Interreg Sudoe
Co-funded by the European Union
ROBOTA-SUDOE

Real-Time Acquisition and ROS2 Integration of BOTA Systems EtherCAT Force/Torque Sensor

Alexis Babut

Centimfe
Centro Tecnológico da Indústria de Moldes, Ferramentas Especiais e Plásticos
UNIVERSIDADE BEIRA INTERIOR
Cerfundão
aiju
INDUSTRIA JUEMA AUXILIAR S.L.
USC UNIVERSIDADE DE SANTIAGO DE COMPOSTELA
CTC centro tecnolóxico da carne
CLERMONT AUVERGNE INP
CIMES Creating Integrated MEchanical Systems Auvergne-Rhône-Alpes

Foreword

This document presents the integration of Bota systems 6-axis force/torque sensors within a collaborative robotic setup. It covers both the hardware configuration and the software components required for real-time data acquisition using the EtherCAT protocol. The proposed solution is designed to support responsive control strategies in ROS2-based robotic systems.

Thesis Subject

Design and control of an operator-assistance system integrated into a cobot, for the manipulation of soft bodies.

Supervisors of the PhD Project

Thesis Supervisor:	Belhassen Chedli Bouzgarrou
Co-supervisor:	Laurent Sabourin
Co-advisor:	Nicolas Bouton

Date:	June 2025
Laboratory:	Institut Pascal
Project:	ROBOTA-SUDOE - S1/1.1/P0125

Table of Contents

Chapter I
Introduction

Accurate force and torque measurement is essential in robotics, particularly in collaborative applications involving physical interaction with the environment. Six-axis force/torque (F/T) sensors provide real-time information about external loads acting on the robot's end-effector, enabling advanced control strategies that can adapt to dynamic contact conditions.

Bota systems [1] develops high-performance six-axis digital F/T sensors adapted for robotics and automation. These sensors are compact, lightweight, and offer excellent measurement precision, making them suitable for both industrial deployments and research environments. They support several communication protocols, among which EtherCAT stands out for its real-time capabilities.

In this work, we employ Bota systems sensors configured for EtherCAT communication. EtherCAT is a widely used industrial Ethernet protocol that enables deterministic, high-bandwidth, and low-latency data exchange, key properties for integrating sensors into time-critical robotic control loops.

This document presents a real-time software interface for acquiring data from Bota systems EtherCAT F/T sensors and integrating it within a ROS2-based robotic control framework. The proposed solution ensures consistent and timely data delivery to support responsive and robust control strategies in collaborative robotics.

Chapter II
Physical Integration of F/T Sensors

This chapter details the mechanical and electrical integration of the Bota systems 6-axis F/T sensors used within the robotic system. Two sensor models are employed: the *SensONE T-15* [2] and the *Rokubi* [3]. Both communicate with the control system using the EtherCAT protocol, which is a widely adopted real-time industrial Ethernet standard. It enables deterministic and low-latency data exchange, which is essential for closed-loop robotic control.

1. Sensors Mounting

The **SensONE T-15** sensor is designed for mounting directly on the tool flange of medium-sized collaborative robots. Its compact form factor and standardised interface allow straightforward and secure installation, ensuring accurate force and torque measurements at the robot's end-effector.

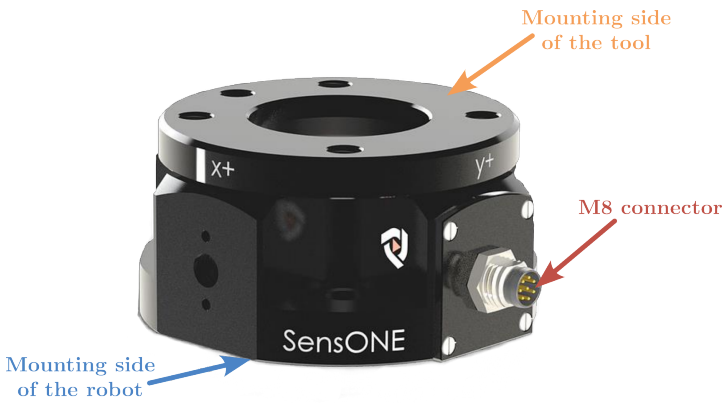

Figure II.1: Schematic of the SensONE F/T sensor.

The **Rokubi** sensor offers flexible installation options and can be attached either directly to the tool flange of smaller cobots or on a tool. This arrangement supports various tooling configurations and measurement requirements.

Figure II.2: Schematic of the Rokubi F/T sensor.

2. Coordinate Systems

Each Bota systems sensor includes internal coordinate frames used to express force, torque, and inertial measurements. The coordinate systems associated with force and torque data are defined by the manufacturer and remain fixed relative to the sensor housing. Figures II.3 and II.4 illustrate the F/T coordinate frames of the SensONE and Rokubi sensors, respectively.

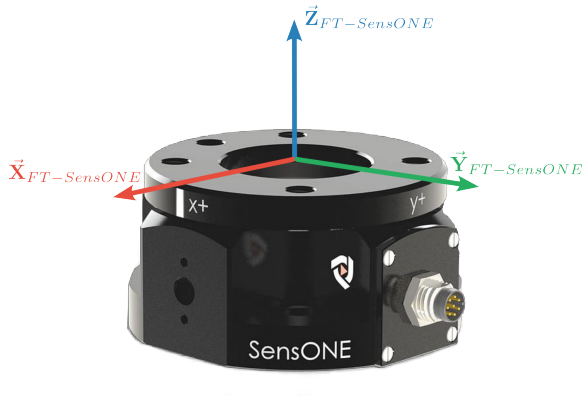

Figure II.3: F/T coordinate frame for the SensONE sensor.

Figure II.4: F/T coordinate frame for the Rokubi sensor.

In addition to force and torque sensing, both sensors are equipped with an integrated Inertial Measurement Unit (IMU), which measures linear acceleration and angular velocity. The IMU frames are fixed relative to the F/T measurement frames and follow the same orientation conventions. Figures II.5 and II.6 show the combined F/T and IMU coordinate systems for each sensor.

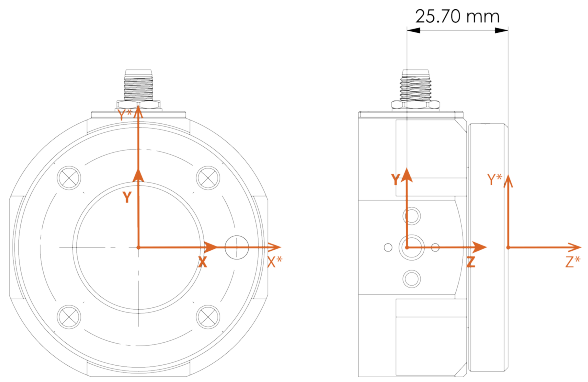

Figure II.5: SensONE: IMU and F/T coordinate systems.

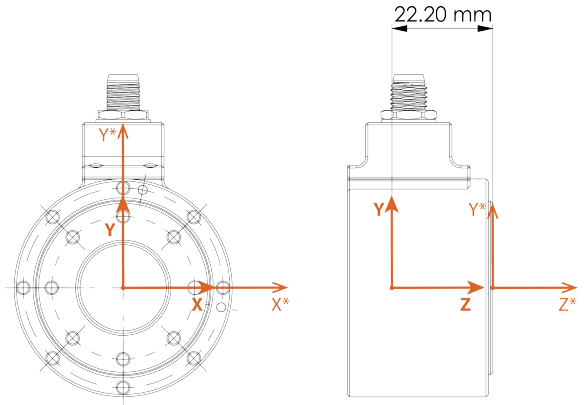

Figure II.6: Rokubi: IMU and F/T coordinate systems.

3. Electrical Connection

Both Bota systems sensors communicate using the EtherCAT protocol. Each sensor is connected using an **M8-to-RJ45** cable, which links it to a **Power over Ethernet (PoE) injector**.

The M8 connector carries both power and EtherCAT communication signals. The pin configuration of the M8 connector is shown in Figure II.7.

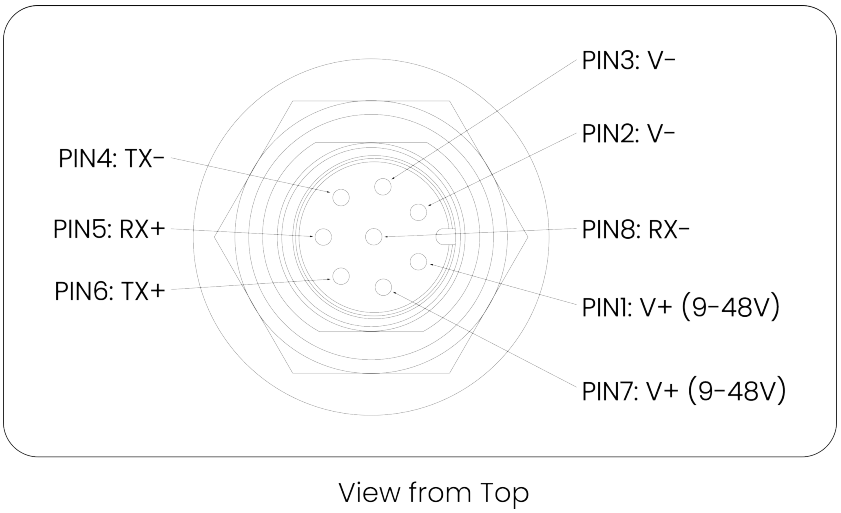

Figure II.7: Pin configuration of the M8 connector used for EtherCAT communication and power delivery.

The **PoE** injector itself is powered by a **DC power supply** with an input voltage range of 9 to 48. The second RJ45 port, labelled **LAN**, is connected to the host computer's network interface using a standard Ethernet cable.

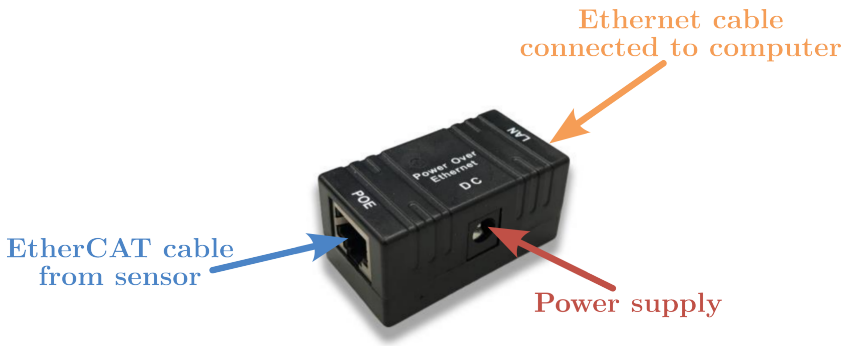

Figure II.8: Power over Ethernet (PoE) injector used to supply power and establish network communication with the sensor.

This configuration enables the sensor to receive power while maintaining real-time EtherCAT communication with the control computer. This setup supports high-frequency data acquisition.

Chapter III
Sensor Initialisation and Configuration

This chapter outlines the steps required to initialise and configure the EtherCAT-based F/T sensor system to ensure its robust integration within a real-time robotic control framework. The configuration process encompasses detecting the EtherCAT device, performing minimal communication tests, and applying the necessary sensor-specific parameters, such as offsets and filtering options.

1. EtherCAT Device Detection and Interface Setup

EtherCAT devices are connected to the host system through a dedicated Ethernet interface. Detection and initial configuration are performed using the `pysoem` Python library [4], which provides a minimal yet effective interface for EtherCAT master communication. Listing III.1 presents the Python script, taken from [5], used to enumerate all available network adapters.

```python
"""Prints name and description of available network adapters."""

import pysoem

adapters = pysoem.find_adapters()

for i, adapter in enumerate(adapters):
    print('Adapter {}'.format(i))
    print('  {}'.format(adapter.name))
    print('  {}'.format(adapter.desc))
```

Listing III.1: *Python script:* Detecting available network adapters.

Before running this script, the `pysoem` package must be installed. The commands in Listing III.2 first install the package using `pip`, then execute the detection script. Elevated privileges (e.g., `sudo`) are necessary to grant low-level access to the network interface, which is required for EtherCAT master operations.

```
sudo pip install pysoem
sudo python3 EtherCAT_interface_finder/find_ethernet_adapters.py
```

Listing III.2: *Bash script:* Installing PySOEM and running the detection script.

When run, this script lists all network adapters present on the host system, displaying their names and descriptions. Identifying the correct adapter (typically a dedicated Ethernet port such as `ens5`) is essential for establishing communication with EtherCAT devices.

After selecting the appropriate network interface, connectivity to the first slave device on the EtherCAT bus can be verified by reading its EEPROM data. The EEPROM (Electrically Erasable Programmable Read-Only Memory) is a non-volatile memory embedded in the slave device that stores crucial device-specific information such as vendor and product identifiers.

Listing III.3 shows a Python script, taken from [5], that opens the EtherCAT master interface, initialises the network, and reads the EEPROM content of the first slave device.

```python
"""Prints name and description of available network adapters."""

import sys
import pysoem


def read_eeprom_of_first_slave(ifname):
    master = pysoem.Master()

    master.open(ifname)

    if master.config_init() > 0:

        first_slave = master.slaves[0]

        for i in range(0, 0x80, 2):
            print('{:04x}:'.format(i), end='')
            print('|'.join('{:02x}'.format(x) for x in first_slave.eeprom_read(i)))

    else:
        print('no slave available')

    master.close()


if __name__ == '__main__':

    print('script started')

    if len(sys.argv) > 1:
        read_eeprom_of_first_slave(sys.argv[1])
    else:
        print('give ifname as script argument')
```

Listing III.3: *Python script:* Reading EEPROM data from the first EtherCAT slave device.

The Bash command III.4 executes the Python script, passing the selected network adapter as an argument to read and display the EEPROM contents of the first EtherCAT slave device on the bus.

```
sudo python3 EtherCAT_interface_finder/read_eeprom.py <adapter>
```

Listing III.4: *Bash script:* Reading EEPROM data from the first slave device.

2. Sensor Initialisation and Minimal Communication

Following successful detection, the EtherCAT slave device can be initialised and tested to ensure that the communication pipeline is operational. A minimal test script is provided by [5], which prints live sensor data to the terminal, confirming correct data transmission.

```
sudo python3 simple_test_scripts/bota_ethercat_minimal_example.py <adapter>
```

Listing III.5: *Bash script:* Run minimal EtherCAT communication test.

This minimal communication script is particularly useful for isolating issues related to physical connectivity, device configuration, or EtherCAT master initialisation.

3. Reading and Writing Configuration Parameters

Sensor parameters such as filter settings, offset values, and IMU range configurations can be read and written on the device by accessing the object dictionary using Service Data Object (SDO) frames.

Object	Sub index	Name	Description
0x2000	0	Calibration	Used to read the device specific calibration values. (Gains, offsets, etc.)
0x6000	0	Sensor Data	Read the Force/Torque, IMU, and other sensor data.
0x8000	0	Force/Torque Offset	Write the Force/Torque offset values.
0x8001	0	Acceleration Offset	Read the Linear Acceleration Offset values.
0x8002	0	Angular Velocities Offset	Read the Angular Velocities Offset values.
0x8003	0	Force/Torque Range	Read the Force/Torque Sensor range.
0x8004	0	Acceleration Range	Read and Write the Linear Acceleration range. (0: ±2g, 1: ±4g, 2: ±8g, 3: ±16g)
0x8005	0	Angular Velocities Range	Read and Write the Angular Velocities range. (0: ±250°/s, 1: ±500°/s, 2: ±1000°/s, 3: ±2000°/s)
0x8006	01	SINC Length	Sets the SINC filtering. (Default: 0x0040)
0x8006	02	FIR disable	Disables FIR filter when bit is HIGH. (Default: 0x01)
0x8006	03	FAST enable	Enables spike detection filter when bit is HIGH. (Default: 0x00)
0x8006	04	CHOP enable	Enables CHOP filtering when bit is HIGH. (Default: 0x00)
0x8007	0	Acceleration Filter	Read and Write the Linear Acceleration filtering.
0x8008	0	Angular Velocities Filter	Read and Write the Angular Velocities filtering.
0x8010	01	Force/Torque Calibration Matrix Active	Activate Calibration Matrix on device. (0 = Raw data, 1 = N/Nm data)
0x8010	02	Temperature Compensation Active	Enable temperature compensation on device. (0 = Not activated, 1 = Activated)
0x8010	03	IMU Active	Activate IMU Readings. (0 = Not activated, 1 = Activated)
0x8011	0	Sampling Rate	Read the sampling rate of the sensor with the current configuration. (Default: 0x0320)
0x8030	01	Command	Control codes for Reading/Writing. (01: save parameters after power-down)
0x8030	02	Status	Read error code if command fails.

Table III.1: SDO object dictionary.

The Python script presented below demonstrates how to read and update several key parameters of the Bota Systems force/torque sensor using SDO messages.

```python
import struct
import time
from collections import namedtuple
import pysoem
import sys
import ctypes

class Sensor_Configuration_SDO_Reader:

    BOTA_VENDOR_ID = 0xB07A
    BOTA_PRODUCT_CODE = 0x00000001
    BOTA_SENSOR_NAME = 'BFT-SENS-ECAT-M8'

    def __init__(self, ifname):
        self._ifname = ifname

        # Initialize variables
        self.sensor_input_as_bytes = None
        self.time_step = 1.0
        self.receive_timeout_us = 2000

        # Initialise EtherCAT master and slaves
        self._master = pysoem.Master()
        SlaveSet = namedtuple('SlaveSet', 'slave_name product_code config_func')
        self._expected_slave_layout = {0: SlaveSet(self.BOTA_SENSOR_NAME, self.BOTA_PRODUCT_CODE, self.bota_sensor_setup)}

        try:
            self._master.open(self._ifname)
        except Exception as e:
            print(f"Failed to open EtherCAT interface '{self._ifname}'.")
            raise

        if not self._master.config_init() > 0:
            print('No EtherCAT slave found.')
            self._master.close()
            raise

        for i, slave in enumerate(self._master.slaves):
            if not ((slave.man == self.BOTA_VENDOR_ID) and (slave.id == self._expected_slave_layout[i].product_code)):
                print('Unexpected slave layout.')
                self._master.close()
                raise
            slave.config_func = self._expected_slave_layout[i].config_func
        self._master.config_map()

        if self._master.state_check(pysoem.SAFEOP_STATE, 50000) != pysoem.SAFEOP_STATE:
            print('Not all slaves reached SAFEOP state.')
            self._master.close()
            raise
        self._master.state = pysoem.OP_STATE
        self._master.write_state()

        self._master.state_check(pysoem.OP_STATE, 50000)
        if self._master.state_check(pysoem.OP_STATE, 50000) != pysoem.OP_STATE:
            print('Not all slaves reached OP state.')
            self._master.close()
            raise

    def bota_sensor_setup(self, slave_pos):
        """Configure Bota Systems slave device."""
        slave = self._master.slaves[slave_pos]
        ## Read sensor configuration
        print("\n--- Reading Default SDO Values ---")
        sdo_map = [
            (0x2000, 0, 'Calibration'),
            (0x6000, 0, 'Sensor Data'),
```

```
            (0x7000, 0, 'Digital Output'),
            (0x8000, 0, 'Force/Torque Offset'),
            (0x8001, 0, 'Acceleration Offset'),
            (0x8002, 0, 'Angular Velocities Offset'),
            (0x8003, 0, 'Force/Torque Range'),
            (0x8004, 0, 'Acceleration Range'),
            (0x8005, 0, 'Angular Velocities Range'),
            (0x8006, 0, 'Force/Torque Filter'),
            (0x8006, 1, 'SINC Length'),
            (0x8006, 2, 'FIR disable'),
            (0x8006, 3, 'FAST enable'),
            (0x8006, 4, 'CHOP enable'),
            (0x8007, 0, 'Acceleration Filter'),
            (0x8008, 0, 'Angular Velocities Filter'),
            (0x8010, 0, 'Device configuration'),
            (0x8010, 1, 'Calibration Matrix Active'),
            (0x8010, 2, 'Temperature Compensation'),
            (0x8010, 3, 'IMU Active'),
            (0x8010, 4, 'Coordinate System Cong. Active'),
            (0x8010, 5, 'Inertia and Gravity Comp. Active'),
            (0x8010, 6, 'Orientation Estimation Active'),
            (0x8011, 0, 'Sampling Rate'),
            (0x8030, 0, 'Control'),
            (0x8030, 1, 'Command'),
            (0x8030, 2, 'Status'),
        ]
        for index, subindex, name in sdo_map:
            try:
                raw_data = slave.sdo_read(index, subindex)
                value_hex = raw_data.hex()
                print(f"0x{index:04X}, Sub {subindex:02d} - {name}: 0x{value_hex.upper()}")
            except Exception as e:
                print(f"0x{index:04X}, Sub {subindex:02d} - {name}: ERROR reading ({str(e)})")

        ## Set sensor configuration
        # calibration matrix active
        slave.sdo_write(0x8010, 1, bytes(ctypes.c_uint8(1)))
        # temperature compensation
        slave.sdo_write(0x8010, 2, bytes(ctypes.c_uint8(0)))
        # IMU active
        slave.sdo_write(0x8010, 3, bytes(ctypes.c_uint8(1)))
        # FIR disable
        slave.sdo_write(0x8006, 2, bytes(ctypes.c_uint8(1)))
        # FAST enable
        slave.sdo_write(0x8006, 3, bytes(ctypes.c_uint8(0)))
        # CHOP enable
        slave.sdo_write(0x8006, 4, bytes(ctypes.c_uint8(0)))
        # Sinc filter size
        slave.sdo_write(0x8006, 1, bytes(ctypes.c_uint16(self.SINC_LENGTH)))
        ## Get sampling rate
        sampling_rate = struct.unpack('h', slave.sdo_read(0x8011, 0))[0]
        print(f"Sensor sampling rate: {sampling_rate} Hz.")
        if sampling_rate > 0:
            self.time_step = 1.0 / float(sampling_rate)
            self.receive_timeout_us = int(self.time_step * 1e6)
            print(f"Time step calculated: {self.time_step} seconds.")
            print(f"Time timeout: {self.receive_timeout_us} us.")
```

Listing III.6: *Python script:* Reading and updating configuration parameters of the Bota Systems EtherCAT sensor through SDO communication using `pysoem`.

```
sudo python3 sensor_configuration_SDO_reader.py <adapter>
```

Listing III.7: *Bash script:* Launch the sensor SDO configuration reader script on the specified network interface.

4. Sensor Output Data Mapping

The sensor periodically transmits measurement data using Process Data Object (PDO) frames, which are designed for real-time, cyclic communication within the EtherCAT network. Each PDO frame contains a fixed set of data fields that represent the current sensor readings and status information. Table III.2 summarises the layout of the transmit `TxPDO1` mapping frame, detailing each data element.

Name	Index	Sub-index	Type	Bit size	Offset bytes
Status	0x6000	0x01	USINT	8	0
Warnings Errors Fatals	0x6000	0x02	UDINT	32	1
Force x	0x6000	0x04	REAL	32	5
Force y	0x6000	0x05	REAL	32	9
Force z	0x6000	0x06	REAL	32	13
Torque x	0x6000	0x07	REAL	32	17
Torque y	0x6000	0x08	REAL	32	21
Torque z	0x6000	0x09	REAL	32	25
Force Torque Saturated	0x6000	0x0a	UINT	16	29
Acceleration x	0x6000	0x0b	REAL	32	31
Acceleration y	0x6000	0x0c	REAL	32	35
Acceleration z	0x6000	0x0d	REAL	32	39
Acceleration Saturated	0x6000	0x0e	UINT	8	43
Angular Velocities x	0x6000	0x0f	REAL	32	44
Angular Velocities y	0x6000	0x10	REAL	32	48
Angular Velocities z	0x6000	0x11	REAL	32	52
Angular Velocities Saturated	0x6000	0x12	UINT	8	56
Temperature	0x6000	0x13	REAL	32	57

Table III.2: TxPDO1 mapping.

To retrieve the sensor's PDO data in a Python-based EtherCAT master setup, such as with the `pysoem` library [4], a typical procedure involves sending and receiving cyclic process data. The following code extract demonstrates how the application exchanges data with the sensor and accesses the received input bytes, which correspond to the mapped TxPDO fields shown in Table III.2.

```
self._master.send_processdata()  # Exchange process data with the EtherCAT network
self._master.receive_processdata(2000)  # Wait up to 2 ms for incoming data
self.sensor_input_as_bytes = self._master.slaves[0].input  # Access the received input
    data buffer

self.Fx = struct.unpack_from('f', self.sensor_input_as_bytes, 5)[0]
self.Fy = struct.unpack_from('f', self.sensor_input_as_bytes, 9)[0]
self.Fz = struct.unpack_from('f', self.sensor_input_as_bytes, 13)[0]
```

Listing III.8: *Python script:* Reading sensor PDO data using `pysoem`.

The `sensor_input_as_bytes` variable now contains a raw byte stream representing the current state of the sensor, as per the mapping defined in `TxPDO1`. Each field can then be decoded using the appropriate byte offsets and data types as detailed in Table III.2.

Sensor faults and warnings are encoded in a 32-bit status field transmitted within the PDO frame. The bit mask shown in Table III.3 allows decoding of sensor anomalies.

Bit index	Severity	Description	Solution
0	Error	Force/Torque (ADC) saturated	Reduce sensor loading
1	Error	Acceleration saturated	Reduce acceleration or increase acceleration range
2	Error	Angular velocities (gyro) saturated	Reduce angular velocities or increase angular velocities range
3	Error	ADC out of sync	Contact support (repair required)
4	Error	Force Torque measurement exceeding the range	Reduce sensor loading
5	Warning	Overtemperature	Power off the sensor and let it cool down
6	Fatal	Wrong supply voltage	Remove sensor from power supply
7-31	-	Reserved or Not Used	-

Table III.3: Bit mask for sensor data field "Warning Errors Fatals".

5. Filter Configuration

Digital filtering is used to reduce measurement noise while managing signal latency and update frequency.

5.1. Force/Torque Filter Configuration

The EtherCAT Force/Torque sensor integrates multiple filtering stages, each affecting the trade-off between signal fidelity, noise suppression, and system responsiveness:

- **SINC filter**: A digital low-pass filter that attenuates high-frequency noise through averaging of consecutive samples. This corresponds to a rectangular impulse response, producing a sinc-shaped frequency response in the frequency domain. The filter length is configurable; longer lengths provide stronger attenuation of high-frequency components but proportionally reduce the sensor's update rate due to increased averaging duration [6].
- **FIR filter**: A finite impulse response (FIR) filter operates by computing a weighted sum of a finite number of past input samples: $y[n] = \sum_{i=0}^{N} b_i \cdot x[n-i]$, where b_i are the filter coefficients. FIR filters are inherently stable, can be designed with linear phase, and do not rely on feedback loops. When enabled, the FIR filter provides additional signal smoothing with a low cut-off frequency. However, it introduces a fixed latency and reduces the maximum output rate due to increased computational demands and buffer requirements [7].
- **FAST filter**: The FAST filter, characterised by a high cut-off frequency, is a special operating mode available only when the FIR filter is active. It dynamically toggles the FIR filter on and off to achieve a compromise between low noise levels (as with FIR enabled) and rapid response time (as with FIR disabled). This mode is not recommended in applications requiring high-frequency dynamic response or strict real-time performance due to potential inconsistency in latency.
- **CHOP filter**: The CHOP filter compensates for systematic offset and thermal drift by modulating the signal to eliminate low-frequency bias. However, in real-time EtherCAT applications, it is generally advised to keep the CHOP filter disabled, as its modulation process can introduce unpredictable latency.

Table III.4 summarises the relationship between the SINC filter length, the resulting sensor update rate, and the effective cut-off frequencies with the FIR filter either enabled or disabled.

SINC length (dec)	Update rate (FIR enabled/disabled)	Cut-off frequency with FIR disabled	Cut-off frequency with FIR enabled
32	1600 Hz	418.5 Hz	62.5 Hz
51	1003 Hz	265.5 Hz	39.5 Hz
64	800 Hz	209.5 Hz	31.5 Hz
128	400 Hz	105.0 Hz	16.0 Hz
205	250 Hz	65.5 Hz	10.0 Hz
256	200 Hz	52.5 Hz	8.0 Hz
341	150 Hz	39.5 Hz	6.0 Hz

Table III.4: Force/Torque filter options.

In the context of our application, the robot's control loop in ROS2 operates at 500 Hz. Therefore, it is desirable for the F/T sensor to provide measurements at a similar or slightly higher rate to ensure timely feedback without introducing unnecessary delay. A suitable configuration would be to set the update rate to 800 Hz by selecting a SINC filter length of 64. In this setup, the FIR filter may be enabled for additional noise reduction, while the FAST and CHOP filters should remain disabled to avoid unpredictable latency and preserve deterministic behaviour.

To achieve this configuration using the `pysoem` library, the following Python code illustrates how to send SDO messages to the EtherCAT F/T sensor to adjust its filter parameters and verify the resulting sampling frequency.

```
BOTA_VENDOR_ID = 0xB07A
BOTA_PRODUCT_CODE = 0x00000001
SINC_LENGTH = 64
# Initialise EtherCAT master and slaves
self._master = pysoem.Master()
SlaveSet = namedtuple('SlaveSet', 'slave_name product_code config_func')
self._expected_slave_layout = {0: SlaveSet('BFT-SENS-ECAT-M8', self.BOTA_PRODUCT_CODE,
    self.bota_sensor_setup)} # Replace the product type 'BFT-SENS-ECAT-M8' with the
    correct one as specified in the product's ESI file.

def bota_sensor_setup(self, slave_pos):
        """Configure Bota Systems slave device."""
        slave = self._master.slaves[slave_pos]
        ## Set force torque filter
        slave.sdo_write(0x8006, 2, bytes(ctypes.c_uint8(1))) # FIR disable
        slave.sdo_write(0x8006, 3, bytes(ctypes.c_uint8(0))) # FAST disable
        slave.sdo_write(0x8006, 4, bytes(ctypes.c_uint8(0))) # CHOP disable
        slave.sdo_write(0x8006, 1, bytes(ctypes.c_uint16(self.SINC_LENGTH))) # Sinc
    filter size
        ## Get sampling rate
        sampling_rate = struct.unpack('h', slave.sdo_read(0x8011, 0))[0]
        print(f"Sensor sampling rate: {sampling_rate} Hz.")
```

Listing III.9: *Python script:* configure FT sensor filters via SDO using `pysoem`.

5.2. IMU Filter Configuration

The IMU integrated in sensors supports several pre-defined filter profiles, each characterised by a distinct bandwidth, output data rate (sampling frequency), delay, and for accelerometers noise density.

- **Bandwidth**: Determines the cut-off frequency of the filter and defines how much high-frequency noise is attenuated. Higher bandwidth allows faster signal response but retains more noise.
- **Delay**: Indicates the latency introduced by the filter due to buffering and processing. Lower delay is advantageous for time-sensitive applications but often results in less noise suppression.
- **Sampling Frequency (Fs)**: The internal update rate of the filtered signal. While the sensor hardware may support higher sampling internally, filtering can reduce the effective output rate to match the selected profile.
- **Noise Density**: For accelerometer channels, this metric reflects the root-mean-square (RMS) noise level per square root of bandwidth and is inversely related to the filter's smoothing capability.

Tables III.5 and III.6 list the available IMU filter profiles for angular velocity and linear acceleration measurements, respectively. These options allow tailoring of the IMU output characteristics to meet specific application requirements.

Filter profile	Bandwidth (Hz)	Delay (ms)	Fs (kHz)
3	184	2.9	1
4	92	3.9	1
5	41	5.9	1
6	20	9.9	1
7	10	17.85	1
8	5	33.48	1
9	3600	0.17	8

Table III.5: IMU filter options – angular velocities.

Filter profile	Bandwidth (Hz)	Delay (ms)	Fs (kHz)	Noise Density (ug/rtHz)
1	460	1.94	1	250
2	184	5.80	1	250
3	92	7.80	1	250
4	41	11.80	1	250
5	20	19.80	1	250
6	10	35.70	1	250
7	5	66.96	1	250
8	460	1.94	1	250

Table III.6: IMU filter options – linear accelerations.

In the context of our robotic cell, the primary goal is to minimise signal latency to preserve the responsiveness of the control system. For this reason, the selected filter configuration prioritises the lowest available delay:

- **Gyroscope filter profile 3**, which provides a bandwidth of 184 Hz with a low delay of 2.9 ms, enabling rapid response with moderate noise filtering;
- **Accelerometer filter profile 1**, which achieves the shortest delay of 1.94 ms while maintaining high bandwidth of 460 Hz and standard noise density, thus maximising responsiveness in fast feedback applications.

Chapter IV
Force/Torque Bias and Gravity Compensation

Force/Torque (F/T) sensors often exhibit bias or offset values, which are non-zero readings in the absence of external forces or moments. These offsets typically arise once the sensor is installed and can be attributed to factors such as mounting preloads, gravitational effects from attached tools, and inertial loads during motion.

1. Bias Estimation

Under static conditions, the sensor offset can be considered primarily static, resulting from gravitational and mechanical preloading. A common practice to compensate for this is to capture the current F/T readings while the sensor is stationary and unloaded. These initial readings are treated as the bias, denoted as $\mathbf{F}_{\text{bias}}$, and are subtracted from subsequent measurements:

$$\mathbf{F}_{\text{corrected}} = \mathbf{F}_{\text{measured}} - \mathbf{F}_{\text{bias}} \tag{IV.1}$$

2. Gravity Compensation

In cases where a tool is attached to the sensor, gravitational effects must also be compensated to isolate externally applied forces. This involves transforming the gravitational vector into the sensor's frame and subtracting the corresponding forces and moments.

Let m denote the tool mass, and $\mathbf{g}_{\text{base}} = [0,\ 0,\ -9.81]^\top$ be the gravity vector in the robot base frame. The orientation of the sensor with respect to the base is given by the rotation matrix ${}^{\text{sensor}}R_{\text{base}}$. The gravity in the sensor frame is given by:

$$\mathbf{g}_{\text{sensor}} = {}^{\text{sensor}}R_{\text{base}}\,\mathbf{g}_{\text{base}} \tag{IV.2}$$

The gravitational force exerted on the sensor is:

$$\mathbf{F}_{\text{gravity}} = m \cdot \mathbf{g}_{\text{sensor}} \tag{IV.3}$$

Assuming the tool's centre of gravity $\mathbf{r}_g$ is known in the sensor frame, the moment due to this gravitational force is:

$$\mathbf{M}_{\text{gravity}} = \mathbf{r}_g \times \mathbf{F}_{\text{gravity}} \tag{IV.4}$$

These are then subtracted from the bias-compensated measurements:

$$\begin{aligned} \mathbf{F}_{\text{final}} &= \mathbf{F}_{\text{corrected}} - \mathbf{F}_{\text{gravity}} \\ \mathbf{M}_{\text{final}} &= \mathbf{M}_{\text{corrected}} - \mathbf{M}_{\text{gravity}} \end{aligned} \tag{IV.5}$$

Chapter V
Real-Time Data Transmission

This chapter presents the design and implementation of a real-time data transmission pipeline between a Bota Systems force/torque sensor and a ROS2 environment. The system consists of a Python-based TCP server that acquires raw sensor data using the `pysoem` library [4], and a ROS2 node acting as a TCP client, which receives, decodes, and republishes this data for use by other components within the robotic control framework.

1. TCP Socket Communication

The communication is based on a standard TCP socket, which establishes a bidirectional link between two processes. The server listens on a specific port, while the client connects to this port and streams the binary data.

As both server and client run on the same machine, the transmission remains local, avoiding external network latency. When executed under a real-time kernel, the end-to-end communication delay is typically in the range of a few microseconds. This allows stable operation at rates up to 800 Hz or higher.

This setup provides a good balance between simplicity and real-time performance. While shared memory or dedicated IPC mechanisms could theoretically offer lower latency, the current TCP socket-based approach remains sufficiently precise for the control requirements, and is fully compatible with standard Python and ROS2 tools. The resulting architecture ensures reliable and synchronised data streaming to other system components.

2. TCP Server for Sensor Data

The TCP server is implemented in Python and uses the `pysoem` library to create an EtherCAT master that communicates with the force/torque sensor, identified as a slave device on the EtherCAT network. At startup, the server initializes and configures the sensor, establishes a TCP connection, and enters a continuous acquisition loop. In each cycle, it reads the raw data frame directly from the EtherCAT interface and immediately sends this byte stream to any connected TCP client.

To ensure a clean shutdown, the server continuously monitors the TCP socket for a shutdown command from the client. Once the command is received, the server stops the acquisition loop, safely deactivates the EtherCAT interface, and closes all network connections.

```
"""
Date: April 2025

Description:
This script implements a TCP server for real-time communication with a Bota Systems EtherCAT force-torque sensor.
It uses the pysoem library to interface with the EtherCAT slave device, read the sensor data, and stream it over a TCP connection.

The script is designed to be launched with two command-line arguments:
    - <ifname>: Name of the network interface connected to the EtherCAT network.
    - <port>: TCP port number to listen on for client connections.

Key Features:
- Detects and configures the Bota Systems EtherCAT slave.
- Applies specific SDO settings for sensor calibration, IMU activation, and filtering.
- Computes the sampling rate from the sensor and adapts the data transmission frequency.
- Sends raw sensor input data over a TCP socket to a connected client.
```

```
- Waits for shutdown commands from the client (sent as JSON with {"action": "shutdown"}).

Classes:
- EtherCAT_FT_Sensor_Logger: Main class handling EtherCAT setup, sensor communication, and TCP server logic.

Typical usage:
-> sudo python3 ethercat_reader.py ens5 BFT-SENS-ECAT-M8 12001
"""

import struct
import ctypes
import time
from collections import namedtuple
import pysoem
import socket
import json
import select
import sys

class EtherCAT_FT_Sensor_Logger:

    BOTA_VENDOR_ID = 0xB07A
    BOTA_PRODUCT_CODE = 0x00000001
    SINC_LENGTH = 64

    def __init__(self, ifname, BOTA_SENSOR_NAME, SERVER_PORT):
        self._ifname = ifname
        self._BOTA_SENSOR_NAME = BOTA_SENSOR_NAME
        self._SERVER_PORT = SERVER_PORT # TCP server port
        self.SERVER_HOST = 'localhost'

        self.time_step = 1.0
        self.receive_timeout_us = 2000

        # Initialise EtherCAT master and slaves
        self._master = pysoem.Master()
        SlaveSet = namedtuple('SlaveSet', 'slave_name product_code config_func')
        self._expected_slave_layout = {0: SlaveSet(self._BOTA_SENSOR_NAME, self.BOTA_PRODUCT_CODE, self.bota_sensor_setup)}

        try:
            self._master.open(self._ifname)
        except Exception as e:
            print(f"Failed to open EtherCAT interface '{self._ifname}'.")
            raise

        if not self._master.config_init() > 0:
            print('No EtherCAT slave found.')
            self._master.close()
            raise

        for i, slave in enumerate(self._master.slaves):
            if not ((slave.man == self.BOTA_VENDOR_ID) and (slave.id == self._expected_slave_layout[i].product_code)):
                print('Unexpected slave layout.')
                self._master.close()
                raise
            slave.config_func = self._expected_slave_layout[i].config_func
        self._master.config_map()

        if self._master.state_check(pysoem.SAFEOP_STATE, 50000) != pysoem.SAFEOP_STATE:
            print('Not all slaves reached SAFEOP state.')
            self._master.close()
            raise
        self._master.state = pysoem.OP_STATE
        self._master.write_state()

        self._master.state_check(pysoem.OP_STATE, 50000)
        if self._master.state_check(pysoem.OP_STATE, 50000) != pysoem.OP_STATE:
```

```
            print('Not all slaves reached OP state.')
            self._master.close()
            raise

        # Initialize variables
        self._shutdown_requested = False
        self.sensor_input_as_bytes = None

        # Create TCP server socket for shutdown communication
        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server_socket.bind((self.SERVER_HOST, self._SERVER_PORT))
        self.server_socket.listen(1)
        print(f"Waiting for client connection on {self.SERVER_HOST}: {self._SERVER_PORT}...")
        self.client_socket, self.client_address = self.server_socket.accept()
        print(f"Client connected from {self.client_address}.")


    def bota_sensor_setup(self, slave_pos):
            """Configure Bota Systems slave device."""
            slave = self._master.slaves[slave_pos]
            ## Set sensor configuration
            slave.sdo_write(0x8010, 1, bytes(ctypes.c_uint8(1))) # Calibration Matrix Active
            slave.sdo_write(0x8010, 2, bytes(ctypes.c_uint8(0))) # Temperature Compensation Active
            slave.sdo_write(0x8010, 3, bytes(ctypes.c_uint8(1))) # IMU Active
            slave.sdo_write(0x8004, 0, bytes(ctypes.c_uint8(0))) # Acceleration Range
            slave.sdo_write(0x8005, 0, bytes(ctypes.c_uint8(0))) # Angular Velocities Range
            slave.sdo_write(0x8007, 0, bytes(ctypes.c_uint8(1))) # Acceleration Filter
            slave.sdo_write(0x8008, 0, bytes(ctypes.c_uint8(3))) # Angular Velocities Filter
            slave.sdo_write(0x8006, 2, bytes(ctypes.c_uint8(1))) # FIR disable
            slave.sdo_write(0x8006, 3, bytes(ctypes.c_uint8(0))) # FAST enable
            slave.sdo_write(0x8006, 4, bytes(ctypes.c_uint8(0))) # CHOP enable
            slave.sdo_write(0x8006, 1, bytes(ctypes.c_uint16(self.SINC_LENGTH))) # SINC Length
            ## Get sampling rate
            sampling_rate = struct.unpack('h', slave.sdo_read(0x8011, 0))[0]
            print(f"Sensor sampling rate: {sampling_rate} Hz.")
            if sampling_rate > 0:
                self.time_step = 1.0 / float(sampling_rate)
                self.receive_timeout_us = int(self.time_step * 1e6)
                print(f"Time step calculated: {self.time_step} seconds.")

    def read_and_send_data_through_tcp_socket(self):
        """Read and send sensor data over a TCP socket."""
        self._master.send_processdata() # Exchange process data with the sensor
        self._master.receive_processdata(self.receive_timeout_us)
        self.sensor_input_as_bytes = self._master.slaves[0].input
        self.client_socket.sendall(self.sensor_input_as_bytes)

    def check_for_shutdown(self):
        """Check if a shutdown signal is received."""
        ready_to_read, _, _ = select.select([self.client_socket], [], [], 0)
        if ready_to_read:
            data = self.client_socket.recv(64)
            if data:
                message = json.loads(data.decode('utf-8'))
                if message.get("action") == "shutdown":
                    self._shutdown_requested = True

    def run(self):
        """Main loop for reading and sending sensor data."""
        while not self._shutdown_requested:
            self.check_for_shutdown()
            if self._shutdown_requested:
                break
            self.read_and_send_data_through_tcp_socket()
            time.sleep(self.time_step)
```

```
        self.shutdown_ethercat()

    def shutdown_ethercat(self):
        """Safely shut down EtherCAT and TCP communication."""
        print("Shutdown signal received, stopping EtherCAT reading process.")
        self._master.state = pysoem.INIT_STATE  # Request INIT state for all slaves
        self._master.write_state()
        self._master.close()
        self.client_socket.close()
        self.server_socket.close()
        print("EtherCAT and TCP connections closed.")


if __name__ == "__main__":
    if len(sys.argv) != 4:
        print("Usage: sudo python3 ethercat_reader.py <interface> <sensor_name> <port>"
    )
        sys.exit(1)
    try:
        interface = sys.argv[1]
        sensor_name = sys.argv[2]
        port = int(sys.argv[3])
        EtherCAT_FT_Sensor_Logger(interface, sensor_name, port).run()
    except Exception as e:
        print(f"Error: {e}")
        sys.exit(1)
```

Listing V.1: *Python script:* TCP server that transmits EtherCAT F/T sensor data frame.

To execute this Python script, root privileges are required due to low-level network access:

```
sudo python3 ethercat_reader.py ens5 BFT-SENS-ECAT-M8 12001
```

Listing V.2: *Bash script:* launch the TCP server script.

3. TCP Client in ROS2 Node

The ROS2 node operates as a TCP client. It connects to the local server, receives raw sensor data, decodes it using Python's `struct` module, and publishes the parsed values as a `std_msgs/Float64MultiArray` message. This node runs in a dedicated terminal and must be launched after the TCP server has started.

Sensor data are published on the topic `/sensone_sensor_data` at a frequency of `800 Hz`, matching the sensor's acquisition rate and the server's transmission frequency, ensuring synchronisation prevents data loss.

Each message contains twelve `float64` values ordered as follows:

- Forces: `Fx`, `Fy`, `Fz` (N)
- Torques: `Mx`, `My`, `Mz` (Nm)
- Linear accelerations: `Ax`, `Ay`, `Az` (m/s^2)
- Angular velocities: `Rx`, `Ry`, `Rz` (rad/s)

```
"""
Date: April 2025

Description:
This ROS 2 node logs data from a Bota Systems SensONE force/torque sensor through a TCP-based server
and publishes the sensor readings at a user-defined frequency.
The node communicates with a local TCP server that streams raw sensor data. Parsed and scaled values
include force, torque, acceleration and angular velocity.

Parameters:
- 'SERVER_PORT' (int): TCP port number for communication with the TCP server (default: 12001)
- 'publishing_frequency' (double): Frequency in Hz at which sensor data is published (default: 800.0)

Subscribed Topics:
- '/supervisor_status_data' ('diagnostic_msgs/KeyValue'): Accepts a shutdown command when a message with key 'ft_sensors' and value '1' is received.

Published Topics:
- '/sensone_sensor_data' ('std_msgs/Float64MultiArray'): Publishes an array of sensor values including force, torque and IMU.
"""

import rclpy
from rclpy.node import Node
from diagnostic_msgs.msg import KeyValue
from std_msgs.msg import Float64MultiArray
import socket
import json
import select
import struct

class FT_Sensor_Logger(Node):

    __slots__ = (
        'publishing_frequency',
        'SERVER_PORT',
        'SERVER_HOST',
        'timer',
        'stop_node',
        'client_socket',
        'raw_data',
        'status',
        'warningsErrorsFatals',
        'Fx', 'Fy', 'Fz',
```

```
        'Mx', 'My', 'Mz',
        'forceTorqueSaturated',
        'Ax', 'Ay', 'Az',
        'accelerationSaturated',
        'Rx', 'Ry', 'Rz',
        'angularRateSaturated'
    )

    def __init__(self):
        super().__init__('ft_sensor_logger')

        self.SERVER_HOST = 'localhost'

        # Declare ROS 2 parameters
        self.declare_parameter('publishing_frequency', 800.0)
        self.declare_parameter('SERVER_PORT', 12001)
        self.publishing_frequency = self.get_parameter('publishing_frequency').
    get_parameter_value().double_value
        self.SERVER_PORT = self.get_parameter('SERVER_PORT').get_parameter_value().
    integer_value

        # ROS2 subscribers
        self.create_subscription(KeyValue, '/supervisor_status_data', self.
    stop_node_callback, 10)

        # ROS2 publisher
        self.ft_sensor_publisher = self.create_publisher(Float64MultiArray, '/
    sensone_sensor_data', 1)

        # Initialize variables
        self.stop_node = False
        self.raw_data = None

        self.status = None
        self.warningsErrorsFatals = None

        self.Fx = None
        self.Fy = None
        self.Fz = None
        self.Mx = None
        self.My = None
        self.Mz = None
        self.forceTorqueSaturated = None

        self.Ax = None
        self.Ay = None
        self.Az = None
        self.accelerationSaturated = None

        self.Rx = None
        self.Ry = None
        self.Rz = None
        self.angularRateSaturated = None

        # Initialisation de la connexion TCP
        self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.client_socket.connect((self.SERVER_HOST, self.SERVER_PORT))
        self.get_logger().info(f'TCP connection established to {self.SERVER_HOST}: {
    self.SERVER_PORT}.')

        # Periodic timer to compute and publish data
        timer_period = 1.0 / self.publishing_frequency
        self.timer = self.create_timer(timer_period, self.timer_callback)

        # End of the initialisation part
        self.get_logger().info(f"Node initialized for publishing SensONE FT sensor data
     at {self.publishing_frequency} Hz.")


    def timer_callback(self):
        """Periodic callback for reading and publishing sensor data."""
```

```
        if self.stop_node:
            self.send_shutdown_signal()
            self.on_shutdown()
        else:
            self.read_data_through_tcp_socket()
            self.publish_ft_sensor_data()

    def stop_node_callback(self, msg: KeyValue):
        if msg.key == 'ft_sensors' and msg.value == '1':
            self.stop_node = True

    def read_data_through_tcp_socket(self):
        """Read sensor data over a TCP socket."""
        ready_to_read, _, _ = select.select([self.client_socket], [], [], 0)
        if ready_to_read:
            self.raw_data = self.client_socket.recv(77)
            if self.raw_data:
                # General status byte
                self.status = struct.unpack_from('B', self.raw_data, 0)[0]
                # Warnings/Errors/Fatals register (32-bit)
                self.warningsErrorsFatals = struct.unpack_from('I', self.raw_data, 1)
    [0]
                # Extract Force-Torque (F/T) data
                self.Fx = struct.unpack_from('f', self.raw_data, 5)[0]
                self.Fy = struct.unpack_from('f', self.raw_data, 9)[0]
                self.Fz = struct.unpack_from('f', self.raw_data, 13)[0]
                self.Mx = struct.unpack_from('f', self.raw_data, 17)[0]
                self.My = struct.unpack_from('f', self.raw_data, 21)[0]
                self.Mz = struct.unpack_from('f', self.raw_data, 25)[0]
                self.forceTorqueSaturated = struct.unpack_from('H', self.raw_data, 29)
    [0]
                # Linear acceleration data
                self.Ax = struct.unpack_from('f', self.raw_data, 31)[0]
                self.Ay = struct.unpack_from('f', self.raw_data, 35)[0]
                self.Az = struct.unpack_from('f', self.raw_data, 39)[0]
                self.accelerationSaturated = struct.unpack_from('B', self.raw_data, 43)
    [0]
                # Angular velocity data
                self.Rx = struct.unpack_from('f', self.raw_data, 44)[0]
                self.Ry = struct.unpack_from('f', self.raw_data, 48)[0]
                self.Rz = struct.unpack_from('f', self.raw_data, 52)[0]
                self.angularRateSaturated = struct.unpack_from('B', self.raw_data, 56)
    [0]

    def publish_ft_sensor_data(self):
        """Publish sensor measurements."""
        msg = Float64MultiArray()
        msg.data = [self.Fx, self.Fy, self.Fz,
                    self.Mx, self.My, self.Mz,
                    self.Ax, self.Ay, self.Az,
                    self.Rx, self.Ry, self.Rz]
        self.ft_sensor_publisher.publish(msg)

    def send_shutdown_signal(self):
        """Sends a signal to the Python script to shut it down properly."""
        shutdown_message = json.dumps({"action": "shutdown"})
        self.client_socket.sendall(shutdown_message.encode('utf-8'))
        self.get_logger().info('Shutdown signal sent to the EtherCAT reader script.')

    def on_shutdown(self):
        """Stop the node properly."""
        self.get_logger().info("Stopping node command received shutting down proprely
    the node.")
        self.client_socket.close()
        self.get_logger().info("Socket closed.")
        self.get_logger().info("Stopping node.")
        raise SystemExit


def main(args=None):
    rclpy.init(args=args)
```

```
    node = FT_Sensor_Logger()
    try:
        rclpy.spin(node) # Run the node
    except SystemExit:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Listing V.3: *Python script:* ROS2 node receiving TCP data and publishing it.

To execute this Python script:

```
ros2 run ft_sensors_manager ft_sensor_sensone
```

Listing V.4: *Bash command:* Run the ROS2 node.

Bibliographic References

[1] Bota Systems. Bota systems official website. https://www.botasys.com/fr. Accessed: 2025-05.

[2] Bota Systems. Rokubi force-torque sensor. https://www.botasys.com/fr/force-torque-sensors/rokubi. Accessed: 2025-05.

[3] Bota Systems. Sensone force-torque sensor. https://www.botasys.com/fr/force-torque-sensors/sensone. Accessed: 2025-05.

[4] Benjamin Partzsch. Pysoem documentation. https://pysoem.readthedocs.io/en/latest/. Accessed: 2025-05.

[5] Bota Systems. Bota systems python interface. https://gitlab.com/botasys/python_interface. Accessed: 2025-05.

[6] Wikipedia. Sinc filter. https://en.wikipedia.org/wiki/Sinc_filter. Accessed: 2025-05.

[7] Wikipedia. Finite impulse response. https://en.wikipedia.org/wiki/Finite_impulse_response. Accessed: 2025-05.