



Xavia ROS2 Driver Manual

Contents

1	Table of Contents	1
2	Introduction	1
3	System Requirements	1
4	Installation	2
4.1	Package Extraction	2
4.2	Environment Setup	2
4.3	Verification	2
5	Getting Started	2
5.1	Launch Files	2
5.2	Initial Network Configuration	3
5.3	Verification	3
6	Node Reference	3
6.1	Lifecycle Pattern	3
6.2	Lidar Node	4
6.3	Camera Node	5
7	Working with Node Data in code	7
7.1	C++ Subscriber Example	7
7.2	Python Subscriber Example	8
7.3	Camera Image Subscriber	9
8	Multiple Sensors	11
9	Troubleshooting	11
9.1	Cannot Connect to Sensor	11
9.2	No Data Published	11
9.3	'xavia_lidar_node' not found	11
9.4	High CPU or Memory Usage	12

9.5	Message Loss or Buffer Overflow	12
9.6	Network Performance	12

Version: 2.1.0

ROS2 Distribution: Humble Hawksbill

Operating system: Ubuntu 22.04 (x86_64)

Document Date: April 2026

1 Table of Contents

1. Introduction
 2. System Requirements
 3. Installation
 4. Getting Started
 - Launch Files
 - Initial Network Configuration
 - Verification
 5. Node Reference
 - Lifecycle Pattern
 - Lidar Node
 - Camera Node
 6. Working with Node Data in Code
 - C++ Subscriber Example
 - Python Subscriber Example
 - Camera Image Subscriber
 7. Multiple Sensors
 8. Troubleshooting
-

2 Introduction

The Xavia ROS2 Driver provides seamless integration between XenomatiX Xavia lidar and camera sensors and the Robot Operating System 2 (ROS2). It enables acquisition of real-time point cloud data from lidar sensors and synchronized camera images through standard ROS2 mechanisms.

The driver is built on the ROS2 lifecycle node pattern for reliable operation, publishes standard `sensor_msgs` for maximum compatibility with ROS2 tools and applications, and supports multiple simultaneous sensors operating independently.

3 System Requirements

Operating System: - Ubuntu 22.04 LTS (x86_64 architecture)

ROS2: - Humble Hawksbill (or compatible distributions) - Install from: <https://docs.ros.org/en/humble/Installation.html>

Network: - Gigabit Ethernet connection with jumbo frames enabled for optimal performance - Direct connection to sensor network recommended

4 Installation

4.1 Package Extraction

Extract the driver package to an appropriate location:

```
tar -xzf Xavia_ROS2_driver_linux_x64_Humble.tar.gz -C /opt/ros/xavia_ros2
```

4.2 Environment Setup

Before launching any nodes, source the required setup files in order:

```
source /opt/ros/humble/setup.bash
```

```
source /opt/ros/xavia_ros2/share/xavia_ros2_driver/local_setup.bash
```

Both sources are required. The first initializes the ROS2 runtime environment, and the second configures the driver package paths and dependencies.

4.3 Verification

Verify successful installation:

```
ros2 pkg list | grep xavia
ros2 pkg executables xavia_ros2_driver
```

Expected output:

```
xavia_ros2_driver xavia_camera_node
xavia_ros2_driver xavia_lidar_node
```

5 Getting Started

5.1 Launch Files

The driver provides three launch files for typical usage scenarios. All launch files automatically activate the nodes on startup.

5.1.1 Lidar Only

Launch the lidar node:

```
ros2 launch xavia_ros2_driver xavia_lidar.launch.py
```

Optional parameters (more information later):

```
ros2 launch xavia_ros2_driver xavia_lidar.launch.py \
  sensor_ip:=10.10.100.11 \
  frame_id:=lidar \
  buffer_size:=224
```

5.1.2 Camera Only

Launch the camera node:

```
ros2 launch xavia_ros2_driver xavia_camera.launch.py
```

Optional parameters (more information later):

```
ros2 launch xavia_ros2_driver xavia_camera.launch.py \  
  camera_ip:=10.10.100.12 \  
  pixel_format:=RGB8
```

5.1.3 Combined Lidar and Camera

Launch both nodes in a single terminal:

```
ros2 launch xavia_ros2_driver xavia_combined.launch.py
```

5.2 Initial Network Configuration

Ensure your system is on the same subnet as the sensor:

Default Sensor Network: - IP: 10.10.100.11 - Subnet: 255.255.255.0 - Port: 4000

Example Linux Configuration:

```
sudo ifconfig eth0 10.10.100.100 netmask 255.255.255.0
```

Verify connectivity: ping 10.10.100.11

5.3 Verification

In a separate terminal, after launching the nodes, verify that data is being published:

```
ros2 topic list  
ros2 topic hz /xavia_lidar_node/points_line
```

6 Node Reference

6.1 Lifecycle Pattern

Both lidar and camera nodes implement the ROS2 lifecycle pattern, which provides structured initialization, activation, and shutdown sequences. The lifecycle state machine has the following states:

- **Unconfigured:** Initial state, node is not configured
- **Inactive:** Node is configured but not publishing data
- **Active:** Node is publishing data
- **Finalized:** Node has shut down

Check the current state of a node:

```
ros2 lifecycle get /xavia_lidar_node
```

Manually transition between states if needed:

```
ros2 lifecycle set /xavia_lidar_node configure  
ros2 lifecycle set /xavia_lidar_node activate  
ros2 lifecycle set /xavia_lidar_node deactivate  
ros2 lifecycle set /xavia_lidar_node cleanup
```

The provided launch files automatically configure and activate nodes on startup.

6.2 Lidar Node

The lidar node publishes point cloud data acquired from the Xavia sensor.

Node Name: xavia_lidar_node

Parameters:

Name	Type	Default	Description
sensor_ip	string	10.10.100.11	IP address of the lidar sensor
sensor_port	int	4000	TCP port of the lidar sensor
frame_id	string	xavia_lidar	TF frame identifier for the point cloud
buffer_size	int	224	Number of scan lines to buffer before publishing

Published Topics:

- `points_line` (`sensor_msgs/PointCloud2`): Point cloud published per scan line at ~1120 Hz. Each point cloud contains a single line of measurements. This topic provides the lowest latency and distributes load evenly in time.
- `points_frame` (`sensor_msgs/PointCloud2`): Point cloud published per frame at ~20 Hz. The node buffers scan lines according to `buffer_size` and publishes a complete frame. This topic reduces message frequency at the cost of higher latency.
- `diagnostics` (`diagnostic_msgs/DiagnosticArray`): Reports node status, errors, and connection issues.

Select either `points_line` or `points_frame` topic for subscription based on application requirements. The line mode is recommended for time-sensitive applications, and if applications can already run on partial data. Frame mode reduces subscriber load and removes the need for aggregation in case of visualization.

Note: Network issues may cause scan lines to be lost. Do not assume every line is received or that frames contain the complete expected number of lines. Design robust applications that handle missing or incomplete data.

Services:

- `start_sensor` (`std_srvs/Trigger`): Start lidar acquisition
- `stop_sensor` (`std_srvs/Trigger`): Stop lidar acquisition
- `reboot_sensor` (`std_srvs/Trigger`): Reboot the sensor

Point Cloud Fields:

Each point in the published `PointCloud2` contains the following fields:

Field	Type	Units	Description
x	float32	mm	X coordinate (forward direction)
y	float32	mm	Y coordinate (lateral, positive=left)
z	float32	mm	Z coordinate (height, positive=up)
intensity	float32	%	Reflectivity percentage
ambient	uint16	-	Background light level
distance	float32	mm	Radial distance from sensor origin
confidence	float32	0.0-1.0	Confidence level of the measurement

6.3 Camera Node

The camera node publishes images acquired from the Xavia 6D camera add-on.

Node Name: xavia_camera_node

Parameters:

Name	Type	Default	Valid Values	Description
camera_ip	string	10.10.100.12	Valid IPv4	IP address of the camera
frame_id	string	xavia_camera	Any	TF frame identifier for images
buffer_size	int	5	1-20	Number of frames to buffer
pixel_format	string	RGB8	RGB8, RGB12, BayerRG8, BayerRG12, BayerRG12packed	Output pixel format
exposure_time	int_us	0	0-100000	Exposure time in microseconds (0 = auto)

Published Topics:

- `image_raw` (`sensor_msgs/Image`): Camera images in the format specified by `pixel_format`
- `camera_info` (`sensor_msgs/CameraInfo`): Camera calibration and intrinsic parameters
- `diagnostics` (`diagnostic_msgs/DiagnosticArray`): Node status and error reporting

Services:

- `start_camera` (`std_srvs/Trigger`): Start image acquisition
- `stop_camera` (`std_srvs/Trigger`): Stop image acquisition

Image and Camera Info Messages:

The camera node publishes two topics: `image_raw` containing the pixel data, and `camera_info` containing calibration parameters.

Image Message (`sensor_msgs/Image`):

Field	Type	Description
encoding	string	Pixel format (e.g., "rgb8", "bayer_rggb16")
width	uint32	Image width in pixels
height	uint32	Image height in pixels
step	uint32	Row stride in bytes (may include padding for alignment)
data	uint8[]	Pixel data in row-major order
header.stamp	Time	Timestamp when image was captured
header.frame_id	string	TF frame identifier (set by <code>frame_id</code> parameter)

Camera Info Message (`sensor_msgs/CameraInfo`):

Contains camera intrinsic calibration and distortion parameters. The message includes:

Field	Type	Description
width	uint32	Image width in pixels
height	uint32	Image height in pixels
K	float64[9]	3x3 intrinsic camera matrix (row-major) - NOT USED
D	float64[]	Distortion coefficients - NOT USED
header.stamp	Time	Timestamp (synchronized with image)
header.frame_id	string	TF frame identifier
distortion_model	string	distortion model name (plumb_bob)

Note: the K and D fields are not filled in. If you need to undistort the camera image, please consider measuring these parameters yourself.

Pixel Formats (encoding):

- RGB8: 8-bit RGB, demosaiced. Recommended for most applications.
- RGB12: 12-bit RGB (but delivered in 16 bit), demosaiced. Largest memory footprint but preserves more color detail.
- BayerRG8: 8-bit Bayer pattern, no processing. Smallest memory and network footprint. Requires external demosaicing.
- BayerRG12: 12-bit Bayer pattern, no processing. Requires external demosaicing. Delivered in 16 bit.
- BayerRG12packed: 12-bit Bayer pattern, packed into 12 bits per pixel. Least network traffic while preserving full bit depth. Delivered in 16 bit.

Demosaicing requires processing power from the device on which the node is running. Bayer formats skip this step, but require demosaicing in a later step.

Note: In ros2, the pixelformat in the message are called as follow:

- RGB8 = rgb8
- RGB12 = rgb16
- BayerRG8 = bayer_rggb8
- BayerRG12 = bayer_rggb16
- BayerRG12packed = bayer_rggb16

General advice:

- If you want to only visualize the images on screen, then use the default RGB8.
- If you want to store as e.g. jpg, use RGB8.
- If you want to process or store the images in full color depth (RAW, TIFF, png, ...), use RGB12.
- If you can demosaic later and want to preserve memory, use one of the Bayer formats:
 - Select BayerRG8 for lowest memory requirements and lower color depth.
 - Select BayerRG12 if full color depth is required.
 - Select BayerRG12packed if the network is congested but you need full color depth.
- The Bayer formats can also be used to reduce processing power if demosaicing can be postponed to replay.

Note: ROS2 image message does not support a 12bit format. As such, the BayerRG12packed format only uses the 12 bit over the network, but expands to 16 bit when translating to ROS2. Hence, BayerRG12 and BayerRG12packed have the same ros2 message, but will reduce the network load, at the cost of extra processing to unpack from 12 to 16 bits.

7 Working with Node Data in code

Using the above drivers will publish the data into the ROS2 framework. Existing nodes and visualizers can then consume the data. This section provides a small intro into creating your own subscriber in code in order to process the data in your own way. These assume that the driver nodes are running and managed elsewhere (or you can create a launch file to launch both the Xavia driver nodes and your own node together).

7.1 C++ Subscriber Example

Subscribe to point cloud data in C++:

```
#include <rclcpp/rclcpp.hpp>
#include <sensor_msgs/msg/point_cloud2.hpp>
#include <sensor_msgs/point_cloud2_iterator.hpp>

class PointCloudSubscriber : public rclcpp::Node
{
public:
    PointCloudSubscriber() : Node("point_cloud_subscriber")
    {
        subscription_ = create_subscription<sensor_msgs::msg::PointCloud2>(
            "/xavia_lidar_node/points_line",
            rclcpp::SensorDataQoS(),
            std::bind(&PointCloudSubscriber::cloudCallback, this, std::placeholders::_1)
        );
    }

private:
    void cloudCallback(const sensor_msgs::msg::PointCloud2::SharedPtr msg)
    {
        // Create iterators for all available fields
        sensor_msgs::PointCloud2ConstIterator<float> iter_x(*msg, "x");
        sensor_msgs::PointCloud2ConstIterator<float> iter_y(*msg, "y");
        sensor_msgs::PointCloud2ConstIterator<float> iter_z(*msg, "z");
        sensor_msgs::PointCloud2ConstIterator<float> iter_intensity(*msg, "intensity");
        sensor_msgs::PointCloud2ConstIterator<float> iter_confidence(*msg, "confidence");
        sensor_msgs::PointCloud2ConstIterator<float> iter_distance(*msg, "distance");
        sensor_msgs::PointCloud2ConstIterator<uint16_t> iter_ambient(*msg, "ambient");

        // Iterate through all points
        for (size_t i = 0; i < msg->width;
            ++i, ++iter_x, ++iter_y, ++iter_z, ++iter_intensity,
            ++iter_confidence, ++iter_distance, ++iter_ambient)
        {
            float x = *iter_x;
            float y = *iter_y;
            float z = *iter_z;
            float intensity = *iter_intensity;
            float confidence = *iter_confidence;
            float distance = *iter_distance;
            uint16_t ambient = *iter_ambient;
            // Process point
        }
    }
}
```

```
    rclcpp::Subscription<sensor_msgs::msg::PointCloud2>::SharedPtr subscription_;
};

int main(int argc, char** argv)
{
    rclcpp::init(argc, argv);
    auto node = std::make_shared<PointCloudSubscriber>();
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}
```

7.2 Python Subscriber Example

Subscribe to point cloud data in Python:

```
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import PointCloud2
from sensor_msgs_py import point_cloud2

class PointCloudSubscriber(Node):
    def __init__(self):
        super().__init__('point_cloud_subscriber')
        self.subscription = self.create_subscription(
            PointCloud2,
            '/xavia_lidar_node/points_line',
            self.cloud_callback,
            10
        )

    def cloud_callback(self, msg):
        # Extract all points with all fields
        for point in point_cloud2.read_points(msg):
            x = point[0]
            y = point[1]
            z = point[2]
            intensity = point[3]
            ambient = point[4]
            distance = point[5]
            confidence = point[6]

            # Process point
            if confidence > 0.7 and distance > 0.0:
                # Point is valid, use it
                pass

def main(args=None):
    rclpy.init(args=args)
    subscriber = PointCloudSubscriber()
    rclpy.spin(subscriber)
    rclpy.shutdown()

if __name__ == '__main__':
```

```
main()
```

See the full example in `examples/ros2/subscriber_example.cpp` for additional patterns including statistics collection and object detection.

7.3 Camera Image Subscriber

Subscribe to camera images in C++:

```
#include <rclcpp/rclcpp.hpp>
#include <sensor_msgs/msg/image.hpp>

class CameraSubscriber : public rclcpp::Node
{
public:
    CameraSubscriber() : Node("camera_subscriber")
    {
        subscription_ = create_subscription<sensor_msgs::msg::Image>(
            "/xavia_camera_node/image_raw",
            rclcpp::SensorDataQoS(),
            std::bind(&CameraSubscriber::imageCallback, this, std::placeholders::_1)
        );

        RCLCPP_INFO(get_logger(), "Camera subscriber started");
    }

private:
    void imageCallback(const sensor_msgs::msg::Image::SharedPtr msg)
    {
        static uint64_t frame_count = 0;
        frame_count++;

        // Log every 20th frame (once per second)
        if (frame_count % 20 == 0)
        {
            RCLCPP_INFO(get_logger(),
                "Received frame %lu | Encoding: %s | Resolution: %u x %u",
                frame_count,
                msg->encoding.c_str(),
                msg->width,
                msg->height);
        }

        // Access image data based on encoding
        const uint8_t* pixel_data = msg->data.data();
        uint32_t width = msg->width;
        uint32_t height = msg->height;
        uint32_t row_stride = msg->step; // Bytes per row (may include padding)

        // Example: Process based on pixel format
        if (msg->encoding == "rgb8")
        {
            // 8-bit RGB: 3 bytes per pixel
            for (uint32_t y = 0; y < height; ++y)
            {
                const uint8_t* row_ptr = pixel_data + (y * row_stride);
```

```
        for (uint32_t x = 0; x < width; ++x)
        {
            uint8_t r = row_ptr[x * 3 + 0];
            uint8_t g = row_ptr[x * 3 + 1];
            uint8_t b = row_ptr[x * 3 + 2];
            // Process RGB pixel
        }
    }
}
else if (msg->encoding == "bayer_rggb8")
{
    // Raw Bayer pattern: 1 byte per pixel, requires demosaicing
    for (uint32_t y = 0; y < height; ++y)
    {
        const uint8_t* row_ptr = pixel_data + (y * row_stride);
        for (uint32_t x = 0; x < width; ++x)
        {
            uint8_t bayer_value = row_ptr[x];
            // Apply demosaicing algorithm for color reconstruction
        }
    }
}
else if (msg->encoding == "bayer_rggb12")
{
    // 12-bit Bayer: 2 bytes per pixel (16-bit storage)
    for (uint32_t y = 0; y < height; ++y)
    {
        const uint16_t* row_ptr =
            reinterpret_cast<const uint16_t*>(pixel_data + (y * row_stride));
        for (uint32_t x = 0; x < width; ++x)
        {
            uint16_t bayer_value = row_ptr[x] & 0xFFFF; // Extract 12 bits
            // Apply demosaicing algorithm for color reconstruction
        }
    }
}
}

rclcpp::Subscription<sensor_msgs::msg::Image>::SharedPtr subscription_;
};

int main(int argc, char** argv)
{
    rclcpp::init(argc, argv);
    auto node = std::make_shared<CameraSubscriber>();
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}
```

8 Multiple Sensors

Multiple lidar and camera nodes can run simultaneously on the same computer, each with independent network connections. Configure each with different namespaces or node names to avoid conflicts:

```
ros2 run xavia_ros2_driver xavia_lidar_node --ros-args \
  -r __ns:=/sensor1 \
  -p sensor_ip:=10.10.100.11 \
  -p frame_id:=lidar_front
```

```
ros2 run xavia_ros2_driver xavia_lidar_node --ros-args \
  -r __ns:=/sensor2 \
  -p sensor_ip:=10.10.101.11 \
  -p frame_id:=lidar_rear
```

This creates topics `/sensor1/xavia_lidar_node/points` and `/sensor2/xavia_lidar_node/points`, allowing independent operation.

It is best practice to assign unique `frame_id` values to each sensor (e.g., `lidar_front`, `lidar_rear`). This ensures each sensor has a distinct identifier in the ROS2 transform tree, allowing proper frame transformations in multi-sensor systems.

Similarly, multiple camera's can be used as well.

9 Troubleshooting

9.1 Cannot Connect to Sensor

Symptoms: Connection timeout, "Failed to activate" messages

Solutions:

1. Verify network connectivity to the sensor: `ping 10.10.100.11`
2. Ensure the sensor is powered on and network-accessible
3. Check firewall settings allow TCP port 4000
4. Confirm your computer is on the correct subnet (10.10.100.0/24 by default)
5. Verify the sensor IP and port match the node parameters

9.2 No Data Published

Symptoms: Topics exist but no messages arrive, `ros2 topic hz` shows 0 Hz

Solutions:

1. Check the node lifecycle state: `ros2 lifecycle get /xavia_lidar_node`
2. If not in the Active state, check diagnostics topic for error messages: `ros2 topic echo /xavia_lidar_node/diagnostics`
3. Try manually starting data acquisition: `ros2 service call /xavia_lidar_node/start_sensor std_srvs/srv/Trigger`
4. Verify the sensor is responding with a direct network test

9.3 'xavia_lidar_node' not found

Symptoms: When launching the lidar (or camera node) a message like this appears:

[launch]: Caught exception in launch (see debug for traceback): executable 'xavia_lidar_node' no

Solutions:

1. Check if you indeed sourced the `share/xavia_ros2_driver/local_setup.bash` file from where you unpacked the driver.
2. Check if the files in `lib/xavia_ros2_driver/` directory have execution permissions.

9.4 High CPU or Memory Usage

Solutions:

1. Reduce the `buffer_size` parameter to use less memory and reduce latency
2. Throttle message publishing rate by filtering in your subscriber
3. Ensure no other applications are using significant bandwidth on the network
4. Consider subscribing only to the specific point cloud fields you need

9.5 Message Loss or Buffer Overflow

Symptoms: Warnings about dropped frames, incomplete point clouds

Causes: The subscriber cannot keep up with the node's publication rate, or network bandwidth is saturated.

Solutions:

1. Increase `buffer_size` to allow temporary processing delays (uses more memory)
2. Optimize subscriber callback execution time to complete processing quickly
3. Reduce other network traffic on the connection to the sensor
4. Ensure the network has jumbo frames enabled for higher throughput
5. If using frame mode (`points_frame`), consider reducing the number of buffered lines through configuration

9.6 Network Performance

The driver requires dedicated network bandwidth. For optimal performance:

- **Gigabit network required:** Camera and lidar data combined can approach gigabit saturation
- **Jumbo frames:** Enable jumbo frames (9000 byte MTU) on both the sensor and computer network interfaces for reduced packet overhead
- **Dedicated connection:** Connect the sensor to a dedicated network interface if possible to avoid contention with other traffic
- **Processing time:** In the subscriber, keep callback execution time under 50 milliseconds. If processing is more intensive, offload work to a separate thread queue to avoid blocking data arrival
- **Large buffer risk:** While increasing `buffer_size` prevents frame loss, it increases latency and memory usage. Balance buffering against application real-time requirements