Robot Drivers Documentation

Authors: Ban Bogdan & Man Darius

Coordinator: Groza Sergiu

Contents

BNO055 IMU driver

Sensor description

A IMU(Inertial Measurement Unit) sensor measures angular rate, force and sometimes magnetic field. In this project, we use a BNO055 sensor that can collect data presented in the following table:

Name	Abbreviation	Description
Acceleration	ACC	Acceleration data along the 3-axis. (m/s^2)
Magnetic field	MAG	Magnetometer – measures magnetic field and provides poles orientation. (micro Tesla)
Gyroscope	GYR	measures/maintains orientation and angular velocity along the 3-axis. (deg/s)
Euler	EUL	Orientation in space. (X,Y,Z) = (Heading, Roll, Pitch) (deg)
Quaternion	QUA	Orientation in space. (quaternion units)
Linear Acceleration	LIA	Linear acceleration of the sensor along the 3-axis. (m/s^2)
Gravity	GRV	Measures the gravity along the 3-axis.
Temperature	TEMP	Measures temperature in Celsius degrees.

Table 1.1: Sensor data

Parameters and functions

Class “Bno055_IMU” represents a ROS node that opens a connection to the sensor (UART/I2C), collects the data and then, closes the connection. It inherits from Sensor abstract class and takes the parameters from the configuration file. The “Sensor” class contains a “Communication*” object which is the connection to the sensor and a message field that represents the future ROS message that will be published on the topic.

UART communication is based on sending requests and receiving responses. These 2 are represented as byte arrays. For example, to read the “CHIP ID” value (which is A0), the following array should be written to the channel: {0xAA, 0x01,0x00,0x01}. The values from the array represent the start byte (always “AA”, the operation: “read”, register address respectively the number of bytes to be read).

I2C communication is done almost like UART. When we want to read/write data from/to the channel, we only write the slave address (usually 0x28 but depends on the system), the register address and, in case it is a write request, the data to be written. Before running the project, the user has to setup the parameters in “/config/params.yaml” file.

The diagram of the class in presented in Figure 1.1:

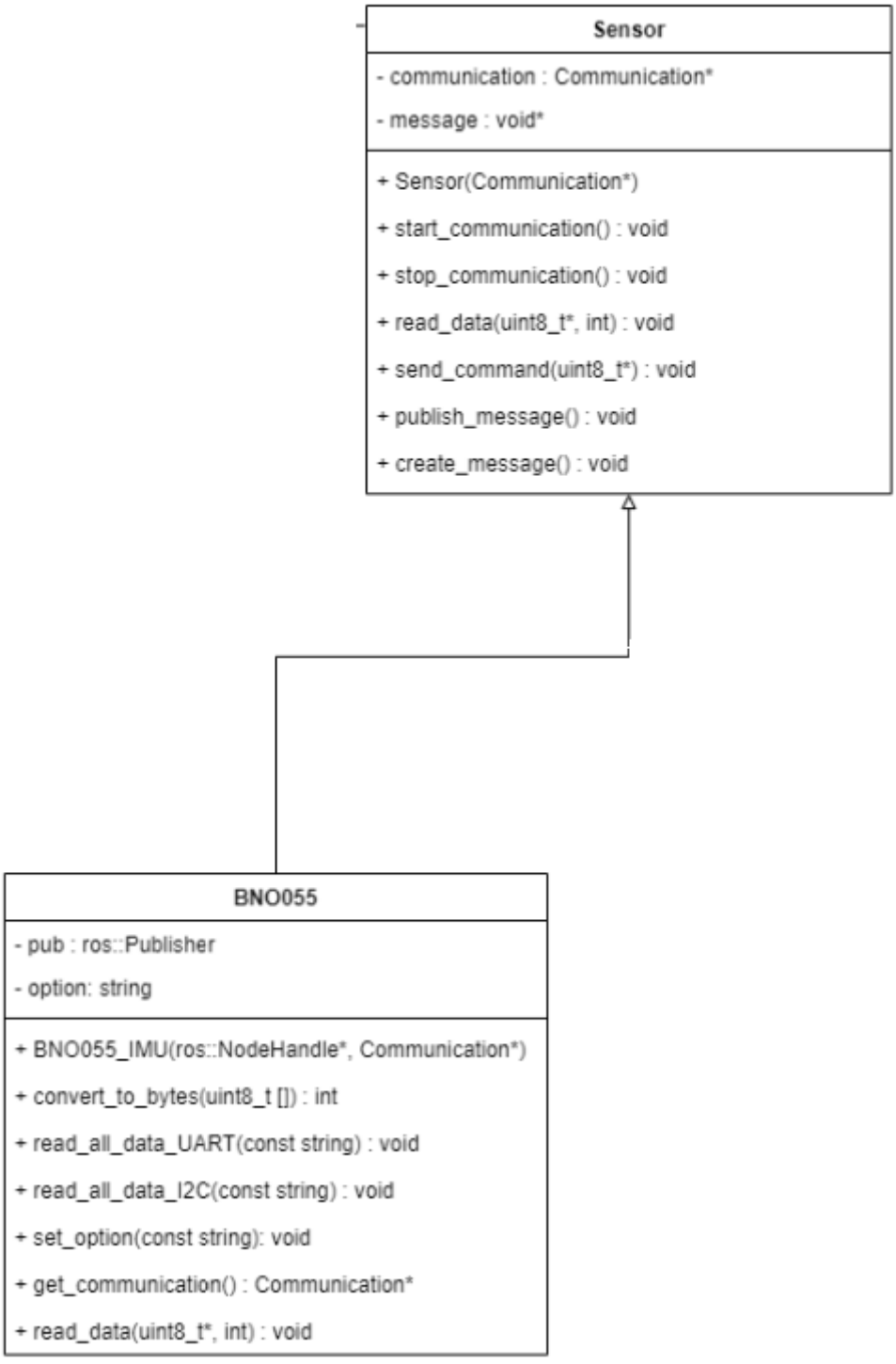


Figure 1.1: Section from the project diagram

The functions implemented inside BNO055 will be presented in the Table 1.2:

Function name	Input	Output	Description
convert_to_bytes(p)	p = vector of 2 bytes	Int	Converts bytes to int (byte[1] is MSB, byte[0] is LSB)
read_all_data_UART(p)	p = name of the data	void	Gets the data from the sensor using UART
read_all_data_I2C(p)	p = name of the data	void	Gets the data from the sensor using I2C
read_data(p1,p2)	p2 = size of the array	p1 = the array read	Calls one of the 2 functions above depending on option
set_option(p)	P = name of the option(UART/I2C)	Void	Sets the option (UART / I2C), so "read_data" knows which function to call
Get_communication()	None	"Communication" object	Getter for communication

Table 1.2: Functions from BNO055

How to use

The ROS package of the project contains a launch file that starts the node. So, to run the project, type the following command into the terminal:

```
$ roslaunch bno055_driver bno055_start.launch
```

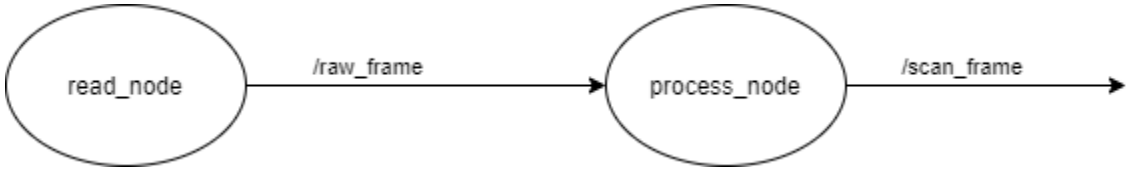
Before starting the node, the parameters inside "params.yaml" file should be set accordingly. An example is presented in Figure 1.2:

```
# 0 for UART & 1 for I2C
communication_type: 1

uart_port_name: "/dev/ttyS0"
baudrate: 115200

i2c_port_name: "/dev/i2c-2"
address: 0x50
```

Figure 1.2: Example of params.yaml

Lidar driver

The lidar driver is composed of two nodes. One of the nodes is used to turn on/off the data intake from the lidar and process it into frames that will be later used by the second node to create a ROS scan message. Each node will create an object that will be used to process the data.



Lidar read class

The class will inherit the Sensor interface and besides the inherited methods new ones were added in order to process the data from the sensor.

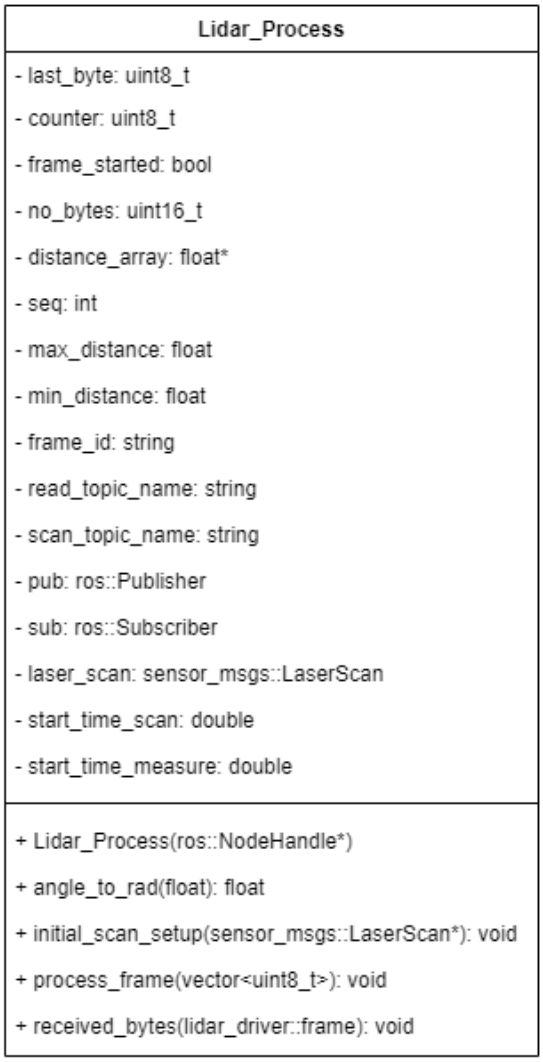
The set of methods *lidar_start_scan*, *ldiar_stop_scan*, *lidar_reset* will send commands to the lidar to start the data flow, to stop the data flow and to soft reset itself.

The *process_bytes* method will read each byte given by the lidar and arrange them in frames that will be published on a ROS topic for the second node to process them.

Lidar process class

All the frames of the published by the *read_node* will be taken by the *process_node* and create a ROS scan message.

The *received_bytes* method is the callback for the ROS subscriber and calls the *process_frame* method with the received frame to be processed.



The *precessed_frame* method will compute the distances and angle of them using the formulas mentioned in the Ydlidar documentation. Whenever a new frame that indicates a start of a point cloud scan is received the previous scan will be published on a topic and a new scan is created.

Configuration

The driver can be configured in two ways, first the yaml can be modified for quick changes, second the header file "*driver_config.h*" contains low level variables that can provide code changes.

The yaml config file

Parameter name	Default value	Description
raw_topic	/raw_frame	The name of the topic on which the raw frames read from the lidar are publish
port_name	/dev/ydlidar	The port to which the lidar is connected (in this case an alias exists)

baurate	128000	The baud rate at which the port needs to be opened to be able to read the data (for the ydlidar)
scan_topic	/scan_frame	The topic name on which the ROS scan message is published
frequency_read	5000 (Hz)	The frequency with which the data is read from the lidar
max_angle	255 (degree)	The max angle of the scan
min_angle	135 (degree)	The min angle of the scan
max_distance	3 (m)	The maximum distance in the scan
min_distance	0.13 (m)	The minimum distance in the scan
frame_id	laser_frame	The frame id used for visualization in rviz

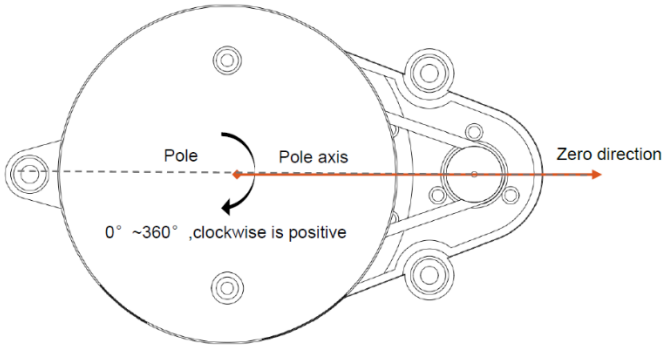


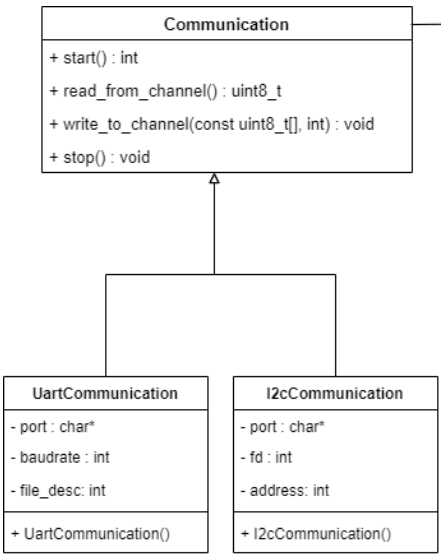
Figure 1 Rotation of the lidar with the max and min angle

Header file "driver_config.h"

Parameter name	Default values	Description
COMMAND_START_BYTE	0xA5	Command's bytes used to communicate with the lidar. These parameters should NOT be changed.
COMMAND_START_SCAN_BYTE	0x60	

COMMAND_STOP_SCAN_BYTE	0x65	
COMMAND_RESET_BYTE	0x80	
ANGLE_INCREMENT	0.615	The different in degrees between two measured distances
ANGLE_INCREMENT_TOLERANCE	0.5	Used to check if the correct number of distance measurements is correct
MAX_FRMAE_LENGTH	90	The maximum number of distance measurements that can be received
TOPIC_BUFFER_SIZE	100	The buffer for the publisher and subscriber
PI	3.1415	Pi constant
MAX_NUMBER_DISTANCES	620	Maximum of distances that can be computed

Uart communication



The UartCommunication class will open a serial communication on a specific port with a specific baud rate.

UartCommunication::start

The method will return a file descriptor with the following settings:

1. A custom baud rate
2. No parity bit
3. 1 bit stop
4. No hardware control
5. Immediate return of the read data with no delay
6. No s/w control
7. Canonical input off
8. No echo

All the attribution of the port are made with the *termios* library.

UartCommunication::read_from_channel

The method read_from_channel will return one byte read from the port using the Linux build in function *read*. If the read failed -1 will be returned.

UartCommunication::wrie_to_channel

If the received buffer has some content the received size is bigger that 0 the buffer will be written to the port using the Linux *write* function.

UartCommunication::stop

The method will close the port if it was opened.