GALVO_AMP_GY_BAD  = 0x0008,

LASER_STAT_HEARTBEAT_TIMEOUT = 0x0010,

LASER_STAT_BOARD_OVER_105    = 0x0020,

LASER_STAT_FAN_PWM_BAD       = 0x0040,

LASER_STAT_HW_SAFETY_BAD     = 0x0080,

LASER_STAT_SYS_ON_SW_OFF     = 0x0100,
  /* Laser was turned off by SYS_ON switch */
LASER_STAT_TLV_CMD_OFF       = 0x0200,
  /* Laser was turned off by a TLV command */
LASER_STAT_ALL_DOWN          = 0x07ff,
  /* Laser is on due to a TLV command */
LASER_STAT_ALL_SW_BAD        = 0x077f,
  /* All  status bits that indicate laser is down */
```

Note: *Documentation of possible TLVs can be found in:*

`include/fpga firmware/sw/lum_tlv_eth_protocol.h`

Heartbeat Packet

The Heartbeat Packet serves two primary purposes:

1. To set the destination IP address and UDP port where the sensor will send the LIDAR data packets.
2. If a heartbeat packet is not received within 3 seconds, the LIDAR will shut off the laser to provide a safe shutdown in the event of a malfunction.

***Note**: It is advisable to send a heartbeat packet several times per second to ensure the timer is not tripped. The sender of this packet does not receive an acknowledgement packet, but will result in LIDAR data packets being sent to the sender if successfully received.*

```
struct packet_acquire_enable_struct
{
  char    packet_type; // LUM_MULTI_DISTANCE_PACKET_TYPE
  char    reserved[3];
};
```

Multi-Return Packet Format

LIDAR data is sent to the client in a group of points.

There are 1,000 points in each line per eye.

Each line is transmitted as a group of UDP packets.

The *lum_group_header* contains the information necessary to identify a line (by the line number). The *lum_group_packet* contains a header and a payload (either in angle or range data).

Angles

Angles are transmitted in 16-bit signed fixed-point values which must be converted to degrees or radians. Within the angle packet header, the values of line, num_samples, and first_sample_index allow you to index angle values within a frame.

Angle payloads are transmitted in 3 angle packets of 334, 334, and 332. Each value represents a point in a line that add up to 1,000 points per eye (334+334+332 = 1000). These points represent the rising edge of each eye. Each point has a pair of angles (azimuth/elevation).

There are 3 angle packets per line per eye.

Ranges

Ranges are transmitted as 32-bit signed integers.

Note:

- *The Time-Of-Flight is the time it takes to travel to the target and back from the target (twice the target distance)*
- *The lsb of the time-of-flight measurement is 13pS.*
- *METER_PER_BIT = (3.0e8 m/s / 2) * 1e-12 s/ps * 13.0 ps/bit * Factor to convert raw time of flight data into meters.*

Along with the line and num_samples values from the header, some bits out of the 32 allow you to index the range data within a frame.

Disclaimer: The return index is not represented in the packet. Returns are provided in order.

Data values (which immediately follow the header: *lum_group_header*) for multi-return packet data are formatted as follows:

Note: *B = bits.*

- B31-12 == Range time Value
- B11 == 0
- B10 == Rising Flag (notifies when the rising/falling edge values are returned, in other words, for every range returned, so is a Rising flag. Rising flag only occurs with Range.
- B9-0 ==X position (the x position refers to the location of the pixel in a line (labeled 0-999).

Note*: There may be multiple returns for one pixel. Multiple return pixels will never cross a packet boundary. All samples in a packet are contained in a single line.*

The multi-return packet format is documented below:

```
typedef struct
{
uint8_t  tag; // 2=angle data, 3=range distance data
```

```
uint8_t  eye; // 0 for green rising, 1 for blue rising, 2 for     green falling, 3 for blue falling
uint16_t line;// horizontal scan line # for first sample
uint16_t num_samples;// number of samples   uint16_t first_sample_index;// index of first sample (angle data only)
uint32_t first_timestamp_us;// Timestamp of first angle sample in microseconds since bootup (not for range data)
} lum_group_header;
```

Calibration Methods

This section outlines the basic calibration values needed for each G2 sensor. The G2 Sensor consists of multiple components. Three of those components are a laser timing system, an azimuth angle galvanometer, and an elevation angle galvanometer. Each of these has calibration values that need to be set to produce correct 3D points.

The laser timing system gives Time-Of-Flight (TOF) measurements. An offset value is automatically added to the time measurement values to compensate for optical and electrical path delays. An additional constant may be added to the TOF value to further compensate for system to system variations.

The azimuth and elevation systems both have true zero positions (the middle), and angle widths. For elevation, this is parameterized in terms of a minimum and maximum angle. These need to be calibrated for each eye. Typical values are as follows:

- Highest elevation angle: 3 degrees
- Lowest elevation angle: -17 degrees
- Azimuth angle width: 60 degrees
- Azimuth angle center: 0 degrees

Along with the sample code, a sample calibration file is provided in a yaml format. These are loaded into a parameter server on software startup.

lumsdk_point_cloud_node.cpp demonstrates how to use the calibration parameters to un-discretize and correct the raw data, in the process of transforming raw data into point clouds.

Calibration is currently implemented as post-processing in software; it therefore does not affect raw data coming from the FPGA.

***Note**: Users are therefore not bound to implement our set of calibration parameters, and can choose a separate set to better fit their needs.*

Initial setup

Prerequisites

- Robot Operating System or ROS (for visualization)
- A network interface on a subnet containing IPv4 address 10.42.0.37
- Curl to download sensor firmware (to download it from our web server).
- Internet connection

Install dependencies:

```
cd <path-to-lumsdk-dir>
./install_deps.sh
```

***Note**: The initial setup is done for you on delivery of a visualization computer.*

Disclaimer: If running on a system not provided by Luminar, you may need to replace enp0s31f6 in install_deps.sh with the name of your Ethernet network interface. This can

be found by running ifconfig. This script installs prerequisites and sets up the network interface.

Command Type Example Payload

Sensor Characteristic	Value	Command
Laser State	**On (up)**	**u**
	Off (down)	**d**
Laser Power	**0 to 100**	**“value”p**
Threshold Value Green	**0 to 100**	**“value”g**
Threshold Value Blue	**0 to 100**	**“value”b**
Change Line Mode	**640 lines**	**0f**
	64 lines	**1f**
Uniform Y Scan	**<fov> centered on horizon**	**15y**
FPGA Holdoff Delay	**150 to 250; intervals of 10**	**“value”h**
Galvo/ADC Offset Value Green	**0 to 30**	**“value”G**
Galvo/ADC Offset Value Blue	**0 to 30**	**“value”B**

Note: The lumsdk_node provides a sample terminal for commanding the sensor.

Key Files for Detailed Documentation of Packet Formats

- Documentation of packet format for raw data, addresses, timestamps:

  `include/fpga-firmware/sw/lum_eth_protocol.h`:
- Documentation of packet format for TLVs:

  `include/fpgafirmware/sw/lum_tlv_eth_protocol.h`
- Demonstrates how to undiscretize the raw data:

  `include/conversions.hpp` and `src/conversions.cpp`
- Demonstrates how to send heartbeat packets, and parse raw data packets:

· `include/fpga_client3.hpp` and `src/fpga_client3.cpp`
- Demonstrates how to send/receive TLV packets. Sample functions for generating yscan distributions:

· `include/tlv/*` and `src/tlv*`
- Sample ROS Node that produces point clouds from the raw data and sends TLVs for sample commands:

  `lumsdk_point_cloud_node.cpp`

Build ROS

Note: *This is already done for you on delivery of a visualization computer.*

1. *cd <path-to-lumsdk-dir>*
2. *catkin_make*

If this command fails, run this source command:

source /opt/ros/kinetic/setup.bash

And run *catkin_make* once again.

Note*: Failure occurs because the environment must be sourced prior.*

LuminView

LuminView is a custom point cloud visualization tool that provides improved image quality over other tools such as rviz due to its angle smoothing feature that reduces noise in the point cloud.

How to Use LuminView

LuminView provides a return index selector and a rising/falling edge toggle button (which allows for quick navigation between viewing different sets of returns).

Rotating the camera is done by left clicking and dragging in the direction of desired rotation.

The arrow keys are used to:

- zoom (without modifier key),
- pan (*arrow key + Shift)*
- orbit the camera(*arrow key+Shift)* in a particular direction.

Run LuminView

1. `cd <path-to-lumsdk-dir>`
2. `source devel/setup.bash`
3. `roslaunch luminar_point_cloud luminview.launch`

Note: If error *`Cannot launch node of type`* appears, that means the file is not executable. In the main directory, run:

```
sudo chmod +x -R
```

Source the environment again (if you have exited the terminal)

***Note**: It will often take several seconds for the laser to come to full effect.*

Run Standard Visualizer (rviz)

After opening up LuminView and setting your desired parameters,open up rviz:

1. `cd <path-to-lumsdk-dir>`
2. `source devel/setup.bash`
3. `roslaunch luminar_point_cloud release.launch`

The following command retrieves the raw green and blue eye data:

rosbag play multiple-returns-sample-data.bag --loop --topics /luminar_raw_blue_data /luminar_raw_green_data