Hochschule Bremerhaven

# M.Sc. Embedded System Design

A Report on

# Temperature Measurement Network Gateway

Embedded Project

Under the guidance of

**Prof. Dr.-Ing. Kai Müller**

**Prof. Dr.-Ing. Karsten Peter**

*Submitted By*


| | |
|---|---|
| **Srujana Chowdary Maddipatla** | **38768** |
| **Anees Ahmed Zuberi** | **38711** |
| **Asher Ali** | **38703** |
| **Khizar Akhtar** | **38708** |
| **Muhammad Ather Hussain** | **38701** |
| **Nikhil Narayanan** | **38746** |


**Submission Date: 06.06.2022**

# ACKNOWLEDGEMENT

In the present world of competition, there is a race of existence in which those are having will to come forward succeed. A project is like a bridge between theoretical and practical working. With this will, we joined this project.

We would like to express our utmost and deepest appreciation to our supervisors Prof. Dr. -Ing. Kai Müller and Prof. Dr.-Ing. Karsten Peter who made all this possible. Their guidance and advice carried was a key source for us to be able to complete our project. We would also like to give our special thanks to Dipl. -Ing (FH) Andreas Menslage and Dipl. -Ing (FH) Class Schott for allowing us to use the University lab and helping us resolve all the issues in the lab we faced during the project.

In the end, we would like to thank our colleagues who have worked hard to complete this project.

# Table of Contents

# List of Figures

# List of Tables

# Work Organization

| S.No. | Team Members | Chapters |
|---|---|---|
| 1 | Muhammad Ather Hussain | 1, 2, 7.1, 7.3 |
| 2 | Srujana Chowdary Maddipatla | 3 |
| 3 | Khizar Akhtar | 6, 7.6 |
| 4 | Anees Ahmed Zuberi | 4, 5,7.1 |
| 5 | Nikhil Narayanan | 7.2, 7.4, 7.5, 9, 10,11 |
| 6 | Asher Ali | 8 |

# 1. Introduction

Secure and reliable communication is important to establish seamless connectivity amongst the end-users. Depending on the physical environment, devices located at the endpoint of the network often operate under harsh conditions. Acquiring data from such sensors demands a robust protocol to identify and report any unfortunate events to the central network. A modern car is equipped with more than hundreds of embedded systems. With the advent of Internet of things (IoT) and 5G communication, and artificial intelligence, this number is expected to be increased more. Along with this, demanding applications like logistics, space, and modern medical systems also demand innovative solutions adhering such needs. So, data flowing in/out of such large number of embedded systems requires robust method that can administer and manage packets transmitted over the network. An example of a modern car connectivity through Gateway can be seen in Figure 1.

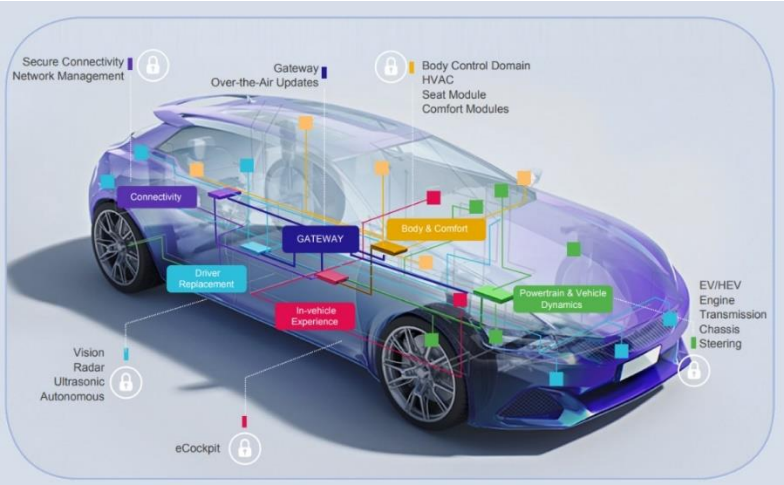



*Figure 1: Modern car gateway interconnection [1]*

To achieve this purpose, a platform is required for building communication bridge amongst various devices. To accomplish this, network management gateway (MNG) comes into place. MNG, also referred as Network management system (NMS), is an application or set of applications that lets network administrators manage a network's independent components inside a bigger network management framework. It allows to record and send data from different nodes to software and hardware components associated with a particular application.

A modern vehicle uses heterogeneous networks like Controller Area Network (CAN), Local interconnect network LIN, FlexRay, Ethernet, and wireless network protocols to communicate with different Electronic Control Units (ECUs). For example, a LIN is used for low speed applications like sensors and actuators (20 kbps), CAN is used for medium-speed applications, including most ECU-to-ECU communications (1-5 Mbps), FlexRay is used for real-time, safety-critical applications (10 Mbps), and Ethernet is used for high-speed applications such as

infotainment and advanced driver-assistance systems (ADAS), as well as wireless interfaces (3G/4G/future 5G, BT, Wi-Fi, V2X) (100 Mbps to gigabit speeds), [1]. MNG acts as a central hub providing secure and reliable interconnection. Its aim is to provide secure, seamless communications between networks, and ECUs It provides physical isolation and protocol translation to route data between functional.

# 2. System Design

In this project, a network management gateway application is developed for swift and reliable communication amongst the different nodes of the network. It uses the client and server approach for communication between remote devices. The application is built over the Linux operating system (OS) running over the Xilinx's Zedboard to interacting with hardware components. At the client side, graphical platform is built to monitor and configure distant sensor devices over the network. The illustration of project can be seen in Figure 2.

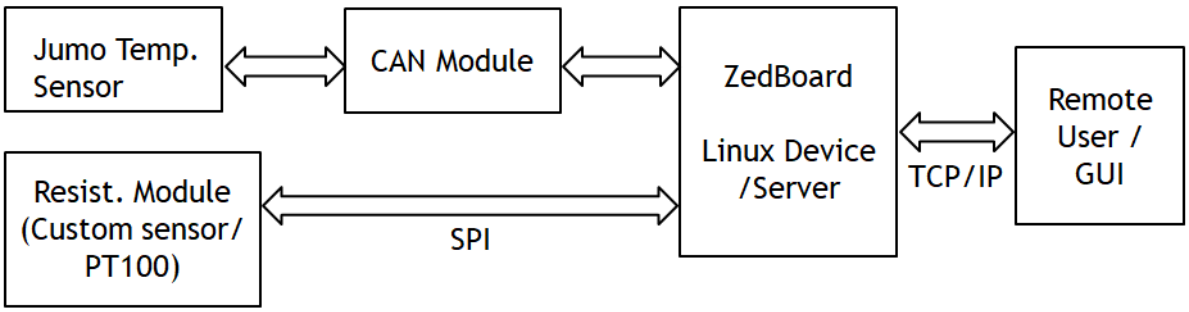



*Figure 2: Server and Client approach between remote devices*

The project is divided into hardware and software parts that fully integrate later to make project working. In Hardware components, CANopen, Resistive Module involving test module and sensor development is included. On the other hand, Linux kernel is built to integrate hardware information into the software providing a platform to run the application. It uses ARM core processors of Xilinx's Zedboard to operate Linux OS. Further, a Remote screen or Graphical user interface is developed which communicates with the hardware using Transmission Control Protocol (TCP/IP). Below, different components of each category are described in further details.

## 2.1. Design Components

The following parts of our project make up the whole design complete and functional.

### 2.1.1. ZedBoard

ZedBoard is an evaluation and development board based on the Xilinx Zynq-7000 Extensible Processing Platform which is the center show of our project. It contains the complete System on Chip package containing dual Corex-A9 Processing System (PS) with 85,000 Series-7 Programmable Logic (PL) cells [2]. The developments kit comes with various on-board peripheral and expansion capabilities making it an ideal platform for both novice and experienced designers. The features provided by the ZedBoard consist of:

- Xilinx® XC7Z020-1CSG484CES EPP
  - Primary configuration = QSPI Flash
  - Auxiliary configuration options
    - Cascaded JTAG
    - SD Card
- Memory
  - 512 MB DDR3 (128M x 32)
  - 256 Mb QSPI Flash
- Interfaces
  - USB-JTAG Programming using Digilent SMT1-equivalent circuit
    - Accesses PL JTAG
    - PS JTAG pins connected through PS Pmod
  - 10/100/1G Ethernet
  - USB OTG 2.0
  - SD Card
  - USB 2.0 FS USB-UART bridge
  - Five Digilent Pmod™ compatible headers (2x6) (1 PS, 4 PL)
  - One LPC FMC
  - One AMS Header
  - Two Reset Buttons (1 PS, 1 PL)
  - Seven Push Buttons (2 PS, 5 PL)
  - Eight dip/slide switches (PL)
  - Nine User LEDs (1 PS, 8 PL)
  - DONE LED (PL)
- On-board Oscillators
  - 33.333 MHz (PS)
  - 100 MHz (PL)
- Display/Audio
  - HDMI Output
  - VGA (12-bit Color)
  - 128x32 OLED Display
  - Audio Line-in, Line-out, headphone, microphone
- Power
  - On/Off Switch
  - 12V @ 5A AC/DC regulator
- Softwares
  - Vitis™ (Software development)
  - Vivado® Design Suite (Hardware development)

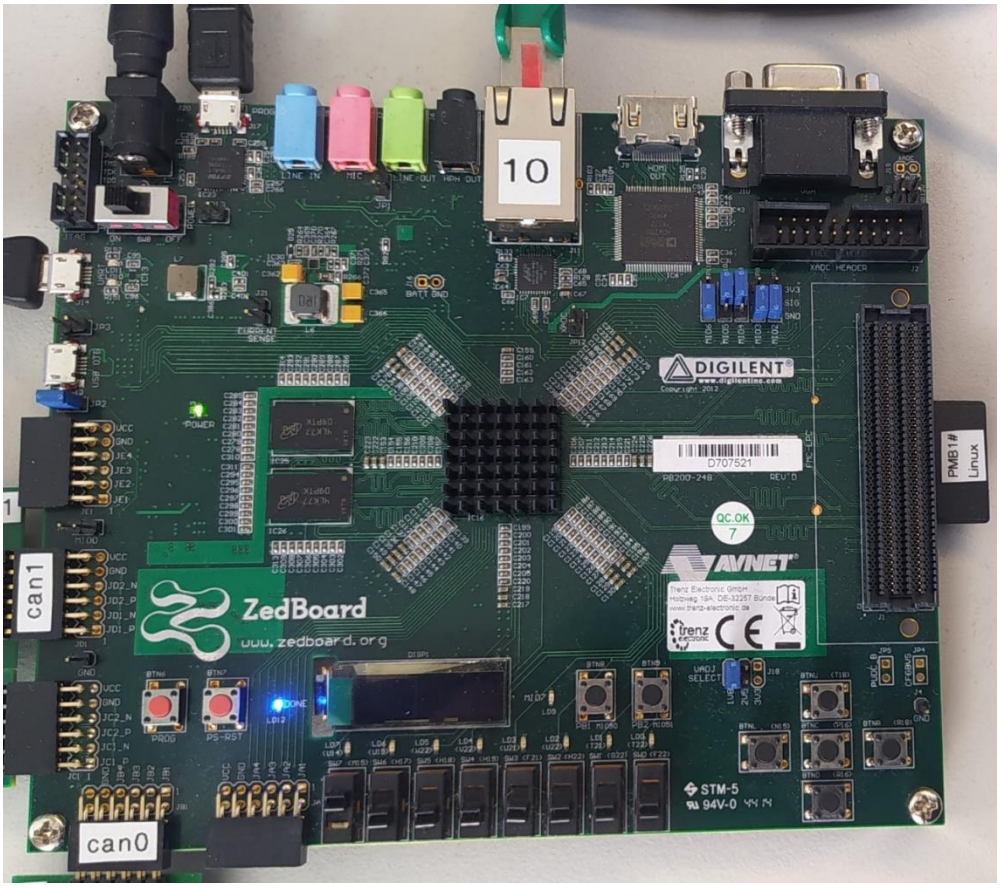



*Figure 3: ZedBoard development platform*

### 2.1.2. CAN Bus Interface Module

An interface is required to establish communication with CANopen device and main application. The module contains comprises of CAN transceiver and the CAN protocol controller. The ISO1050 Isolated CAN Transceiver converts Transmit (Tx) and Receive (Rx) signals into differential CAN Low and CAN High signals and vice versa.The CAN protocol controller is in most cases on-chip of the micro-controller – sometimes named host controller. The CANopen protocol stack implements the CANopen protocols and the CANopen object dictionary [3]. Further details are explained later in Section 3. The CAN Bus interface module used in our project is developed at HS Bremerhaven.


*Figure 4: CAN Bus Interface module*

### 2.1.3. CANopen device / CAN Sensor

The CANopen device or CAN Sensor is the frontline device that measures or send output signal. In this project, such data acquisition system is a temperature sensor which part of series of devices manufactured by JUMO GmbH & Co. KG, Germany. The *JUMO CANtrans T*, can be seen in Figure 4, is a resistance temperature detector (RTD) temperature probe which is the preferred choice for temperature measurement in liquids and gases. The measuring insert is equipped with a PT1000 temperature sensor according to DIN EN 60751:2009 / IEC 60751:2008, class B, as a standard feature. The measured temperature value is digitalized, linearized, and made available for further processing through the CANopen serial bus protocol (CAN slave) [4].

*Figure 5: CANopen Device / Temperature Sensor*

The sensor is attributed with the following features:

- Measure temperatures from -50 to +450 °C
- As single or double RTD temperature probe
- Vibration-resistant construction
- Limit value monitoring function
- Setting via standard CANopen software tools

CAN sensor is connected to CAN Bus interface via M12, 5-pole connector, can be seen in Figure 7. The device is powered on with 12V DC and GND via CAN Sensors Harness, developed at HS Bremerhaven. The CAN Bus interface module also gets connected to the network with via CAN Bus. Whole interconnection of the hardware setup is shown in Figure 6.

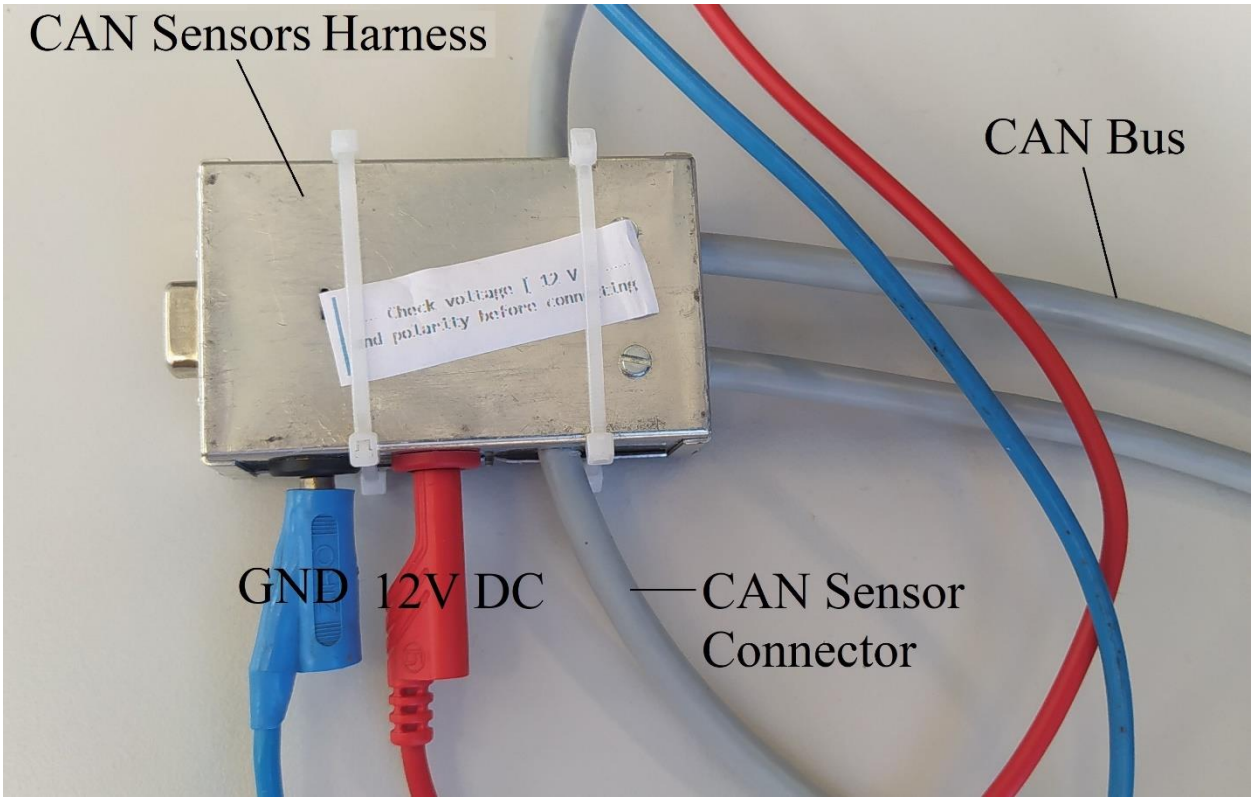



*Figure 6: CAN interconnection between CAN sensor and CAN interface module*

*Figure 7: M12, 5-pole, CAN Sensor Connector*

### 2.1.4. Resistive Peripheral Module

The Resistive peripheral module is a platinum RTD resistive module which is setup to operate with PT100 platinum RTD. Module is packaged with the necessary hardware to interface the MAX31865 RTD-to-digital converter to any system that utilizes Pmod compatible expansion ports configurable for SPI communication. The IC is an easy-to-use resistance-to-digital converter with in-built 15-bit Analog to Digital convertors (ADC), fault detection, and active noise cancellation properties [5]. The MAX31865PMB1 peripheral module features following list of characteristics:

- Simple Conversion of Platinum RTD Resistance to a Digital Value
- Handles 100Ω to 1kΩ (at 0°C) Platinum RTDs (PT100 to PT1000)
- Compatible with 2-, 3-, and 4-Wire Sensor Connections
- 6-Pin Pmod-Compatible Connector (SPI)
- RoHS Compliant
- Proven PCB Layout
- Fully Assembled and Tested

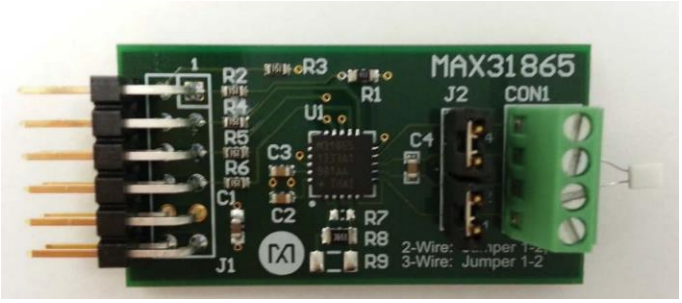



*Figure 8: MAX31865PMB1 peripheral module [5]*

The MAX31865 IC contains 8-bit registers that contain conversion, status, and configuration data. These registers are accessed using the 0xh addresses for reads and the 8xh addresses for writes. Whereas data is read from or written to the registers with MSB first. Further details about registers and their can be seen in Table 1.

*Table 1: MAX31865 IC's registers information*

| REGISTER NAME | READ ADDRESS (HEX) | WRITE ADDRESS (HEX) | READ/WRITE |
|---|---|---|---|
| Configuration | 00h | 80h | R/W |
| RTD MSBs | 01h | - | R |
| RTD LSBs | 02h | - | R |
| High Fault Threshold MSB | 03h | 83h | R/W |
| High Fault Threshold LSB | 04h | 84h | R/W |
| Low Fault Threshold MSB | 05h | 85h | R/W |
| Low Fault Threshold LSB | 06h | 86h | R/W |
| Fault Status | 07h | - | R |

In addition to this, to calculate the resistance *R(T)* temperature *T*, following Callendar-Van Dusen equation is used.

Resistance *R* at given temperature *T*: $R(T) = R_0(1 + \alpha T + \beta T^2)$ (1)

Where,

*T* = Temperature (°C)

*R(T)* = Resistanc at temperature *T*

$R_0$ = Resistance at 0°C

$\alpha$ = 3.9083 x $10^{-3}$ (1/°C)

$\beta$ = -5.775 x $10^{-7}$ (1/°$C^2$)

### 2.1.5. Custom Developed Sensor

A resistive wound sensor was built in Hochschule Bremerhaven. This sensor would be used to check the level of liquid nitrogen in a cryogenic temperature. This sensor would be used in aerospace application. After development this sensor would be tested at DLR. Sensor development is discussed in detail in chapter 6.

### 2.1.6. Linux Kernel

It is the most important part of the project. This serves as the platform to every individual piece of hardware device and software application to run on the ZedBoard. The Linux OS is needed to various network protocols, multithreading, automatic interrupt handling etc. which makes it simple to use as compared to bare metal application. The OS Kernal can be developed using a PetaLinux which is basically a command line tool for embedded Linux provided by AMD's Xilinx. PetaLinux tools eases the development of Linux-based products; all the way from system boot to execution [6]. Petalinux is based on open source and de-facto standard YOCTO. PetaLinux simplifies the process of creating bootable image by generating *BOOT.bin*, *image.ub*, and *boot.scr* which can be then loaded into the SD card.

### 2.1.7. Programmable Logic Interface

The Programmable logic (PL) interface creates hardware logic into the FPGA. It ensures the processing System (PS) of the design is able to communicate with the logic developed inside PL. It allows user to interact with the RTD resistive module on software side of the design using Serial Peripheral Interface (SPI). It also routes to the various multiplexed IOs, UART, external DDR RAM, and on-board peripheral such as Switches, LEDs and Buttons. Following is the view of block design of PL interface inside the IP Integrator tool of the Vivado Design suite.

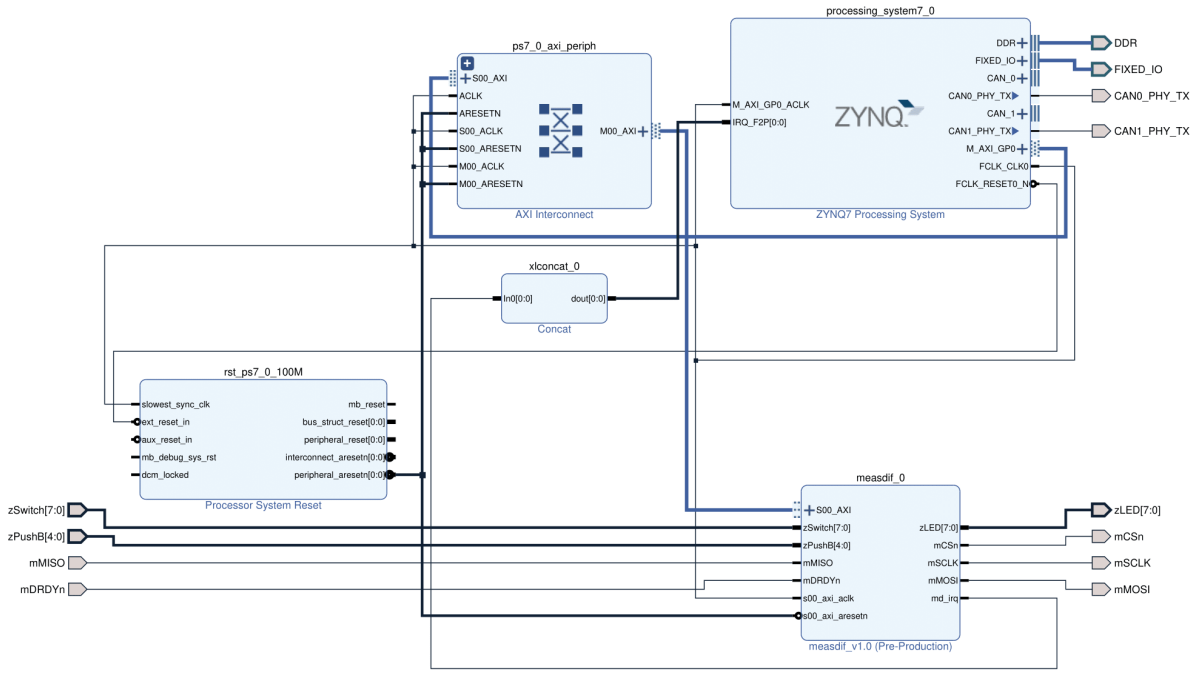



*Figure 9: Programmable Logic Implementation in an FPGA*

# 3. Literature

## 3.1. Introduction to Controller Area Network (CAN)

The Controller Area Network (CAN) is a serial communication bus designed for robust and flexible performance in harsh environments, and particularly for industrial and automotive applications. More precisely CAN bus is a robust vehicle bus standard designed to allow microcontrollers and devices to communicate with each other's applications without a host computer. CAN was developed to reduce cable wiring, so the separate electronic control units (ECUs) inside a vehicle could communicate with only a single pair of wires.

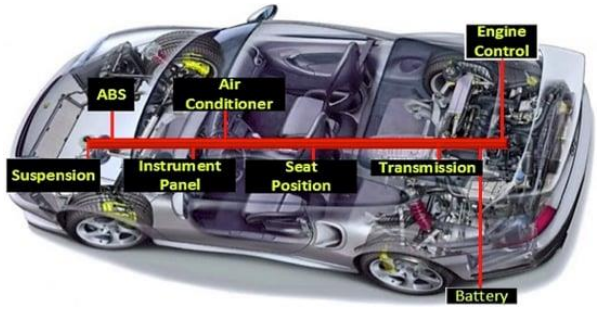



*Figure 10: CAN used for ECU communication in cars [39]*

It is a message-based protocol, designed originally for multiplex electrical wiring within automobiles to save on copper, but it can also be used in many other contexts. For each device, the data in a frame is transmitted sequentially but in such a way that if more than one device transmits at the same time, the highest priority device can continue while the others back off. Frames are received by all devices, including by the transmitting device.

An ECU can prepare and broadcast information like sensor data via the CAN bus. The broadcasted data is accepted by all other ECUs on the CAN network and each ECU can then check the data and decide whether to receive or ignore it.

In this project, the higher layer protocol CANopen is implemented. This is an open-source stack, easy to understand and it contains the full functionality for the integration of the CANopen standards as defined in CiA 301, 302 and 305. It also provides low storage and optimized performance range.

## 3.2. History of CAN

Originally invented by Bosch and later codified into the ISO11898-1 standard, CAN defines the data link and physical layer of the Open Systems Interconnection (OSI) model, providing a low-level networking solution for high-speed in-vehicle communications.

The CAN bus history in short

- **Pre CAN:** Car ECUs relied on complex point-to-point wiring
- **1986:** Bosch developed the CAN protocol as a solution

- **1991:** Bosch published CAN 2.0 (CAN 2.0A: 11 bits, 2.0B: 29 bit)
- **1993:** CAN is adopted as international standard (ISO 11898)
- **2003:** ISO 11898 becomes a standard series
- **2012:** Bosch released the CAN FD 1.0 (flexible data rate)
- **2015:** The CAN FD protocol is standardized (ISO 11898-1)
- **2016:** The physical CAN layer for data-rates up to 5 Mbit/s standardized in ISO 11898-2

In February of 1986, CAN was born: at the SAE congress in Detroit, the new bus system was introduced as the 'Automotive Serial Controller Area Network. Uwe Kiencke, Siegfried Dais, and Martin Litschel introduced the multi-drop network protocol. It was based on a non-destructive arbitration mechanism, which grants bus access to the frame with the highest priority without any delays.

In the early 1990s, the time as right to find a user's group to promote the CAN protocol and to foster its use in many applications. In January of 1992, Holger Zeltwanger, at that time editor of the VMEbus magazine, brought users and manufacturers together to establish a neutral platform for the technical enhancement of CAN as well as the marketing of the serial bus system. Two month later, the 'CAN in Automation' (CiA) international users and manufacturers group was officially founded.

Since 1993, within the scope of the Esprit project Aspic, a European consortium led by Bosch had been developing a prototype of what would become CANopen. The first CANopen networks were used for internal machine communication, especially for drives. CANopen offers very high flexibility and configurability. The higher-layer protocol, which has been used in several very different application areas (industrial automation, maritime electronics, military vehicles, etc.) has in the meantime been internationally standardized as EN 50325-4 (2003). CANopen is being used especially in Europe. Injection molding machines in Italy, wood saws and machines in Germany, cigarette machines in Great Britain, cranes in France, handling machines in Austria, and clock-manufacturing machines in Switzerland are just a few examples within industrial automation and machine building.

## 3.3. The CAN Standard

CAN is an International Standardization Organization (ISO) defined serial communications bus originally developed for the automotive industry to replace the complex wiring harness with a two-wire bus. The specification calls for high immunity to electrical interference and the ability to self-diagnose and repair data errors. These features have led to CAN's popularity in a variety of industries including building automation, medical, and manufacturing. The CAN communications protocol, ISO-11898: 2003, describes how information is passed between

devices on a network and conforms to the Open Systems Interconnection (OSI) model that is defined in terms of layers. Actual communication between devices connected by the physical medium is defined by the physical layer of the model. The ISO 11898 architecture defines the lowest two layers of the seven-layer OSI/ISO model as the data-link layer and physical layer.

For embedded networking applications, not all the layers in the OSI model are implemented. The interface is required between any two layers to implement the OSI model. This results in an overhead that is not acceptable in embedded applications. Due to this reason, higher layer CAN protocol only implement selected functionality from the higher layers, to minimize the overhead.

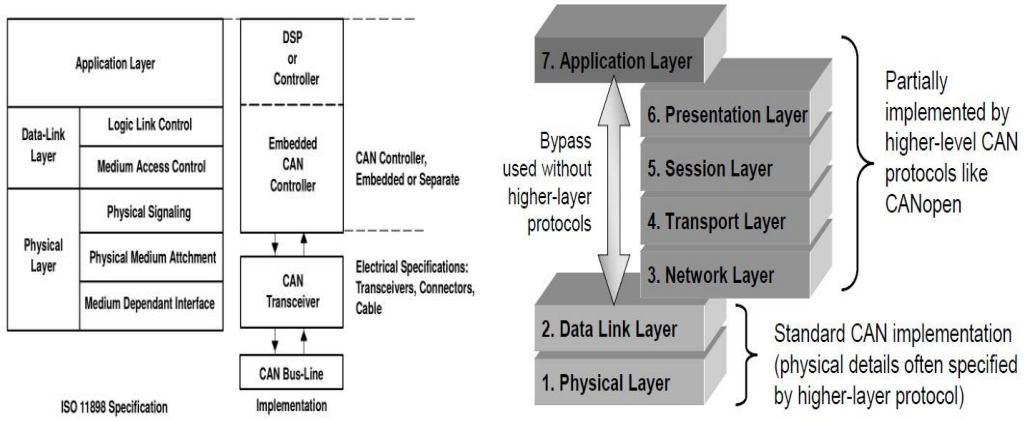



*Figure 11: OSI model CAN/CANopen [9]*

All seven layers of OSI model from figure 11 is explained with respect to CAN and CANopen in below section.

### 3.3.1. Physical Layer:

The lowest layer of the OSI Model is concerned with electrically or optically transmitting raw unstructured data bits across the network from the physical layer of the sending device to the physical layer of the receiving device. It can include specifications such as voltages, pin layout, cabling, and radio frequencies. At the physical layer, one might find “physical” resources such as network hubs, cabling, repeaters, network adapters or modems.

- **Baud rate:** CAN nodes must be connected via a two-wire bus with baud rates up to 1 Mbit/s (Classical CAN) or 5 Mbit/s (CAN FD-flexible data rate).
- **Cable length:** Maximal CAN cable lengths should be between 500 meters (125 kbit/s) and 40 meters (1 Mbit/s).
- **Termination:** The CAN bus must be properly terminated using a 120 Ohms CAN bus termination resistor at each end of the bus.

### 3.3.2. Data Link Layer:

At the data link layer, directly connected nodes are used to perform node-to-node data transfer where data is packaged into frames. The data link layer also corrects errors that may have occurred at the physical layer.

The data link layer encompasses two sub-layers of its own. The first, media access control (MAC), provides flow control and multiplexing for device transmissions over a network. The second, the logical link control (LLC), provides flow and error control over the physical medium as well as identifies line protocols.

### 3.3.3. Network Layer:

The network layer is responsible for receiving frames from the data link layer and delivering them to their intended destinations among based on the addresses contained inside the frame. The network layer finds the destination by using logical addresses, such as IP (internet protocol). At this layer, routers are a crucial component used to quite literally route information where it needs to go between networks.

### 3.3.4. Transport Layer:

The transport layer manages the delivery and error checking of data packets. It regulates the size, sequencing, and ultimately the transfer of data between systems and hosts. One of the most common examples of the transport layer is TCP or the Transmission Control Protocol.

### 3.3.5. Session Layer:

It allows to establish, communicate, and terminate sessions between processes running on two different devices performing security, name recognition, and logging.

- It allows any host to begin and end communication sessions. This is not used in CANopen.
- CANopen SDO channel allows only the side holding the token (master) can perform critical operations like write access to the shared database.
- Synchronization is partially supported by CANopen SDO block transfer of data with abort, but no resume of interrupted transfer is supported.

### 3.3.6. Presentation Layer:

The most important function of this layer is defining data formats such as ASCII text, BINARY, BCD, and JPEG. It acts as a translator for data into a format used by the application layer at the receiving end of the station.

- CANopen Object Dictionary and defined data types handle data representation and encode data in a standardized way.
- Data encryption and decryption, data compression is not supported by CANopen.

### 3.3.7. Application Layer:

It serves as a window for users and application processes to access network services. The common functions of the layers are resource sharing, remote file access, network management.

## 3.4. Working Principle of CAN

Every CAN mechanism consists of a CAN device. This CAN device sends data across CAN network in the form of packets and these packets are called CAN frame. Every CAN frame consists of arbitration ID, a data field, a remote frame, an overload frame and an error frame.it converts the data stream from CAN-bus level to CAN-controller level. And, while transmitting it converts the data stream from the CAN controller to CAN-bus levels.

In CAN transmission, logical 1 is called recessive state and logical 0 as dominant state.

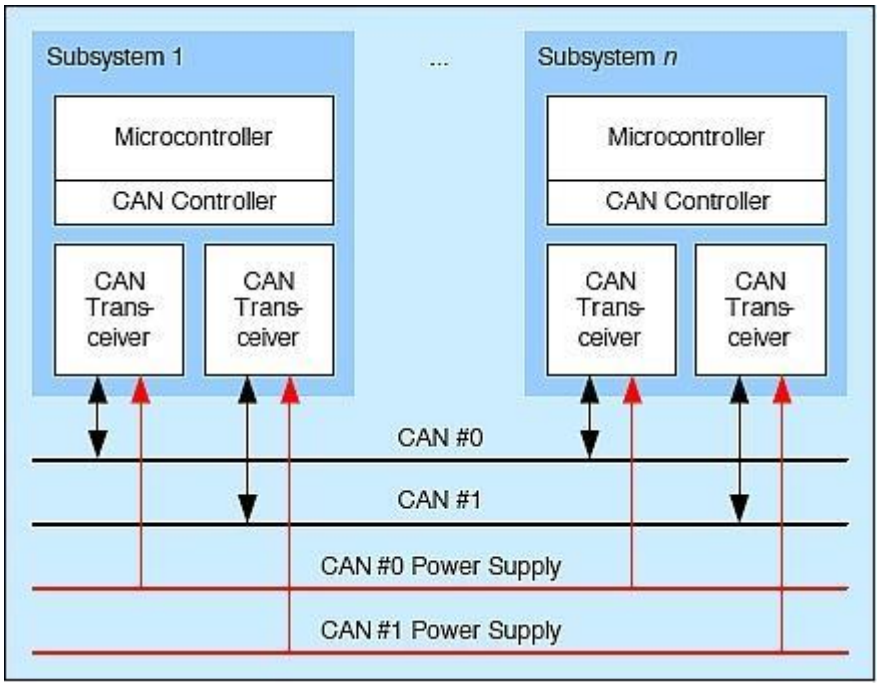



*Figure 12: CAN-Bus Architecture [38]*

Suppose, if one or more node transmits logical 0 (dominant), and logical 1(recessive) is transmitted by rest of the nodes, then logical 0 is seen by all nodes including nodes that sent 1. Since 0 has the highest priority. CAN frame with 11 or 29-bit identifier is transmitted at the start, the node with lowest identifier transmits more zeros at the start of the frame, and that node has the priority.

The two signal lines CANH and CANL of the bus are in the quiescent recessive state as shown in Figure 13. They are passively biased to 2.5V. The dominant state on the bus takes CANH to 3.75V and CANL to 1.25V creating a typical 2.5V differential signal. By sending the data in equal and opposite ways, this allows for greater noise immunity and therefore less chance of the data being corrupted.

- Status of bit with the value 0 = 2.5V differential voltage = dominant state
- Status of bit with the value 1 = 0V differential voltage = recessive state

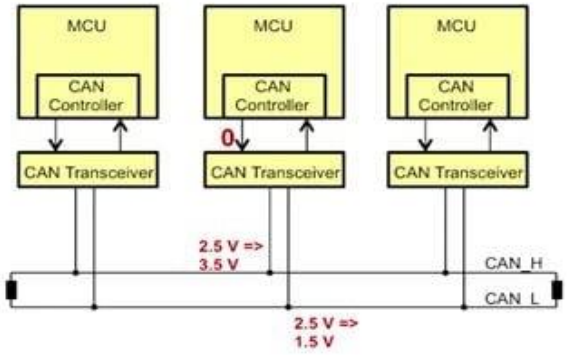



*Figure 13: CAN Bus line Voltage level [10]*

## 3.5. Bit fields of Standard CAN and Extended CAN

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **SOF** | **11-bit Identifier** | **RTR** | **IIDE** | **rr00** | **DLC** | **0…8Bytes Data** | **CRC** | **ACK** | **EEOOFF** | **IFS** |

*Figure 14: Standard CAN Frame 11-bit Identifier [12]*

- SOF–The single dominant start of frame (SOF) bit marks the start of a message and is used to synchronize the nodes on a bus after being idle.
- Identifier-The Standard CAN 11-bit identifier establishes the priority of the message. The lower the binary value, the higher its priority.
- RTR–The single remote transmission request (RTR) bit is dominant when information is required from another node. All nodes receive the request, but the identifier determines the specified node. The responding data is also received by all nodes and used by any node interested. In this way, all data being used in a system is uniform.
- IDE–A dominant single identifier extension (IDE) bit means that a standard CAN identifier with no extension is being transmitted.
- r0–Reserved bit (for possible use by future standard amendment).
- DLC–The 4-bit data length code (DLC) contains the number of bytes of data being transmitted.
- Data–Up to 64 bits of application data may be transmitted.
- CRC–The 16-bit (15 bits plus delimiter) cyclic redundancy check (CRC) contains the checksum
- (number of bits transmitted) of the preceding application data for error detection.
- ACK–Every node receiving an accurate message overwrites this recessive bit in the original message with a dominate bit, indicating an error-free message has been sent. Should a receiving node detect an error and leave this bit recessive, it discards the message and the

sending node repeats the message after arbitration. In this way, each node acknowledges (ACK) the integrity of its data. ACK is 2 bits, one is the acknowledgment bit and the second is a delimiter.

- EOF–This end-of-frame (EOF), 7-bit field marks the end of a CAN frame (message) and disables bit stuffing, indicating a stuffing error when dominant. When 5 bits of the same logic level occur in succession during normal operation, a bit of the opposite logic level is stuffed into the data.
- IFS–This 7-bit interframe space (IFS) contains the time required by the controller to move a correctly received frame to its proper position in a message buffer area.


*Figure 15: CAN Extended Frame 29-Bit Identifier [12]*

As shown in Figure 15, the Extended CAN message is the same as the Standard message with the addition of:

- SRR–The substitute remote request (SRR) bit replaces the RTR bit in the standard message location as a placeholder in the extended format.
- IDE–A recessive bit in the identifier extension (IDE) indicates that more identifier bits follow. The 18-bitextension follows IDE.
- r1–Following the RTR and r0 bits, an additional reserve bit has been included ahead of the DLC bit.

## 3.6. CAN Frames

### 3.6.1. Data Frame

The data frame is the most common message type, and comprises the arbitration field, data field, the CRC field, and the acknowledgment field, which are explained below.

*SOF – Start of Frame bit*

It indicates the start of the message and is used to synchronize the nodes on a bus. A dominant bit in the field marks the start of the frame. **Arbitration Field - 12 or 32 Bits**

- Identifier 11or 29 bits - to determine which node has access to the bus and to identify the type of message.
- An 11-bit identifier (standard format) allows a total of $2^{11}$ (2048) different messages. A 29-bit identifier (extended format) allows a total of $2^{29}$ (536 million) messages.

- ACK – Acknowledge (ACK) field. It compromises the ACK slot and the ACK delimiter. When the data is received correctly the recessive bit in the ACK slot is overwritten as the dominant bit by the receiver.

**EOF – End of Frame (EOF)**
The 7-bit field marks the end of a CAN frame.

As shown in Figure 15, the extended CAN frame is same as standard message with added fields as shown below.

- SRR – The substitute remote request (SRR) bit replaces the RTR bit in the standard message location as a placeholder in the extended format.
- IDE – A recessive bit in the identifier extension (IDE) indicates that more identifier bits follow. The 18-bit extension follows IDE.
- r1 – Following the RTR and R bits, an additional reserve bit has been included ahead of the DLC bit

### 3.6.2. Remote Frame

The intended purpose of the remote frame is to solicit the transmission of data from another node. The remote frame is like the data frame, with two important differences. First, this type of message is explicitly marked as a remote frame by a recessive RTR bit in the arbitration field, and secondly, there is no data.

### 3.6.3. Error Frame

The error frame is a special message that violates the formatting rules of a CAN message. It is transmitted when a node detects an error in a message and causes all other nodes in the network to send an error frame as well. The original transmitter then automatically retransmits the message. An elaborate system of error counters in the CAN controller ensures that a node cannot tie up a bus by repeatedly transmitting error frames.

### 3.6.4. Overload Frame

The overload frame is mentioned for completeness. It is like the error frame about the format, and it is transmitted by a node that becomes too busy. It is primarily used to provide for an extra delay between messages.

### 3.6.5. Valid Frame

A message is error free when the last bit of the ending EOF field of a message is received in the error-free recessive state. A dominant bit in the EOF field causes the transmitter to repeat a transmission.

## 3.7. CANopen Protocol

CANopen is a "Layer 7" CAN protocol that defines communication and device functions for CAN-based systems. It is a standardized, highly flexible, and highly configurable embedded network architecture used in industries such as Railway, Medical, Industrial, Agriculture, Heavy Truck & Bus, Marine, Off-Highway, Factory Automation, Aerospace, and many others.

CANopen is also becoming a popular choice for closed, company-specific embedded networks. The CANopen profile family specifies standardized communication mechanisms and device functionalities. The CANopen Standard is maintained by "CAN in Automation (CiA).

## 3.8. Device model

A unified view of CANopen devices requires the use of a general device model so that different devices can be described by one standard.

The device model consists of three main components:

- Communication
- Object Dictionary
- Application

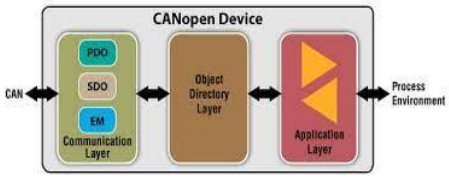



*Figure 16: CANopen [11]*

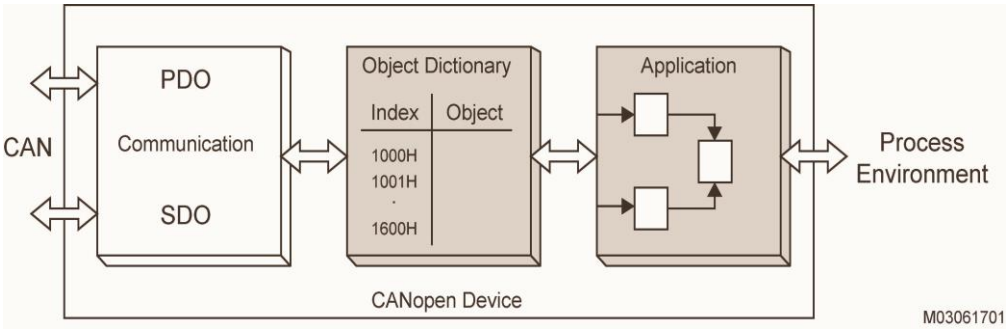



*Figure 17: CANopen device architecture [11]*

## 3.9. Object Dictionary

The Object Dictionary is a collection of all the data items which have an influence on the behaviour of the application objects, the communication objects and the state machine used on this device. It serves as an interface between the communication and the application. The object dictionary is essentially a grouping of objects accessible via the network in an ordered pre-defined fashion. Each object within the object dictionary is addressed using a 16-bit index and an 8-bit sub-index.

CANopen devices must have an object dictionary (OD), which is used for configuration and communication with the device as shown in above Fig (17).

An entry in the object dictionary is defined by:

- **Index**, the 16-bit address of the object in the dictionary
- **Object name** (Object Type/Size), a symbolic type of the object in the entry, such as an array, record, or simple variable
- **Name**, a string describing the entry
- **Type,** gives the datatype of the variable (or the datatype of all variables of an array)
- **Attribute**, which gives information on the access rights for this entry, this can be read/write, read-only or write-only
- The **Mandatory/Optional** field (M/O) defines whether a device conforming to the device specification must implement this object or not.

Each object is addressed using a 16-bit index represented as a four-digit hexadecimal number

*Table 2: Overview of OD areas [10]*

| Index(hex) | Object group |
|---|---|
| 0000h | Reserved |
| 0001 h – 009F h | Static and Complex Data types |
| 00A0 h – 0FFF h | Reserved |
| 1000 h – 1FFF h | Communication profiles (e.g., DS 301, DS 302) |
| 2000 h – 5FFF h | Manufacturer-specific device profiles |
| 6000 h – 9FFF h | Standardized device profiles |
| A000 h – FFFF h | Reserved |

Electronic Datasheets (EDS) are files which describe the capabilities of CANopen nodes and are therefore central to CANopen. EDS files are most commonly used when CANopen modules are sold or made available to third parties. They provide a standardized and easy to use format for describing how CANopen nodes can be integrated into networks. They can also serve as an in-house documentation of the node.

CANopen Architect allows quick and easy generation and editing of EDS files. Files can be built from scratch or based on one of the versions in the included library. The user interface presents the contents of the EDS file in a tree view, allowing quick and easy editing of any aspect.

CANopen Architect also supports the creation and editing of Device Configuration Files (DCF). DCFs are the same as EDS files but contain real settings from a specific node. Unlike many other EDS editors, CANopen Architect supports the editing of data that is specific to DCFs.

## 3.10. Canopen Communication objects

A **SDO** (Service Data Object) is providing direct access to object entries of a CANopen device's object dictionary. As these object entries may contain data of arbitrary size and data type.

SDOs may be used to transfer multiple data sets (each containing an arbitrary large block of data) from a client to a server and vice versa. The client shall control via a multiplexer (index and sub-index of the object dictionary) which data set shall be transferred. The content of the data set is defined within the object dictionary.

A **PDO** (Process Data Object) is providing real-time data transfer of object entries of a CANopen device's object dictionary. The transfer of PDO is performed with no protocol overhead. The PDO correspond to objects in the object dictionary and provide the interface to the application objects. Data type and mapping of application objects into a PDO is determined by a corresponding PDO mapping structure within the object dictionary.

## 3.11. Process Data Object (PDO)

- High priority real-time data exchange
- Broadcast
- Non-acknowledged
- No protocol overhead
- Synchronous / asynchronous, cyclic/acyclic, event driven transmission
- Maximum 8 bytes of data
- The PDO is used for normal process data communication, the primary purpose of the CANopen network.

## 3.12. Service Data Object (SDO)

- Access to object dictionary
- Peer-to-peer communication
- Acknowledged service
- Low priority
- Transfer of domains
- The SDO is used for configuration and diagnostic access to a device. It is not as fast as PDO but provides the most flexible means to read/write any amount of data.

### 3.12.1. CANopenCommunication Channels

CANopen uses three relationships between the network nodes

- Master/slave relationship
- Client/server relationship
- Producer/consumer relationship

## 3.13. Master/slave relationship

In a master/slave relationship, a master controls the message traffic and the slaves only respond to master requests. Messages can be exchanged on an unacknowledged or acknowledged basis. An

unacknowledged message can be received by all nodes, single nodes or no node. For an acknowledged message, the master requests a message from the slave. The slave responds to the frame with the requested data.

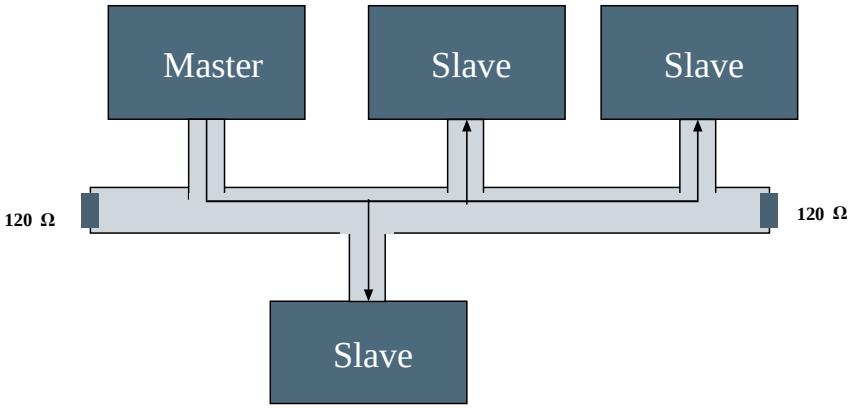



*Figure 18: CANopen Master/slave model [12]*

### 3.13.1. Client/server relationship

A client/server relationship is always established between two nodes and is bidirectional. The exchange of messages is always initiated by the client. It makes a request to the server and expects an acknowledgment (that normally contains the response data). Therefore, a client/server relationship always has at least two frames (request/response).

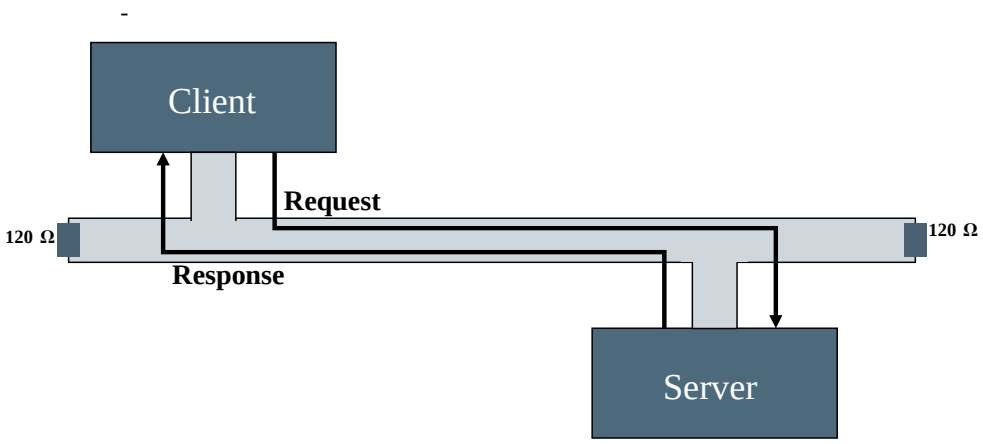



*Figure 19: CANopen Client/server model [12]*

### 3.13.2. Producer/consumer relationship

A producer/consumer relationship is used where quick data exchange without management data is required. The producer sends a frame that can be received by one or more nodes (consumers).

To avoid unnecessary reduction of the bus bandwidth, data transfer is unacknowledged. This broadcast connection is exactly like the CAN protocol's broadcast capability. The Producer takes the initiative during the unconfirmed relationship. The Consumer takes the initiative during the confirmed relationship (N Producers, M Consumers)

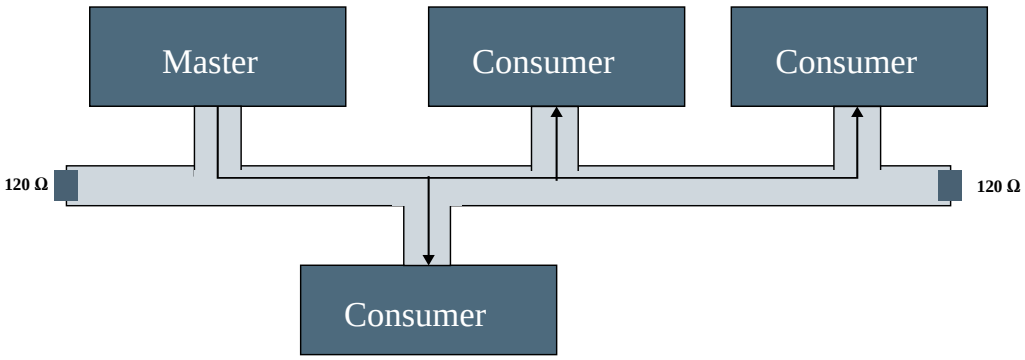



*Figure 20: CANopen Producer/consumer model [12]*

## 3.14. CANopen Communication Protocols

The Communication Unit (CU) uses the OD to set and get configuration settings, inform the device application about new received process data and transmit the device process data to the CAN network. For the different requirements of these activities, different CANopen protocol elements are specified. The CU takes care of all necessary operations to be compliant with the CANopen standard. The following table shows an overview of the UC modules and their responsibility:

- Network Management (NMT) protocol
- Service Data Object (SDO) protocol
- Process Data Object (PDO) protocol
- Synchronization Object (SYNC) protocol
- Time Stamp Object (TIME) protocol
- Emergency Object (EMCY) protocol
- Layer Setting Services (LSS) protocol

### 3.14.1. Network Management (NMT) protocol

The Network Management (NMT) Objects perform tasks that can be divided into two groups:

- Device control services to
  - initialize the network and network nodes
  - control the operating states of the nodes.
- Connection monitoring services to monitor the nodes during network operation.

- Node Guarding function
- Heartbeat function

The network management state machine gives an overview of the possible operating states and illustrates the correlations:

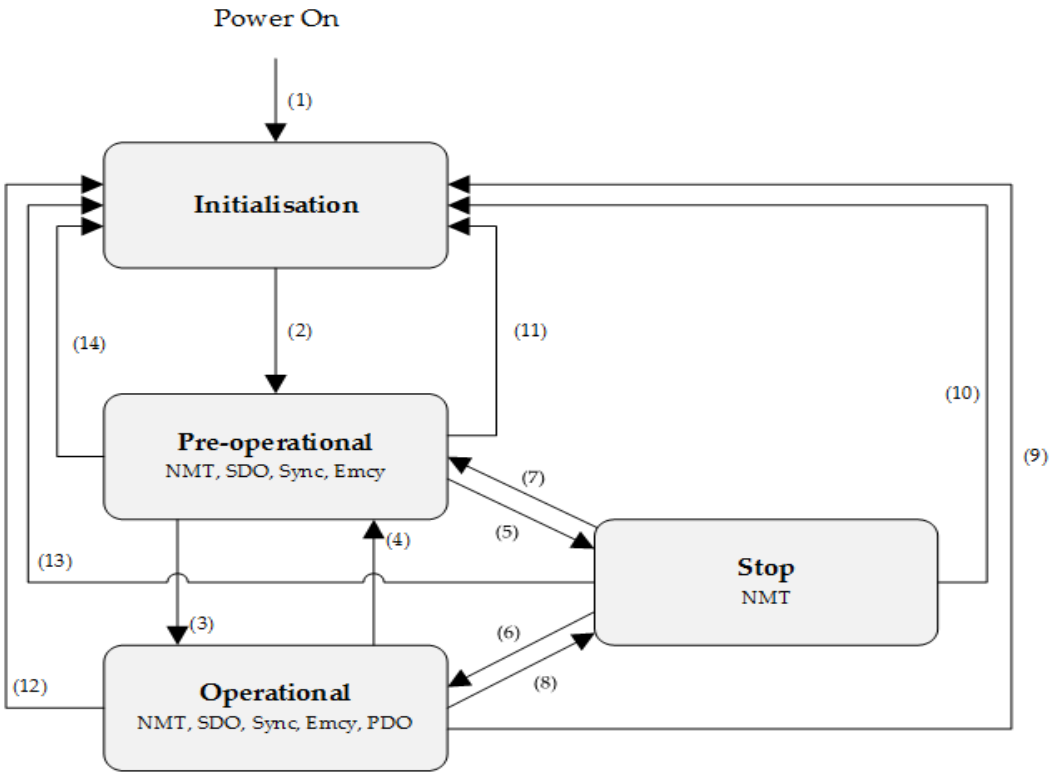



*Figure 21: NMT State Machine [13]*

*Table 3: Transition table of NMT [13]*

| Transition | Event |
|---|---|
| (1) | After power on the system goes directly to *initialization* state |
| (2) | Once *initialization* is completed the system enters to *Pre-operational* state |
| (3), (6) | Reception of *Start remote node* command |
| (4), (7) | Reception of *Enter pre-operational state* command |
| (5), (8) | Reception of *Stop remote node* command |
| (9), (10), (11) | Reception of *Reset node* command |

| | |
|---|---|
| (12), (13), (14) | Reception of *Reset communication* command |

The NMT state machine defines the communication behaviour of a CANopen device. The CANopen NMT state machine consists of an Initialization state, a Pre-operational state, an Operational state, and a Stopped state. After power-on or reset, the device enters the Initialization state.

After the device initialization is finished, the device automatically transits to Pre-operational state and indicates this transition by sending the boot-up message. This way the device indicates that it is ready to work.

A device that stays in Pre-operational state can start to transmit SYNC-, Time Stamp- or Heartbeat messages if these services are supported and configured in the right way. In contrast to PDO communication that must be disabled in this state, the device can communicate via SDO. PDO communication is only possible in the Operational state. During Operational state, the device can use all supported communication objects. A device that was switched to the Stopped state only reacts to receive NMT commands. In addition, the device indicates the current NMT state by supporting the error control protocol during Stopped state.

### *NMT state Initialization*

To ensure a controlled network start and to monitor the connections of the nodes, the devices are initialized, and the operating modes are controlled in a master/slave relationship.

These services are transmitted unidirectionally with COB-ID 0 and are therefore assigned the highest priority on the bus.

The data frame consists of two bytes.

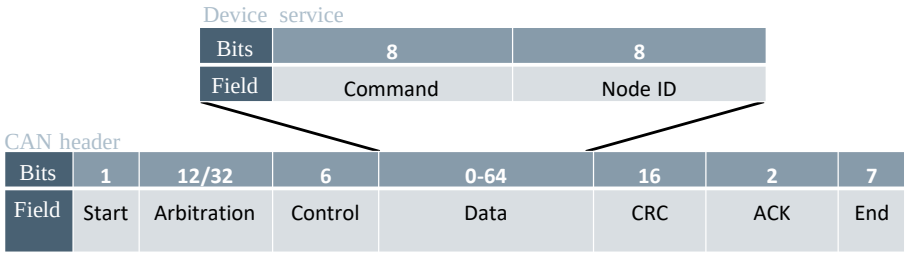



*Figure 22: CANopen NMT frame structure [13]*

*Table 4: NMT Command Specifier to request a specific state of the device [13]*

| Section | Description |
|---|---|

| | | |
|---|---|---|
| **Command** | The “Command Specifier” field defines the NMT service to be used. Possible values: | |
| | 01 h | Start network node |
| | 02 h | Stop network node |
| | 80 h | Go to “Pre-operational” |
| | 81 h | Reset node |
| | 82 h | Reset communication |
| **Node ID** | This 1-byte field addresses the receiver of the NMT message with the node ID. A message with node ID 0 addresses all NMT slaves. | |

### NMT state pre-operational

In Pre-Operational state, the communication using SDO messages is possible. PDO message are not yet defined and therefore communication using these messages is not allowed. The device will pass to Operational message after receiving an NMT start node command. Normally the master puts a node in Pre-Operational state during the set-up and configuration of device parameters.

### NMT state operational

In Operational state all kind of messages are active, even PDO messages.

### NMT state stopped

When entering in Stopped state, the device is forced to stop all communications except for the NMT commands. (Node Guarding & Life Guarding).

### NMT states and communication object relation

Following table shows the relation between communication states and communication objects. Services on the listed communication objects may only be executed if the devices involved in the communication are in the appropriate communication states

## 3.14.2. Monitoring functions

CANopen provides monitoring services for the network connections to allow response to a failure of a node or network interrupts. The following mechanisms are available to secure communication:

- Guarding
    - Node Guarding (master)
    - Life Guarding (slave)

- Heartbeat

A CANopen node must support at least one monitoring function. Monitoring frames can be identified by COB-ID 700h + node ID.

### 3.14.3. Guarding function

With Node Guarding, it is the master's responsibility to cyclically request NMT status messages from the slaves. If a slave does not respond within a defined time or sends an unexpected operating state, the master detects an error and sends an EMCY Object.

Slaves that support Life Guarding can also monitor the cyclic request frame from the master. If a frame from the master is not received within a defined time, the slave detects an error and sends an error frame.

The times are configured using two Network Management Objects: "Guard Time" (100Ch) and "Life Time Factor" (100Dh).

| | | | | | | |
|---|---|---|---|---|---|---|
| 100Ch | Guard time | 00h | Used together with "Life time factor" to decide the node life-time in (ms) | U16 | RW | C |
| 100Dh | Life time factor | 00h | If the node has not been guarded within its lifetime ("Life time factor"*"Guard time"), an error event is logged and a remote node error is indicated | U8 | RW | C |

*Figure 23: Configured times for guard and lifetime [37]*

Guarding frames are always sent bidirectionally following the master-slave principle.

Without Life Guarding, a failure of the NMT master is not detected by the NMT slaves.

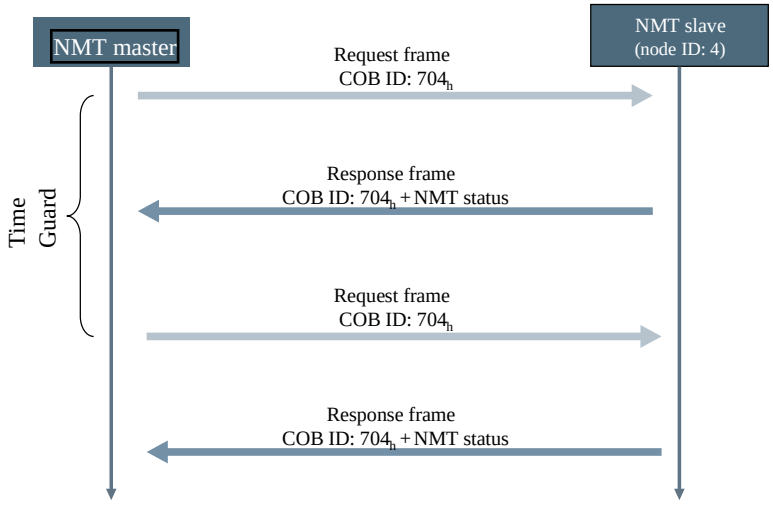



*Figure 24: NMT master slave chart [37]*

### 3.14.4. Heartbeat function

The CANopen network management uses the Heartbeat message as the confirmation of the NMT command sent by the NMT master device. Additionally, any CANopen device can use it, to check the availability of any other CANopen device. This is necessary in all cases, in which CANopen devices transmit PDO messages only on state-of-change events. The subscriber of such PDO messages could not know, if there is no event or if the device is not more available. The reception of the Heartbeat message of the related device indicates that it is still alive.

The transmission period is configurable by means of the heartbeat-producer-time parameter in the object dictionary of the transmitting device. The indication that a device is missing can be configured in any interested device by means of the heartbeat-consumer-time array. As a rule of thumb, the consumer time should twice of the producer time.

In some applications, CANopen devices crosscheck each other by consuming the Heartbeat messages. Sensors normally do not consume Heartbeat messages, they just produce their PDO messages and their Heartbeat messages to indicate that they are still alive respectively to confirm the NMT command

With heartbeat, the nodes (NMT master and NMT slaves) cyclically transmit their current operating state to all other bus nodes on their own initiative. An explicit request via the NMT master is not required. The individual cycle time of each node can be set using object $1017_h$.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1017h | Producer Heart-beat Time | 00h | Defines the cycle time of the heartbeat. Not used if 0 | U16 | RW | T m |

Nowadays, Heartbeat is preferred over Node Guarding as, due to the missing request frame, it causes less bus load and, in addition, a failure of the NMT master is also detected by the NMT slaves.

The 1-byte Heartbeat message uses a CAN data frame with the ID $700_{16}$ plus producer node-ID. The one-byte content indicates the NMT status. This means, the consumer can also detect that a device does not sent PDO messages because of its NMT state. This can be evaluated in the consumer's application program.

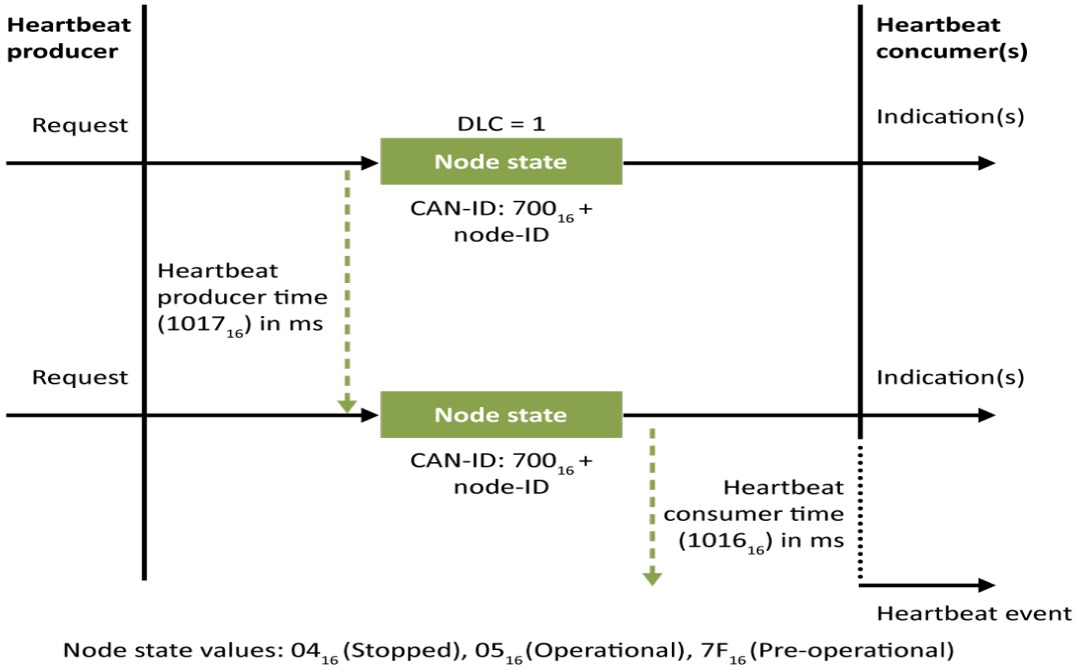



*Figure 25: Heartbeat protocol [14]*

### 3.14.5. Service Data Object (SDO) Protocol

Service data objects (SDOs) enable access to all entries of a CANopen object dictionary. One SDO consists of two CAN data frames with different CAN-Identifiers. This is a confirmed communication service. With an SDO, a peer-to-peer client-server communication between two CANopen devices can be established on the broadcast medium CAN. The owner of the accessed object dictionary acts as a server of the SDO. The device that accesses the object dictionary of the other device is the SDO client.

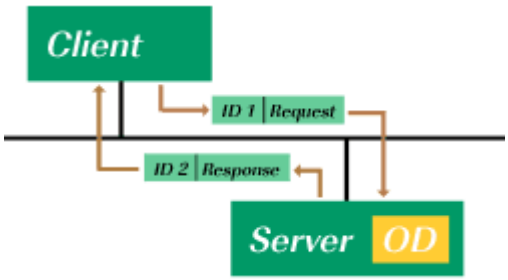



*Figure 26: Working of SDO protocol [36]*

### 3.14.6. Variants of the SDO protocol

A CANopen device can support different variants of the SDO protocol:

- The expedited transfer,
- The normal (segmented) transfer, or
- The block transfer.

During this initiation, the client device indicates which information is going to be accessed from the server's object dictionary, which SDO type is used, and if the information is to be read or written. The server device acknowledges the inquiry and the client device then starts to transmit the first data segment. The normal transfer allows to communicate any amount of data in a segmented way. Each segment can carry up to seven byte of application data, as one byte of the CAN frame is required for protocol information. In normal SDO transfer (see figure) an unlimited number of segments and therefore of application data can be exchanged.

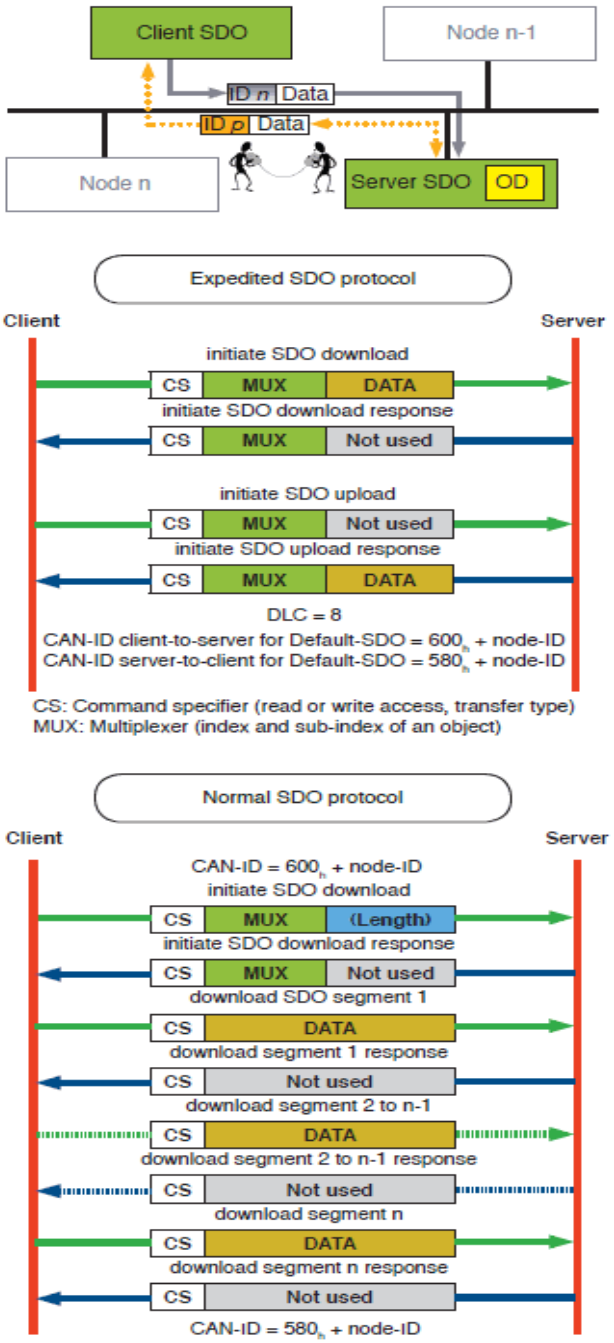



*Figure 27: Working of normal, expedited and block transfer variant of the SDO protocol [36]*

To speed up an SDO transfer of small amounts of data (less or equal 4 byte), the expedited SDO transfer can be used. In such an SDO connection, the data is directly transferred during the initiation of the SDO connection. Speeding up the transmission of huge amounts of data can be achieved by supporting the SDO block transfer. During the block transfer, no single data segment (like during the normal transfer) but only a block of data (up to 127 segments) is confirmed by the receiver.

Device manufacturers should base their choice of SDO variant on the amount of data that a device is going to communicate.

### 3.14.7. SDO parameter set

The SDO parameter sets are arranged in the object dictionary index range $12xx_h$. The SDO server channels are described in the range from $1200_h$ to $127F_h$. The parameter sets of the client channels must be provided in the range from $1280F_h$ to $12FF_h$. An SDO parameter set contains the two communication object identifiers (COB-IDs) and the node-ID of the related communication partner. The COB-ID entries cover the CAN-Identifiers of the CAN frames used to transmit information in the direction "server to client" and vice versa.

| Index | Sub-Index | Description |
|---|---|---|
| $12xx_h$ | $00_h$ | Number of entries |
| | $01_h$ | COB-ID client-to-server |
| | $02_h$ | COB-ID server-to-client |
| | $03_h$ | Node-ID of server/client |

*Figure 28: SDO parameter set [36]*

### 3.14.8. Process data objects (PDOs)

Process data objects (PDOs) are used in CANopen for broadcasting high-priority control and status information. A PDO consists of a single CAN frame and communicates up to 8 byte of pure application data. Device designers must evaluate the amount of process data that the device needs to receive and transmit. Based on the result of this evaluation process, they must provide the related amount of receive and transmit PDOs within the device.

### 3.14.9. PDO parameter sets

In case a device is supposed to support the transmission/reception of a PDO, the corresponding parameter sets for this PDO have to be provided in the object dictionary of that device. A single PDO requires a set of communication parameters (PDO communication parameter record) and a set of mapping parameters (PDO mapping record).

Among others, the communication parameters indicate the CAN-Identifier that is used by this PDO, and the triggering event that prompts the transmission of the related PDO. The mapping parameters indicate which information of the local object dictionary is supposed to be transmitted and where the received information is to be stored.

The communication parameters receive PDOs are arranged in the index range $1400_h$ - $15FF_h$ and for transmit PDOs in the range $1800_h$ - $19FF_h$. The related mapping entries are managed in the index ranges $1600_h$ -$17FF_h$ and $1A00_h$ - $1BFF_h$.

### 3.14.10. PDO triggering events

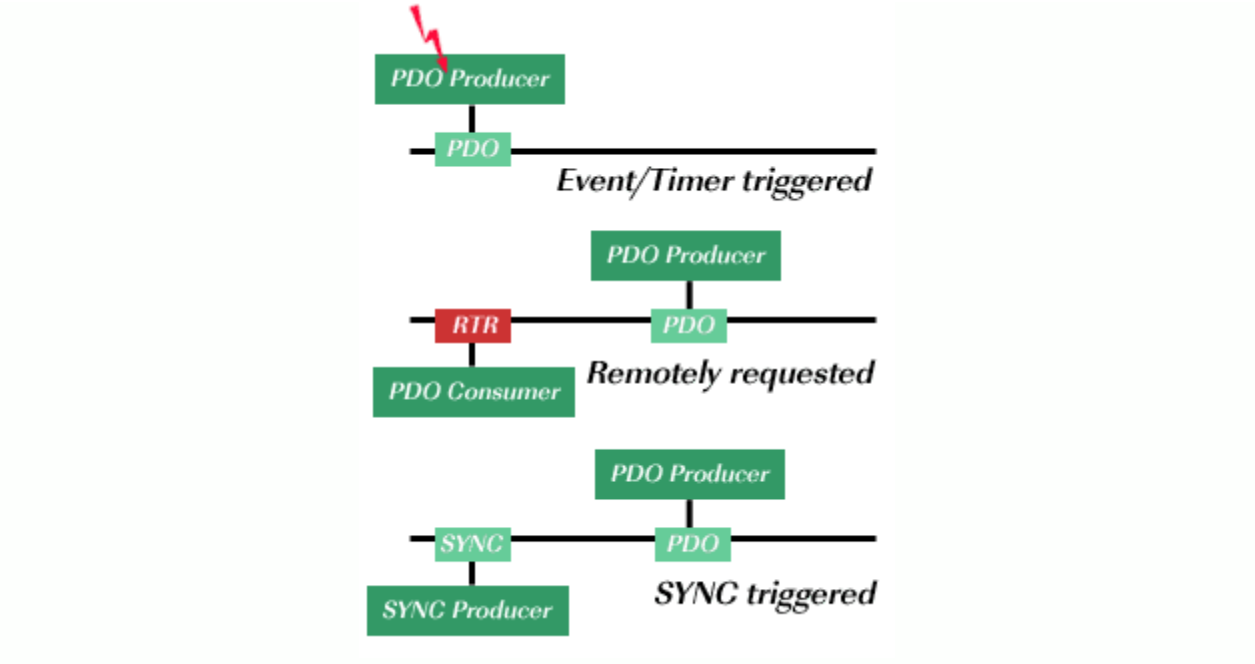



*Figure 29: Working of PDO protocol [16]*

The following triggering events for PDOs are defined:

**Event- or timer-driven:** A device-internal event triggers the PDO transmission (e.g. when the temperature value exceeds a certain limit; event-timer elapses; etc.).

**Remotely-requested:** As PDOs consist of a single CAN data frame, they can be requested via remote transmission request (RTR).

**Synchronous transmission (cyclic):** The transmission of the PDO can be coupled to the reception of the SYNC message.

**Synchronous transmission (acyclic):** These PDOs are triggered by a defined device-specific event but transmitted with the reception of the next Sync message.

### 3.14.11. PDO mapping

PDO mapping defines which application objects are transmitted within a PDO. It describes the sequence and length of the mapped application objects. The mapping of the application objects is described in the related CANopen object dictionary entries for each PDO. CANopen distinguishes between three types of PDO mapping:

**Static PDO mapping:** In case static mapping is supported for a PDO, the content of the PDO is strictly predefined by the device manufacturer and cannot be changed via the CANopen interface.

**Variable PDO mapping:** Variable PDO mapping describes that the mapping entries of a PDO can be changed during NMT pre-operational state.

**Dynamic PDO mapping:** A device supports dynamic mapping if the PDO mapping entries in the CANopen object dictionary can be changed during NMT operational state as well.

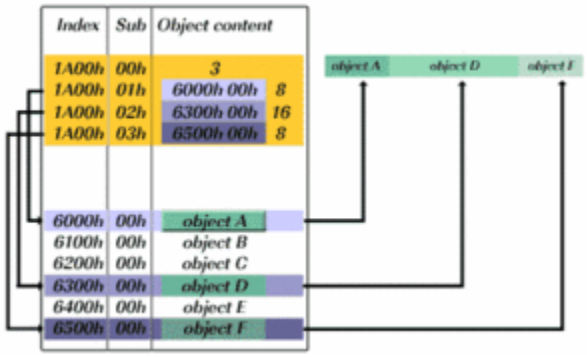



*Figure 30: PDO mapping [16]*

### 3.14.12. Synchronization Object (SYNC) Protocol

The SYNC protocol provides the basic network synchronization mechanism. The SYNC producer triggers the synchronization object (SYNC) periodically. The transmission period of this SYNC protocol is configurable. Any CANopen FD device can be configured as SYNC consumer. The purpose for which the SYNC protocol is used, is manufacturer-specific. In many cases, the SYNC protocol is used for busload management purposes. In such use cases, TPDOs in the SYNC consumers are triggered by the reception of a pre-configured number of SYNCs. To establish an explicit relationship between the current SYNC cycle and PDO transmissions, the SYNC start value (sub-index 06h in the TPDO communication parameters) is used. The current SYNC cycle is provided in the one-byte SYNC counter value.

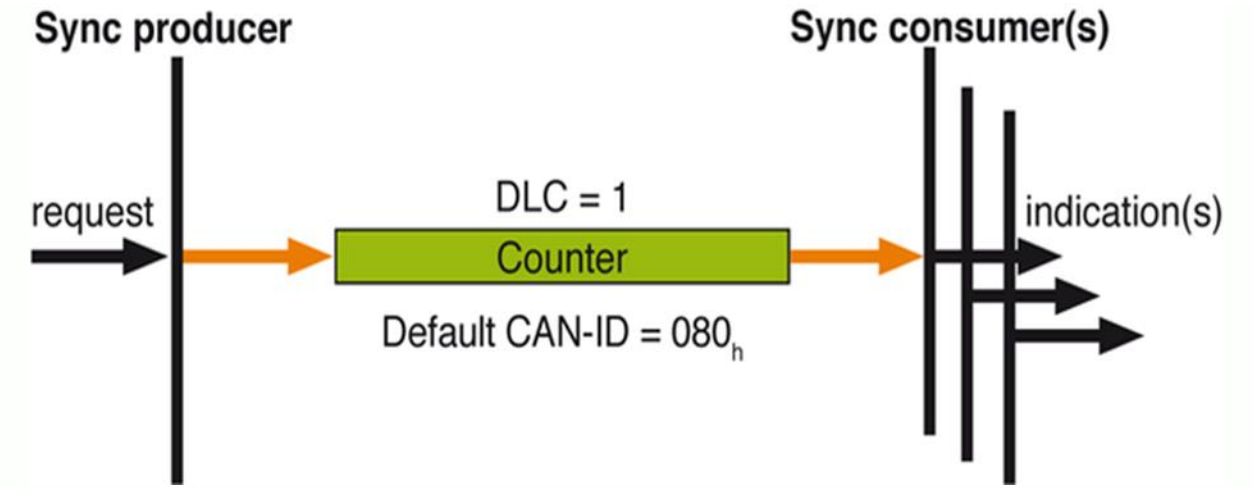



*Figure 31: Working of SYNC protocol [17]*

The SYNC counter is incremented by 1 with every SYNC transmission. The initial value of the counter is 1. The maximum value is configurable in the data object synchronous counter overflow $1019_h$. In case the maximum value is reached the SYNC, counter is set to 1 with the next transmission. The value of the counter is reset to 1 if the CANopen FD device leaves the NMT state stopped.

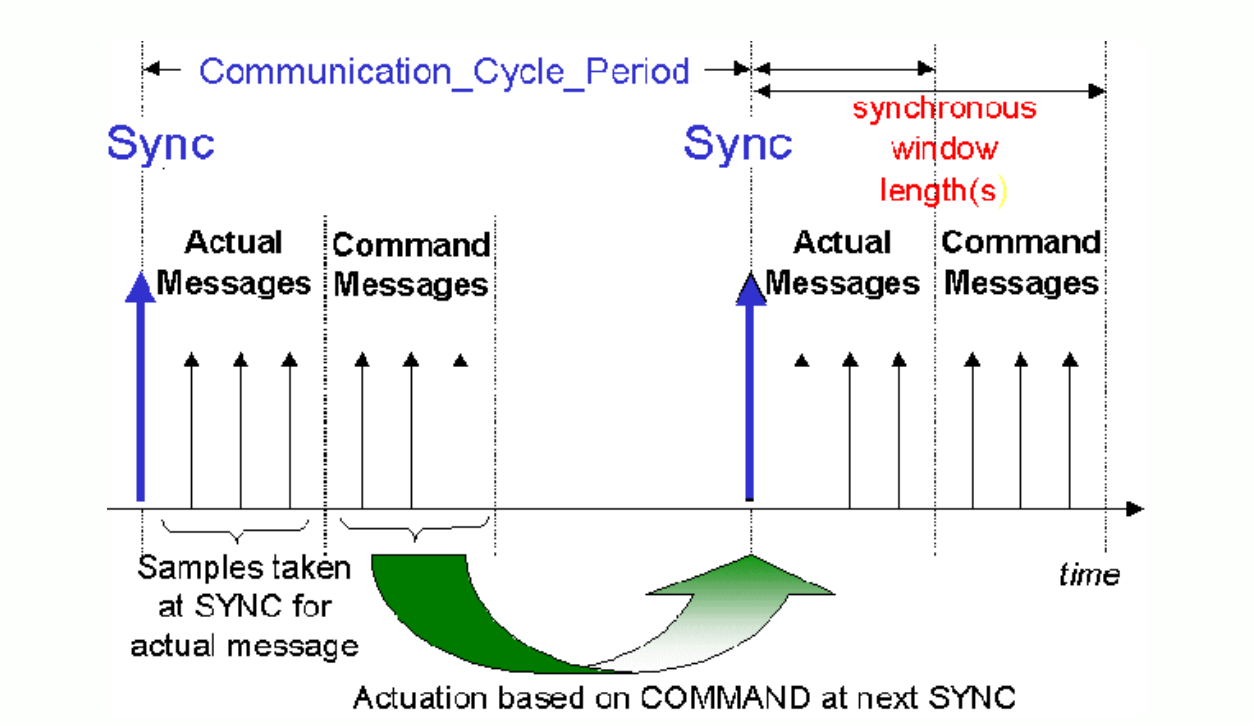



*Figure 32: Timing overview of SYNC protocol [17]*

It is not only for drive applications that it is worthwhile to synchronise the determination of the input information and the setting the outputs. For this purpose, CANopen provides the SYNC object, a CAN telegram of high priority but containing no user data, whose reception is used by the synchronised nodes as a trigger for reading the inputs or for setting the outputs.

### 3.14.13. Time Stamp Object (TIME) Protocol

The Time-stamp protocol enables the user of CANopen systems to adjust a unique network time. The Time-stamp is mapped to one single CAN frame with a data length code of 6 byte. These six data bytes provide the information "Time of Day". This information is given as milliseconds after midnight (Datatype: Unsigned28) and days since January 1, 1984 (Datatype: Unsigned16). The associated CAN frame has by default the CAN-Identifier 100h.

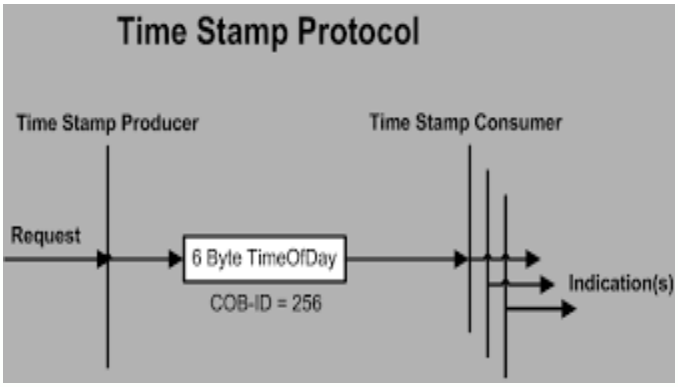



*Figure 33: Working of Time stamp Protocol [15]*

Sometime critical applications especially in large networks with reduced transmission rates require very accurate synchronization; it may be necessary to synchronize the local clocks with an accuracy in the order of microseconds. This is achieved by using the optional high-resolution synchronization protocol which employs a special form of timestamp message to adjust the inevitable drift of the local clocks.

The high-resolution timestamp is encoded as unsigned32 with a resolution of 1 microsecond which means that the time counter restarts every 72 minutes. It is configured by mapping the high-resolution time-stamp (object 1013h) into a PDO.

### 3.14.14. Emergency Object (EMCY) Protocol

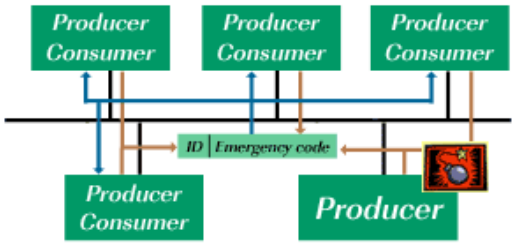



*Figure 34: Working of Emergency object protocol [15]*

The Emergency messages are triggered by a device-internal error. The Emergency message, transmitted by the Emergency producer, is mapped to one single CAN frame, which covers up to eight bytes of data. The data content is defined as a 1-byte Error register (Index $1001_h$ of the local object dictionary), a 16-bit Emergency error code, and up to 5 byte of manufacturer-specific error information. By default, a device that supports the Emergency producer functionality assigns the CAN-Identifier $80_h$ + (node-ID) to the Emergency message.
An Emergency message is transmitted only once per error event. If no new errors occur on a device, no further emergency messages are transmitted. Zero or more Emergency consumers may receive these messages and may initiate suitable, application-specific counter measures.

### 3.14.15. Layer Setting Services (LSS) Protocol

LSS distinguishes between an LSS master (typically residing in the host controller) and the LSS slaves. LSS enables the LSS master to modify the LSS slaves' CANopen node-ID and to switch the entire network from one data rate to another. LSS utilizes exactly two CAN frames. The CAN data frame $7E5_h$ carries the command from the LSS master to one or several LSS slaves. The CAN frame $7E4_h$ is used to provide the response(s) to the LSS master. LSS is specified in the document CiA 305**.**

#### CANopen node-ID assignment via LSS

The entire 128 Bit of the Identity object $1018_h$ (vendor-ID, product-code, revision number, serial number) are called the LSS address. The LSS address allows the LSS master to differentiate between the LSS slaves, even if one or several LSS slaves do not own a valid CANopen node-ID. Via the LSS switch state selective command, the LSS slaves forces exactly one LSS slave to enter the LSS configuration state. Only in this state the LSS slave accepts a new CANopen node-ID that is proposed by the LSS master. It is the LSS master's task to ensure that during the assignment of the CANopen node-ID, there is only one LSS slave in the LSS configuration state.

In case the LSS addresses of the LSS slaves are unknown to the LSS master, the LSS master may have several means to detect the LSS addresses. In case the LSS slaves own already a valid CANopen node-ID, the LSS master just reads the content of the object $1018_h$ of all LSS slaves in the system, via SDO. In case the LSS slaves do not own a valid CANopen node-ID the LSS master has to rely on additional LSS services. The LSS fastscan service enables the LSS master to scan via bit masks, whether there exists an unconfigured LSS slave, that's LSS address is in a given LSS address range. By executing several scanning cycles, the LSS master can identify exactly one unconfigured LSS slave and can provide subsequently a valid CAN open node-ID.

## 3.15. Serial Peripheral Interface

The Serial Peripheral Interface (SPI) is a synchronous serial communication interface specification used for short-distance communication, primarily in embedded systems. The interface was developed by Motorola in the mid-1980s and has become a de facto standard. Typical applications include Secure Digital cards and liquid crystal displays.

SPI devices communicate in full duplex mode using a master-slave architecture usually with a single master (though some Atmel devices support changing roles on the fly depending on an external (SS) pin). The master (controller) device originates the frame for reading and writing. Multiple slave-devices may be supported through selection with individual chip select (CS), sometimes called slave select (SS) lines.

Sometimes SPI is called a four-wire serial bus, contrasting with three-, two-, and one-wire serial buses. The SPI may be accurately described as a synchronous serial interface, but it is different from the Synchronous Serial Interface (SSI) protocol, which is also a four-wire synchronous serial communication protocol. The SSI protocol employs differential signalling and provides only a

single simplex communication channel. For any given transaction SPI is one master and multi slave communication

### 3.15.1. Interface

The SPI bus specifies four logic signals:

- SCLK: Serial Clock (output from master)
- MOSI: Master Out Slave In (data output from master)
- MISO: Master In Slave Out (data output from slave)

- CS /SS: Chip/Slave Select (often active low, output from master to indicate that data is being sent)

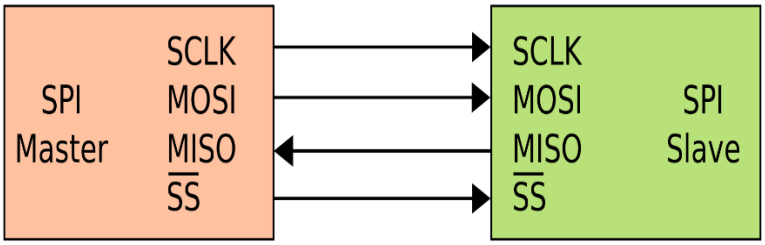



*Figure 35: Basic Spi configuration with master and slave [18]*

The signal names above can be used to label both the master and slave device pins as well as the signal lines between them in an unambiguous way and are the most common in modern products. Pin names are always capitalized e.g. "Chip Select," not "chip select."

Many products can have nonstandard SPI pin names:

Serial Clock:

- SCK

Master Output → Slave Input (MOSI):

- SIMO, MTSR - correspond to MOSI on both master and slave devices, connects to each other
- SDI, DI, DIN, SI - on slave devices; connects to MOSI on master, or to below connections
- SDO, DO, DOUT, SO - on master devices; connects to MOSI on slave, or to above connections

Master Input ← Slave Output (MISO):

- SOMI, MRST - correspond to MISO on both master and slave devices, connects to each other

- SDO, DO, DOUT, SO - on slave devices; connects to MISO on master, or to below connections
- SDI, DI, DIN, SI - on master devices; connects to MISO on slave, or to above connections

Slave Select:

- SS, SS, SSEL, nSS, /SS, SS# (slave select)
- CS, CS (chip select)
- CSN (chip select/enable)
- CE (chip enable)

## 3.16. How SPI works

### 3.16.1. Clock

The clock signal synchronizes the output of data bits from the master to the sampling of bits by the slave. One bit of data is transferred in each clock cycle, so the speed of data transfer is determined by the frequency of the clock signal. SPI communication is always initiated by the master since the master configures and generates the clock signal.

Any communication protocol where devices share a clock signal is known as *synchronous.* SPI is a synchronous communication protocol. There are also asynchronous methods that don't use a clock signal.

For example, in UART communication, both sides are set to a pre-configured baud rate that dictates the speed and timing of data transmission.

The clock signal in SPI can be modified using the properties of *clock polarity* and *clock phase*. These two properties work together to define when the bits are output and when they are sampled. Clock polarity can be set by the master to allow for bits to be output and sampled on either the rising or falling edge of the clock cycle. Clock phase can be set for output and sampling to occur on either the first edge or second edge of the clock cycle, regardless of whether it is rising or falling.

### 3.16.2. Slave select

The master can choose which slave it wants to talk to by setting the slave's CS/SS line to a low voltage level. In the idle, non-transmitting state, the slave select line is kept at a high voltage level. Multiple CS/SS pins may be available on the master, which allows for multiple slaves to be wired in parallel. If only one CS/SS pin is present, multiple slaves can be wired to the master by daisy-chaining.

### 3.16.3. Multiple slaves

SPI can be set up to operate with a single master and a single slave, and it can be set up with multiple slaves controlled by a single master. There are two ways to connect multiple slaves to the master.

If the master has multiple slave select pins, the slaves can be wired in parallel like this:

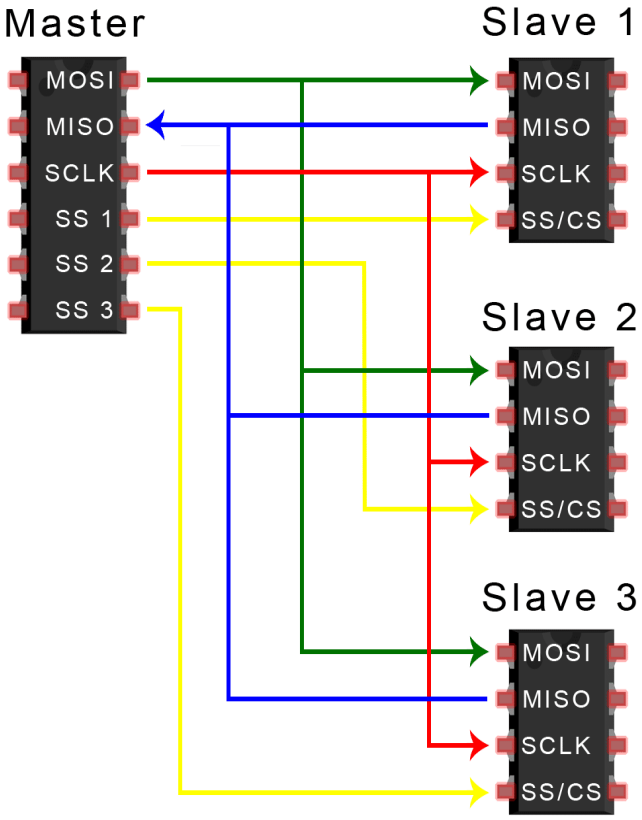



*Figure 36: Multiple slaves SPI [19]*

If only one slave select pin is available, the slaves can be daisy-chained like this

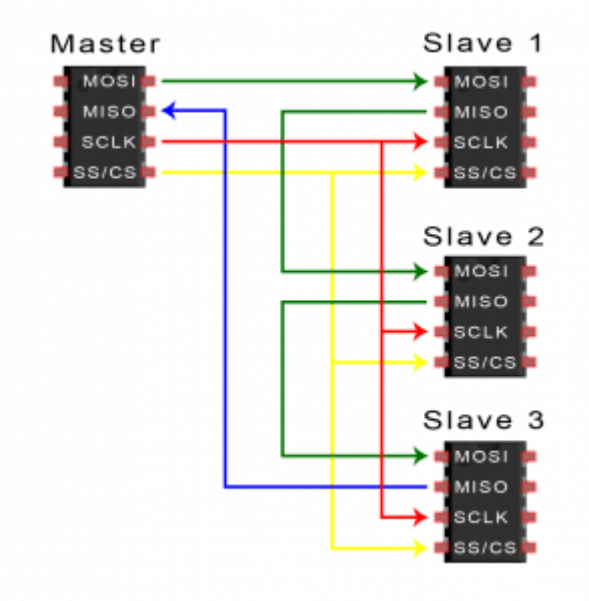



*Figure 37: Single Slave SPI [19]*

### 3.16.4. MOSI and MISO

The master sends data to the slave bit by bit, in serial through the MOSI line. The slave receives the data sent from the master at the MOSI pin. Data sent from the master to the slave is usually sent with the most significant bit first.

The slave can also send data back to the master through the MISO line in serial. The data sent from the slave back to the master is usually sent with the least significant bit first.

## 3.17. Steps of SPI data transmission

To begin SPI communication, the master must send the clock signal and select the slave by enabling the CS signal. Usually, chip select is an active low signal; hence, the master must send a logic 0 on this signal to select the slave. SPI is a full-duplex interface; both master and slave can send data at the same time via the MOSI and MISO lines respectively.

During SPI communication, the data is simultaneously transmitted (shifted out serially onto the MOSI/SDO bus) and received (the data on the bus (MISO/SDI) is sampled or read in). The serial clock edge synchronizes the shifting and sampling of the data. The SPI interface provides the user with flexibility to select the rising or falling edge of the clock to sample and/or shift the data. Please refer to the device data sheet to determine the number of data bits transmitted using the SPI interface

1. The master outputs the clock signal:

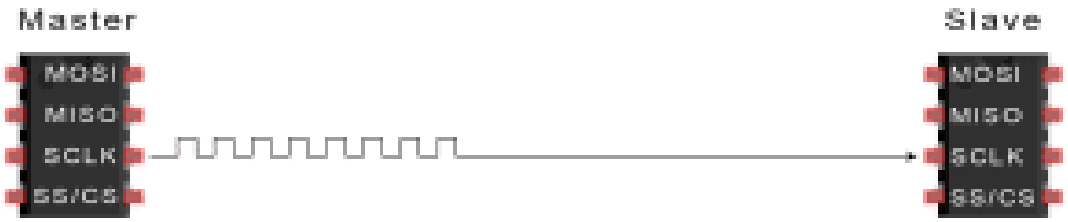



*Figure 38: Master to Slave clock signal [19]*

2. The master switches the SS/CS pin to a low voltage state, which activates the slave:

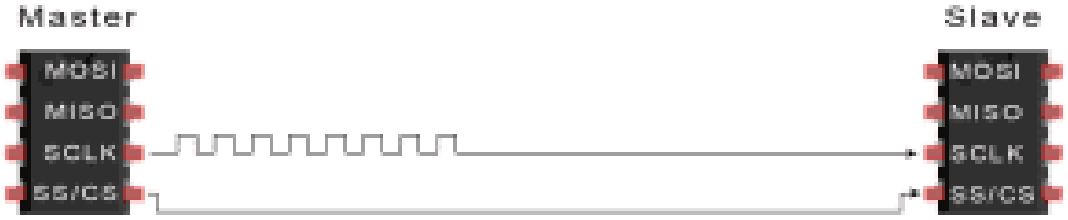



*Figure 39: Master to slave chip select signal [19]*

3. The master sends the data one bit at a time to the slave along the MOSI line. The slave reads the bits as they are received:

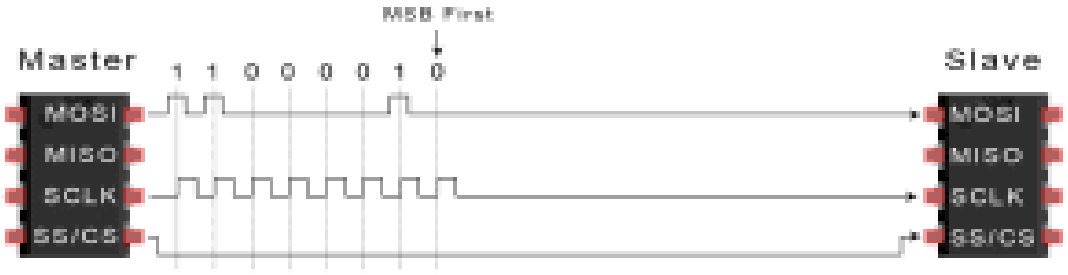



*Figure 40: Data signal transmission [19]*

4. If a response is needed, the slave returns data one bit at a time to the master along the MISO line. The master reads the bits as they are received:

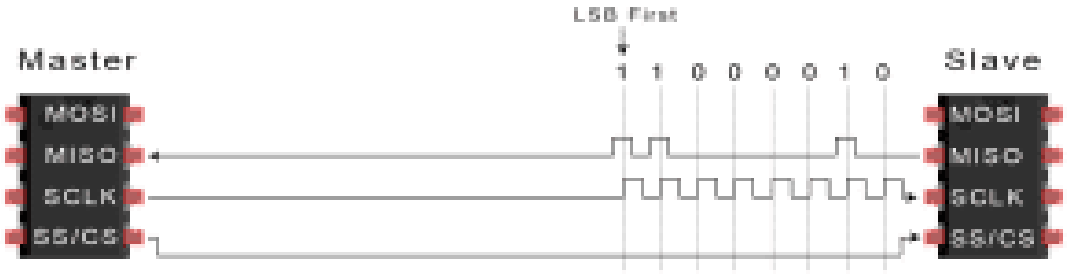



*Figure 41: Data transmission slave to master [19]*

## 3.18. Advantages and disadvantages of SPI

There are some advantages and disadvantages to using SPI, and if given the choice between different communication protocols, you should know when to use SPI according to the requirements of your project:

**ADVANTAGES**

- No start and stop bits, so the data can be streamed continuously without interruption
- No complicated slave addressing system like I2C
- Higher data transfer rate than I2C (almost twice as fast)
- Separate MISO and MOSI lines, so data can be sent and received at the same time

**DISADVANTAGES**

- Uses four wires (I2C and UARTs use two)
- No acknowledgement that the data has been successfully received (I2C has this)
- No form of error checking like the parity bit in UART
- Only allows for a single master

## 3.19. Applications

The board real estate savings compared to a parallel I/O bus are significant, and have earned SPI a solid role in embedded systems. That is true for most system-on-a-chip processors, both with higher end 32-bit processors such as those using ARM, MIPS, or PowerPC and with other microcontrollers such as the AVR, PIC, and MSP430. These chips usually include SPI controllers capable of running in either master or slave mode. In-system programmable AVR controllers (including blank ones) can be programmed using a SPI interface.

Chip or FPGA based designs sometimes use SPI to communicate between internal components; on-chip real estate can be as costly as its on-board cousin.

The full-duplex capability makes SPI very simple and efficient for single master/single slave applications. Some devices use the full-duplex mode to implement an efficient, swift data stream

for applications such as digital audio, digital signal processing, or telecommunications channels, but most off-the-shelf chips stick to half-duplex request/response protocols.

SPI is used to talk to a variety of peripherals, such as

- Sensors: temperature, pressure, ADC, touchscreens, video game controllers
- Control devices: audio codecs, digital potentiometers, DAC
- Camera lenses: Canon EF lens mount
- Communications: Ethernet, USB, USART, CAN, IEEE 802.15.4, IEEE 802.11, handheld video games
- Memory: flash and EEPROM
- Real-time clocks
- LCD, sometimes even for managing image data

# 4. Transmission Control Protocol / Internet Protocol (TCP/IP)

There are seven layers of networking protocols in the Open Systems Intercommunication (OSI). Some of these layers are interrelated to each other. So, these layers are considered as four.

*Table 5: OSI Layer model [20]*

<table>
<tr><th>OSI Layer</th><th>Name</th><th colspan="2">Common Protocols</th></tr>
<tr><td>7</td><td>Application</td><td colspan="2" rowspan="3">HTTP FTP SMTP DNS</td></tr>
<tr><td>6</td><td>Presentation</td></tr>
<tr><td>5</td><td>Session</td></tr>
<tr><td>4</td><td>Transport</td><td>TCP</td><td>SPX</td></tr>
<tr><td>3</td><td>Network</td><td>IP</td><td>IPX</td></tr>
<tr><td>2</td><td>Data Link</td><td colspan="2" rowspan="2">Ethernet</td></tr>
<tr><td>1</td><td>Physical</td></tr>
</table>

In this project, Transmission Control Protocol (TCP) which lies in the Transport Layer of the OSI Model is used for data transmission. The transport layer is responsible for the effective transit of data packets between networks, as its name implies. Using this protocol, most of the data which is not transmitted as a single piece is broken into smaller data packets. These data packets are tagged with a header to identify the correct sequence. The packets do not have to follow the same path across the network. When the packets arrive at the client's end, they are reassembled in the correct order. If a packet fails to arrive, a message is sent to the server's end requesting that the message be sent again. Basically, it is a network protocol that defines how data is sent and received through network adapters, hubs, switches, routers, and other network communications hardware.

Information can be sent in both directions via TCP. Like in a telephone conversation, computer systems or other devices communication over TCP can send and receive data at the same time. The protocol's main transmission units are segments (packets), which can contain control information in addition to user data and are limited to 1,500 bytes in size. Data transmission, establishment, and breakdown of the connections are performed by the TCP software in the network protocol stack of the corresponding operating system.

TCP software is managed via specialized interfaces by various network programs such as web browsers and servers. So that each connection must always be differentiated by two properly defined endpoints (server and client). It is essential that the TCP protocol has a unique pair of “Internet Protocol Address (IP address)” and “Port” for both ends (which is known as “2-tuple” or “socket”) [21].

Each IP address has the ability to open and connect with up to 65535 distinct "ports" for sending and receiving data to and from other network devices. IP Address is just to identify the devices onthe network while Port Number identifies the distinct connection in between two devices.

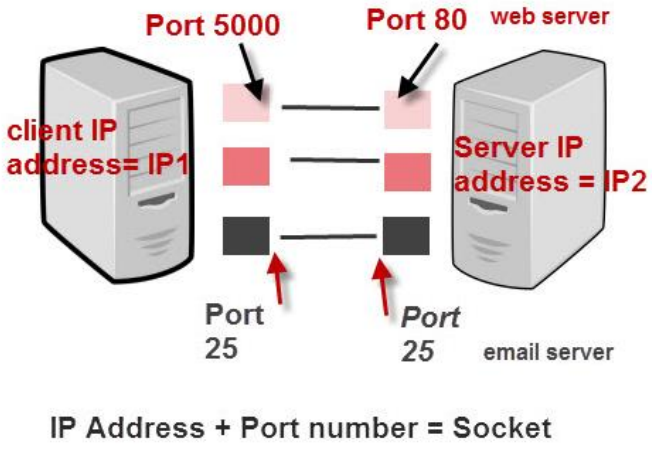



*Figure 42: TCP Server-Client communication network*

Source:[22]

This shows the main concept of TCP server-client connection. In this project, the gateway device is used as the server end, and the PC represent client end.

One of the main advantages of using TCP protocol is automatic error detection. Low-level drivers used for receiving and sending data performs error checking on all data to ensure that no data is delivered or received with errors.

## 4.1. TCP Server-Client Connection

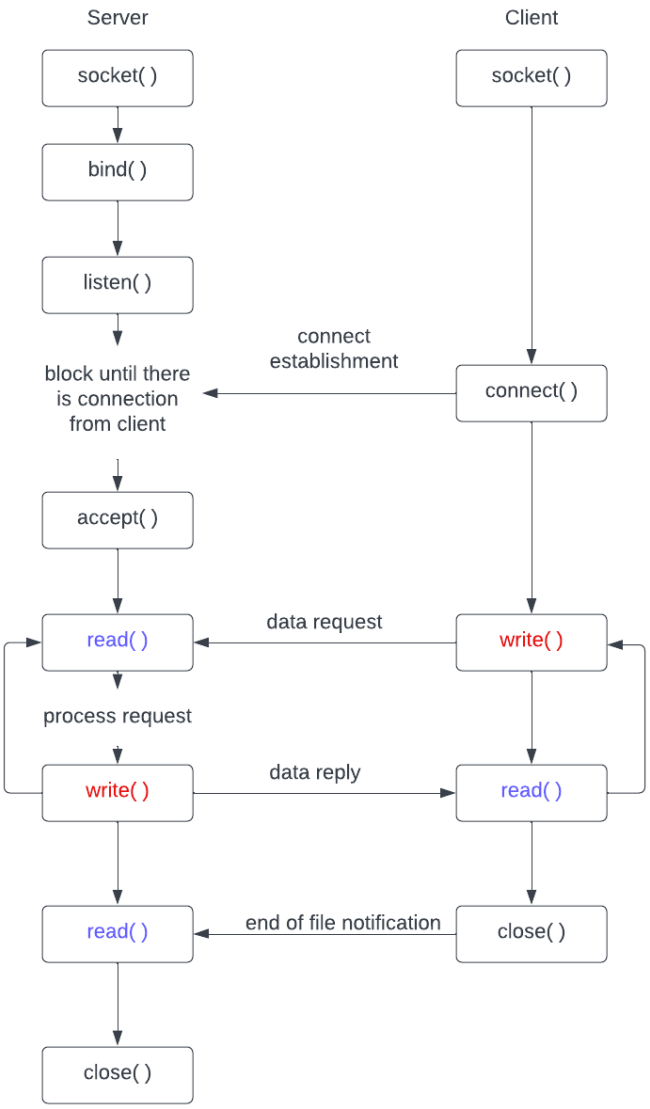



*Figure 43: TCP Client-Server setup connection*

Source: [23]

The client socket (node) has an IP address and listens on a specific port, whereas the server socket (node) reaches out to establish a connection with the Server. The following is a detailed description of the TCP Server implementation:

*socket() : creates an endpoint for communication*

With the creation of endpoint socket() also returns a file descriptor that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

```
int socket(int domain, int type, int protocol);
```

This is the syntax for the creation of the socket. The *domain* argument specifies a communication domain; this selects the protocol family which will be used for the communication. These families are defined in <*sys/socket.h*> header file. The socket has the indicated *type*, which specifies the communication semantics e.g.: *SOCK_STREAM* for TCP (reliable, connection-oriented) and *SOCK_DGRAM* for UDP (unreliable, connectionless). The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family in case *protocol* can be specified as 0, [24].

*bind( ) : bind a name to a socket*

When a socket is created with *socket*( ), it exists in a namespace (address family) but has no address assigned to it. *bind*( ) assigns the address specified by *addr* to the socket referred to by the file descriptor *sockfd*. *addrlen* specifies the size, in bytes, of the address structure pointed to by *addr*.

```
int bind(int sockfd, const struct sockaddr *addr,
socklen_t addrlen);
```

The purpose of this structure is to cast the structure pointer passed in *addr* in order to avoid compiler warnings.

*listen( ) : listen for connections on a socket*

*listen*( ) marks the socket referred to by *sockfd* as a passive socket, that is, as a socket that will be used to accept incoming connection requests using *accept*( ). The sockfd argument is a file descriptor that refers to a socket of type *SOCK_STREAM* or *SOCK_SEQPACKET*.

```
int listen(int sockfd, int backlog);
```

The *backlog* argument defines the maximum length to which the queue of pending connections for *sockdf* may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED or, if the underlying protocol supports retransmission, the request may be ignored so that later reattempt at connection succeeds [25].

*accept( ) : accepts a connection on a socket*

The *accept*( ) system call is used with connection-based socket types (*SOCK_STREAM*, *SOCK_SEQPACKET*). It extracts the first connection request on the queue of pending connections for the listening socket, *sockfd*, creates a new connected socket, and returns a new file descriptor

referring to that socket. The newly created socket is not in the listening state. The original socket *sockfd* is unaffected by this call.

```
int accept(int sockfd, struct sockaddr *restrict addr,
socklen_t *restrict addrlen);
```

The argument *addr* is a pointer to a *sockaddr* structure. This structure is filled in with the address of the peer socket, as known to the communications layer. The exact format of the address returned *addr* is determined by the socket's address family. When *addr* is NULL, nothing is filled in; in this case, addrlen is not used, and should also be NULL.

The *addrlen* argument is a value result argument: the caller must initialize it to contain the size (in bytes) of the structure pointed to by *addr*; on return, it will contain the actual size of the peer address. If no pending connections are present on the queue, and the socket is not marked as nonblocking, *accept*( ) clocks the caller until a connection is present.

# 5. Multithreading

Multithreading is a term used for the techniques or procedures in which multiple applications can run simultaneously. These procedures could be hardware or software based. It helps in improving the performance by increasing the computing speed as more than one application can run at the same time in parallel. The application must be ready for this operation and can be divided into meaningful threads. An essential interaction between hardware and software is required before applying multithreading.

A thread has its own register, counter, and stack and is a flow to execution through the process code. Thread is a type of lightweight process. Threads provide a way to improve application performance through the parallel operation. Each thread performs exactly one routine, and no thread exists out of the routine. Every thread follows a distinct flow of control. Figure 44 shows the basic concept of multithreading:

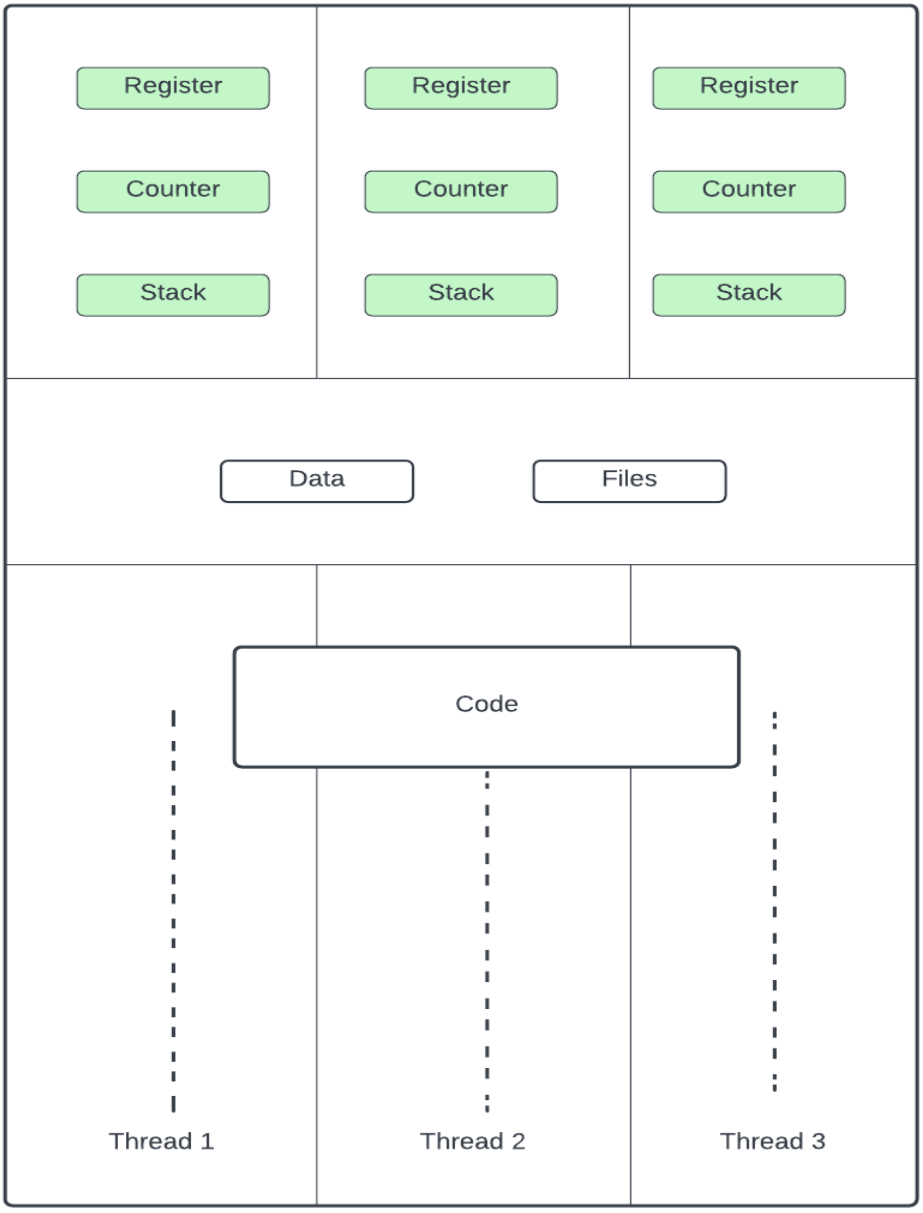



*Figure 44: Multithreading concept [25]*

To use the multithreading in the coding "pthread.h" library is required. This library includes all the commands used for the multithreaded processes. The commands used in this project related to multithreading are explained below:

*pthread_create( ): creates new thread*

```
int pthread_create(pthread_t *thread_ID, const pthread_attr *attr, void
*(*start_raoutine)(void*), void *arg);
```

To start any thread this function is called. It involves four parameters. *thread_ID* is the pointer to store the ID of thread. The thread is created executing *start_routine* with *arg* as its sole argument. *attr* is the specified attribute within the process. If *attr* is NULL, the default attributes are used. If the attributes specified by *attr* are modified later, the thread's attributes are not affected. If *pthread_create( )* returns zero means thread has been successfully created otherwise there is an error in creation.

*pthread_join( ) : join with a terminated thread*

```
int pthread_join(pthread_t thread, void **retval);
```

The *pthread_join( )* function waits for the running thread specified by the *thread* to terminate successfully. If that thread has already finished, this function returns immediately. If *retval* is other than NULL, the *ptherad_join( )* function copies the exit status of the target thread into the location pointed to by *retval.* If multiple threads simultaneously try to join with the same thread, the results are undefined. On success, it returns zero otherwise error number.

*pthread_exit( ): terminate calling thread*

```
noreturn void pthread_exit(void *retval);
```

This function terminates the calling thread and returns a value via *retval* that is available to another thread in the same process that calls *pthread_join( ).*

*pthread_cond_wait( ) : blocks on conditional variable*

```
int pthread_cond_wait(pthread_cond_t *cond,
pthread_mutex_t *mutex);
```

Whenever a conditional variable is involved in the coding this function is required. There is always a Boolean logic involving shared variables to each condition wait. The thread should proceed when this predicate become true.

*pthread_cond_signal( ) : signal a condition*

```
int pthread_cond_signal(pthread_cond_t *cond);
```

This function shall unblock at least one of the threads that are blocked on the specific conational variable "*cond"*.

A mutex (mutual exclusion object) is a program object that is created so that multiple program thread can take turns sharing the same resource. It is used to avoid errors that can cause by parallel

processing of various threads. A mutex is needed to be created at the beginning of the program by using a unique ID for it. Any thread working on a resource must use the mutex to lock it first, so that other threads can't access the resource at the same time and must unlock it after completing the task. If a resource is locked, the other threads trying to access the same sources have to wait until the mutex gets unlocked.

*pthread_mutex_init( ) : initialize the mutex*

**pthread_mutex_init(pthread_mutex_t *retval, const pthread_mutexattr_t *mutexattr);**

This function initializes the mutex referenced by *mutex* with attributes specified by *attr*. If the attribute is NULL the default mutex attributes are used. After successful initialization, mutex is available for all threads in the unlocked condition.

*pthread_mutex_lock ( ) : lock the mutex*

**int pthread_mutex_lock(pthread_mutex_t *mutex);**

This function locks the mutex object referenced by a *mutex*. In case the mutex is already locked, the thread calling this function must block until mutex becomes available.

*pthread_mutex_unlock ( ) : unlock the mutex*

**int pthread_mutex_unlock(pthread_mutex_t *mutex);**

Mutex unlocking function is used to release the mutex object referenced by *mutex*. The sequence in which mutex is released is dependent on the type of attribute of mutex. If there are threads blocked on the mutex object referenced by mutex when function is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

*pthread_mutex_destroy ( ) : kills the mutex*

**int pthread_mutex_destroy(pthread_mutex_t *mutex);**

Every mutex which is created must be destroyed. This function is used to destroy the mutexes created in the start of the program. A destroyed mutex object can be reinitialized using *pthread_mutex_init( )*, the results of otherwise referencing the object after it has been destroyed are undefined. For safe programming, mutex must be unlocked before destroying. Otherwise destroying locked mutex results in undefined behavior.

# 6. Sensor Development

For measurement of liquid nitrogen level, a finite circular cylindrical cryogenic tank is used. These shapes of tanks are used commonly as cryogenic fuel tanks for liquid-propellant launch vehicle. And these lightweight tanks to store cryogenic hydrogen are important components of spacecraft upper-stages, single stage to orbit vehicles, and solar thermal propulsion systems. Following are sample photos of cryogenic tank that is available at DLR.

*Figure 45: Cryogenic tanks at DLR*

For experimentation purposes two sensors were built. One was a basic sensor with singular winding, other one was with bifilar winding. Both sensors are built with single layer of winding i.e. there is no overlapping.

## 6.1. Explanation

To measure level of liquid nitrogen in cryogenic tank a resistive wound sensor will be used. When a sensor is placed in cryogenic tank the change in level of liquid nitrogen will affect the resistance of sensor. Hence this change in resistance of sensor will indirectly shows the level of liquid nitrogen. This change in resistance can be obtained from following formula.

$$R(T) = R(T_a)\,[1 + \alpha(T - T_a) + \beta(T - T_a)^2] \qquad (2)$$

Here

$R(T)$ = Resistance of liquid nitrogen at given temperature T

$T_a$ = Temperature at point a

$R(T_a)$ = Resistance at temperature point a

$\alpha$ and $\beta$ are constants

## 6.2. Flow rate of Liquid Nitrogen

The change in resistance values of Sensor will indirectly tell the height of liquid nitrogen level in tank. Hence Flow rate of liquid nitrogen can be calculated easily by measuring the change in height of liquid nitrogen in cryogenic with respect to time.

Following is a diagram showing experimental setup of resistive sensor.

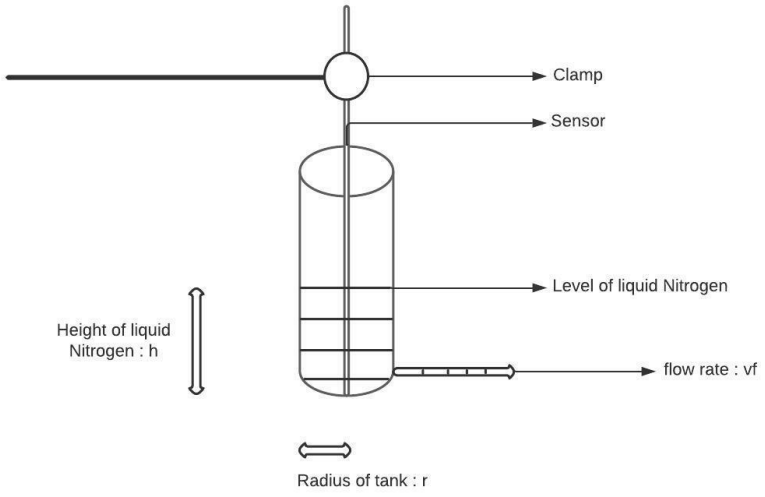



*Figure 46: Experimental setup*

Flow rate of liquid nitrogen can be calculated easily if we know the volume liquid nitrogen in tank. Following is a mathematical description of how flow rate of liquid nitrogen will change with respect to change in its level.

Volume of liquid Nitrogen: $V$

Radius of tank: $r$

The flow rate out of liquid nitrogen can also be found out by following equation

$$Flow\ rate : v_f = \frac{dV}{dt} \qquad (3)$$

Here volume is

$$Volume: V = \pi r^2 h \qquad (4)$$

Substituting values from eq *(4)* into eq *(3)*

$$Flow\ rate:\ v_f = \pi r^2 \frac{dh}{dt} \qquad (5)$$

Here $\frac{dh}{dt}$ is the change of liquid nitrogen level in cryogenic tank. This change can be detected by sensor, hence flow rate of liquid nitrogen can be easily measured from this which is essential in aerospace applications.

## 6.3. Sensor development details

Following are the technical details of sensor that were used to develop it.

## 6.4. Sensor 1

Sensor 1 is a single wire wound sensor. It is wounded on plastic pipe which is hollow from centre. This sensor is only built to get the know how of dynamic of resistance change with respect to level of water.

*Figure 47: Sensor 1*

### 6.4.1. Mechanical drawing

Figure 48 shows the physical dimensions of Sensor 1.

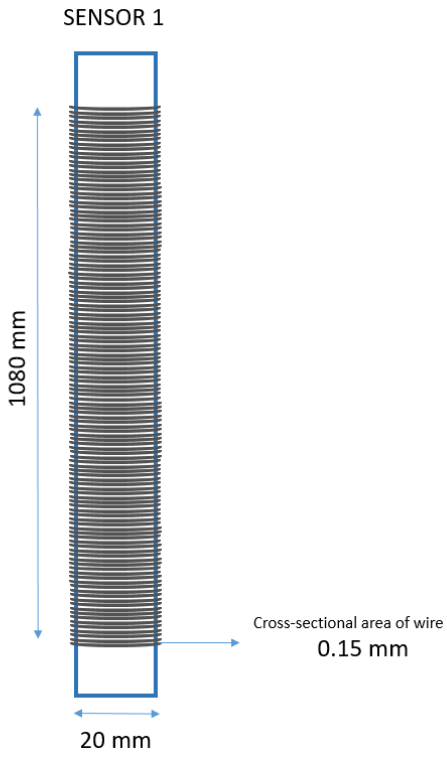



*Figure 48: Sensor 1 dimensions*

Following are its dimension and specification**.**

*Table 6: Sensor 1 Parameters*

| | |
|---|---|
| Material of Pipe | Plastic |
| Copper wire thermal expansion coefficient | 16.5 $\frac{\mu m}{mK}$ |
| Height of Pipe | 1760 mm |
| Length of wired Sensor | 1080 mm |
| Pitch of wire | 0.18 mm |
| Cross-sectional diameter of wire | 0.15 mm |
| Outer diameter of Pipe | 20 mm |

With the above data number of turns, length of wire, and resistance of wire and inductance of sensor can be calculated.

Following are the calculation showing them.

### 6.4.2. Number of turns

Number of turns of wire on Sensor 1 can be found by dividing wired sensor length by the pitch of winding.

$$Number\ of\ turns: N = \frac{length\ of\ wired\ sensor}{Pitch\ of\ wire} \qquad (6)$$

$$N_1 = \frac{1080\ \text{mm}}{0.18\ \text{mm}} = 6000$$

Here $N_1$ reperesents the number of turns of Sensor 1

From number of turns and the rod radius we can calculate the length of wire.

### 6.4.3. Length of wire

Length of wire that is wounded around the sensor rod can be found out by multiplying number of turns with the circumference of rod which is writen in following formula.

$$Length\ of\ wire:\ L = N \times (2\pi r) \qquad (7)$$

Length of wire of sensor 1 is as following

$$L_1 = 6000 \times \left(2\pi(0.01\text{m})\right) = 376.8\ \text{m}$$

Here $L_1$ reperesents the number of turns of Sensor 1

Once we know the length of wire we can calculate the resistance of that wired sensor.

### 6.4.4. Resistance

Its resistance at 20 ℃ can be found out by following formula

$$R = \rho\frac{L}{A} \qquad (8)$$

Resitivity constant ($\rho$) at 20 °C for copper is 1.68 $\times 10^{-8}$ Ωm . L is the length of wire and A is the cross sectional area of wire

Calculating resistance of sensor at 20 °C by putting values in eq *(8)*

$$R_1 = 1.68 \times 10^{-8}\Omega\text{m} \quad \frac{376.8\,m}{\pi\,(7.5 \times 10^{-5}\ \text{m}^2)^2} = 359.6\ \Omega$$

Here $R_1$ reperesents the number of turns of Sensor 1.

### 6.4.5. Inductance

To calculate inductance of sensor following formula will be used.

$$L = \frac{\mu N^2 A}{d} \qquad (9)$$

Here *N* is the number of turns *A* is the cross sectional area of rod, *d* is length of wired sensor and $\mu$ is the permittivity of free space.putting these values in eq *(9)*

$$L_1 = \frac{\left(4\pi \times 10^{-7}\,\frac{\mathrm{H}}{\mathrm{m}}\right) \times (6000)^2 \times (\pi \times (0.01\ \mathrm{m})^2)}{1.080\ \mathrm{m}} = 13.14\ \mathrm{mH}$$

### 6.4.6. Comparison between Theoretical and Actual value

Following is the comparison between theoretical and actual value of resistance and inductance of sensor 1.

*Table 7: Comparison between actual and theoretical value of Sensor 1*

| Theoretical Value | Actual Value |
|---|---|
| Resistance($R_1$)= 359.6 Ω at (20 °C) | Resistance($R_1$)= 382.3 Ω (at 22.4 °C ) |
| Inductance($L_1$)=13.14 mH | Inductance($L_1$)=13.01 mH |

## 6.5. Sensor 2

Sensor 2 is a more refined sensor to measure cryogenic temperature. Its rod is made up of PE 1000 UHMWPE (polyethylene), natural white. The purpose of choosing this material was because it can withstand temperature upto -200 °C without any physical deformation. It has a good resistivity of $>10^{14}$ Ω, essentially making it an excellent insulator. This rod is not hollow from centre.
Apart from material copper wire was wounded in bifilar fashion across the rod. Purpose of this bifilar wounding is to reduce its induction which can hamper with measuring instrument.
Figure 49 is the photo of sensor 2.

*Figure 49: Sensor 2*

### 6.5.1. Mechanical drawing

Figure 50 shows the physical dimensions of sensor 2.

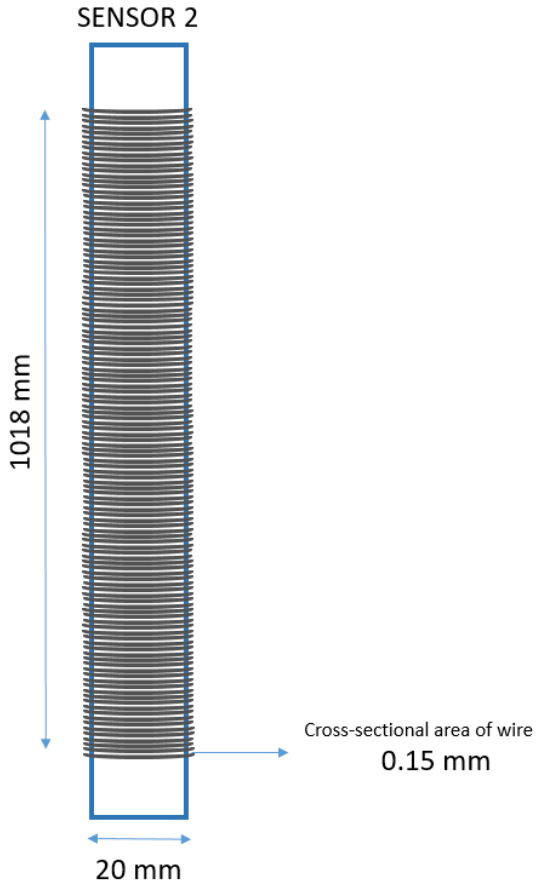



*Figure 50: Sensor 2 dimensions*

Table 8 shows its dimension and specification of sensor 2**.**

*Table 8: Sensor 2 Parameters*

| | |
|---|---|
| Material of Pipe: | PE 1000 UHMWPE (polyethylene), natural white |
| Copper wire thermal expansion coefficient | 16.5 $\frac{\mu m}{mK}$ |
| Height of Pipe | 1760 mm |
| Height of wired Sensor | 1018 mm |
| Pitch of wire | 0.175mm |
| Cross-sectional diameter of wire | 0.15 mm |
| Outer diameter of Pipe | 20 mm |

With the above data we can calculate the number of turns, length of wire, and resistance of wire and inductance of sensor.

Following are the calculation showing them.

### 6.5.2. Number of turns

Number of turns of wire on Sensor 2 rod can be found by by putting values in eq *(6).*

$$N_2 = \frac{1018\ \text{mm}}{0.175\ \text{mm}} = 5817$$

Here $N_2$ reperesents the number of turns of Sensor 2

### 6.5.3. Length of wire

Length of wire of sensor 2 can be found out by putting values in eq *(7*).

$$L_2 = 5817 \times \left(2\pi(0.01\text{m})\right) = 365.3\ \text{m}$$

Here $L_2$ reperesents the number of turns of Sensor 2

Once we know the length of wire we can calculate the resistance of that wired sensor.

### 6.5.4. Resistance of sensor

Resistivity constant (ρ) at 20 °C for copper is $1.68 \times 10^{-8}\ \Omega\text{m}$. L is the length of wire and A is the cross-sectional area of wire.

Calculating resistance of sensor 2 at 20 °C by putting values in *eq (8).*

$$R_2 = 1.68 \times 10^{-8}\ \Omega\text{m} \quad \frac{365.3\text{m}}{\pi\ (7.5 \times 10^{-5}\ \text{m}^2)^2} = 347.4\ \Omega$$

Here $R_2$ reperesents the number of turns of Sensor 2

### 6.5.5. Inductance

Theoretically Inductance of a bifilar coil is zero because of the opposite current flow with respect to each other in parallel winding.

### 6.5.6. Comparison between Theoretical and Actual value

Following is the comparison between theoretical and actual value of resistance and inductance of sensor 2.

*Table 9: Comparison between actual and theoretical value of Sensor 2*

| Theoretical Value | Actual Value |
|---|---|
| Resistance($R_2$)= 347.4 Ω at (20 °C) | Resistance($R_2$)= 365.3 Ω (at 22.4 °C ) |
| Inductance($L_2$)=0 mH | Inductance($L_2$)=1.301 mH |

It can be seen inductance is very low as compare to the sensor 1. In numbers its 10 times low as compare to sensor one with singular winding.Although its inductance is reduced but its capacitance is increased because of the parallel winding.

## 6.6. Test Specification

As the sensor is designed and now it needs to be tested. Following are some of its test specification that needs to address.

- In this experiment the dynamics of sensor and its resistance will be checked as the liquid Nitrogen level in cryogenic tank will change.
- These change in resistance will help to identify the change in level of liquid nitrogen hence also indicating any inwards or outwards flow rate.
- Any mechanical stress or deformation on Sensor would also be checked as the operating range of sensor would be 25 °C to -200 °C. (Ideally the thermal constants of copper wire and PE 1000 UHMWPE (polyethylene), natural white should be same to avoid any stress on sensor).
- Any distortion due to external electromagnetic interferences will also be checked.
- Sensor inductance effect on measuring device needs to be checked.
- Thermal time constant of sensor needs to be determined as well. i.e how quickly it will respond to change in resistance and whether that will be in acceptable range as keeping in view of the maximum flow rate.

## 6.7. Test pass /fail criteria

Below are some of its pass fail criteria that needs to be predefined.

- Is there any physical deformation.
- Is there any distortion in values due to electromagnetic interferences?
- Does induction of coil hampering with the measuring device.
- Are the resistive values regeneratable i.e what is the standard deviation after multiple tests.
- Is the error less then acceptable range?
- Is thermal time constant of sensor is in acceptable range.

## 6.8 Test Plan at DLR

- The test was conducted for DLR on 29th of April at 09:30 A.M on sensor 2. Room temperature was 20.95 °C
- The sensor was hooked with the clamp and was inserted into the cryogenic tank.
- Figure 51 shows the experimental setup of test setup at DLR

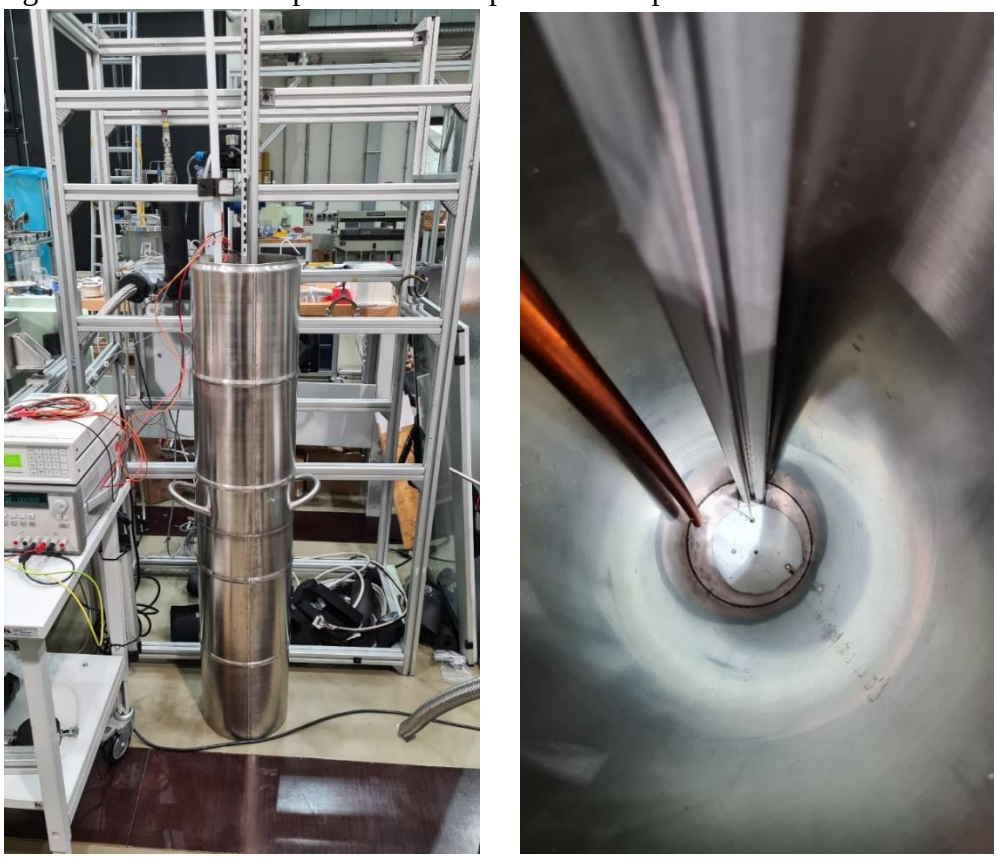


*Figure 51: Experimental setup*

Figure 52 shows the schematic setup of experiment. Sensor was clamped in such a way that its height was 40 mm above the bottom of tank

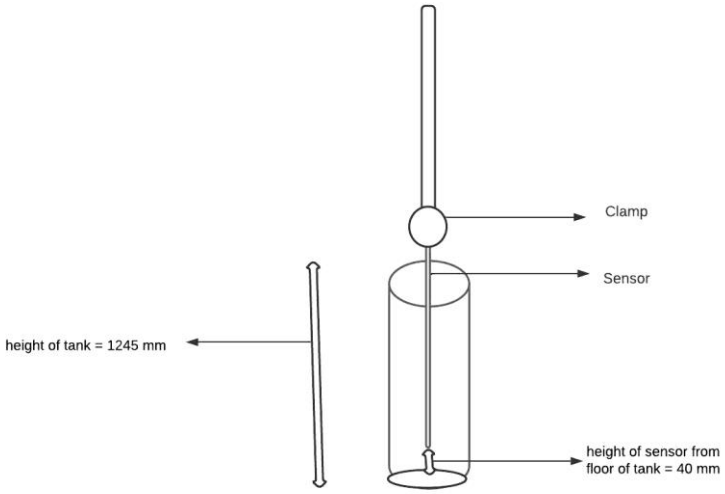



*Figure 52 : Experimental schematic*

The sensor was connected with the high precision meter (keysight 34465A) of results showing up to 6 significant digits in a 4 wire configuration. A USB stick was used to record data from the measuring device. The tank was first filled with liquid nitrogen up to a certain predetermined level, after which the sensor took some time for the resistance to reach a steady state. Figure 53 shows cryogenic tank filled with liquid nitrogen.

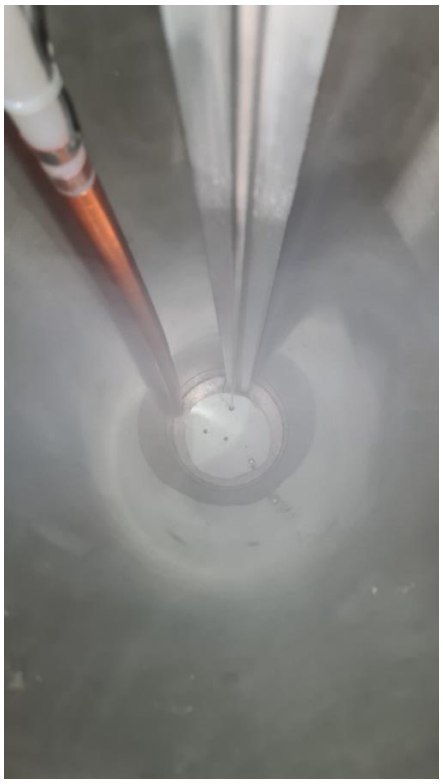


*Figure 53 : Tank with liquid nitrogen inside*

- This process of filling was done 6 times and readings of resistance of sensor were recorded with respect to the level of liquid nitrogen in tank. Following were the readings that were recorded after the sensor showed steady resistance readings.These values can be seen in Table 10.

*Table 10 : Resistance vs level graph of liquid nitrogen during filling stage*

| **Level of Liquid Nitrogen in tank *(mm)*** | **Resistance of Sensor (Ω)** |
|---|---|
| 0 mm | 362.200 Ω |
| 40 mm | 207.000 Ω |
| 143 mm | 155.700 Ω |
| 344 mm | 108.333 Ω |
| 544 mm | 77.1112 Ω |
| 655 mm | 61.1765 Ω |
| 965 mm | 44.6651 Ω |

Figure 54 shows graph that is obtained from the Table 10.

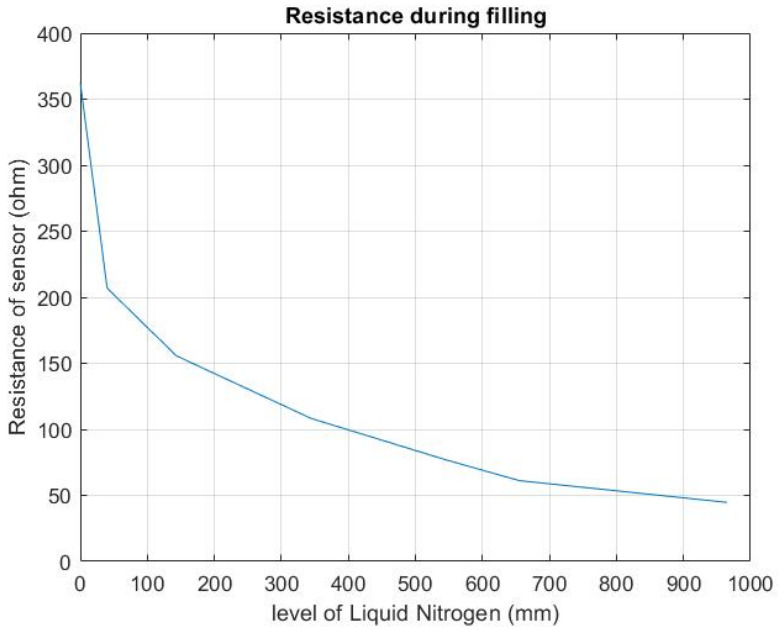



*Figure 54 : Resistance vs level graph of liquid nitrogen during filling*

### 6.8.1. During Evaporation

After the tank was filled with liquid nitrogen it was set for evaporation. Following data was recorded of resistance against level of liquid nitrogen in tank during evaporation phase.It can be seen in Table 11.

*Table 11: Resistance vs level graph of liquid nitrogen during evaporation stage*

| Level of Liquid Nitrogen in tank (mm) | Resistance of Sensor (Ω) |
|---|---|
| 965 mm | 45.1065 Ω |
| 937 mm | 45.5123 Ω |
| 913 mm | 45.6387 Ω |
| 889 mm | 46.7084 Ω |
| 869 mm | 47.6419 Ω |
| 849 mm | 48.6719 Ω |
| 585 mm | 70.4086 Ω |
| 571 mm | 72.2544 Ω |
| 555 mm | 74.1171 Ω |
| 540 mm | 76.1032 Ω |
| 511 mm | 79.7328 Ω |
| 483 mm | 83.3745 Ω |
| 251 mm | 120.295 Ω |
| 240 mm | 120.538 Ω |
| 228 mm | 121.064 Ω |
| 205 mm | 124.400 Ω |
| 194 mm | 125.831 Ω |
| 183 mm | 127.737 Ω |
| 172 mm | 129.954 Ω |
| 0 mm | 212.698 Ω |

Figure 55 shows the plot of the level of liquid nitrogen against resistance that is obtained from Table 11 during evaporation phase.

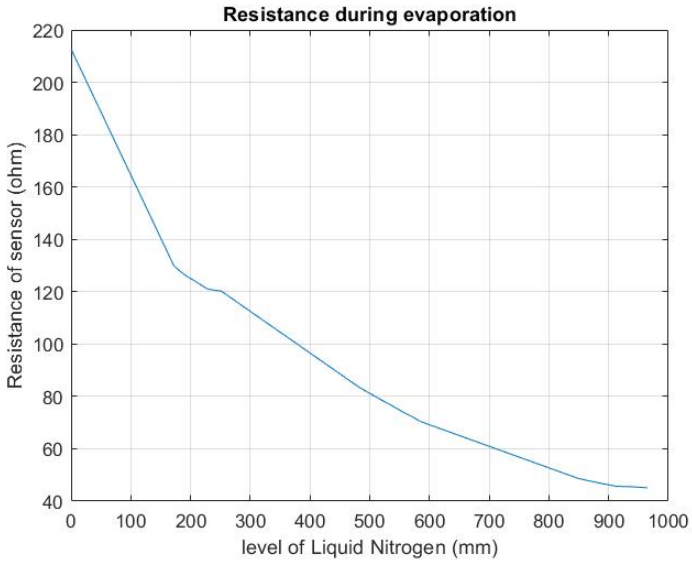



*Figure 55 : Resistance vs level graph of liquid nitrogen during evaporation*

During evaporation, data was also recorded on USB stick with a sampling period of 10 seconds. Since the evaporation took longer than expected so it was recorded in 2 files. The data from the files are plotted below in Figure 56 and 57.

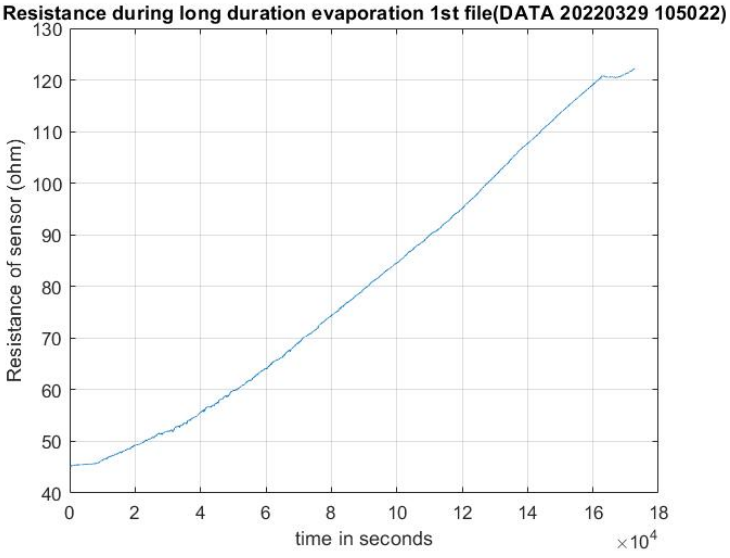



*Figure 56: Resistance vs level graph of liquid nitrogen during evaporation 1st file*

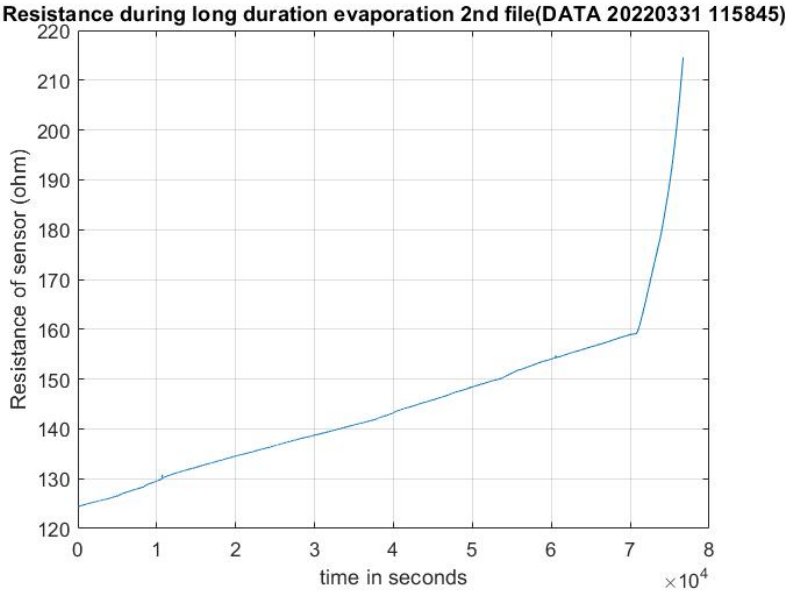



*Figure 57 : Resistance vs level graph of liquid nitrogen during evaporation 2nd file*

### 6.8.2. Thermal Time constant

To check the thermal dynamics of sensor its thermal time constant was calculated.it is the time required for a material to change 63.2% of the total difference between its initial and final body temperature. There were total 5 filling were recorded. Against each filling data was recorded in USB stick with sampling period of 1 second. After this data was analysed to find out the thermal time constant of sensor.For calculation of time constant, starting maximum value of resistance will be taken after filling was stopped and last reading of sample data will be taken as final reading

### Time constant during 1st filling:-

To calculate the thermal time constant, initial few seconds were ignored because at that time USB stick was being set to record data and it also took some time to bring Liquid nitrogen pipe into the tank. At that time irregular values can also be noticed that be seen in the Figure 58.

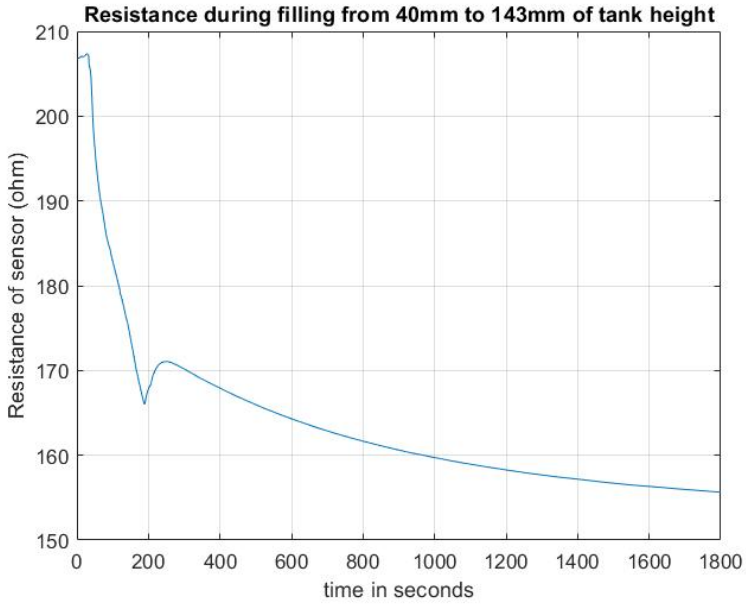



*Figure 58 : First filling graph*

Figure 59 shows filtered graph which is obtained from figure 58, from which time constant will be calculated.

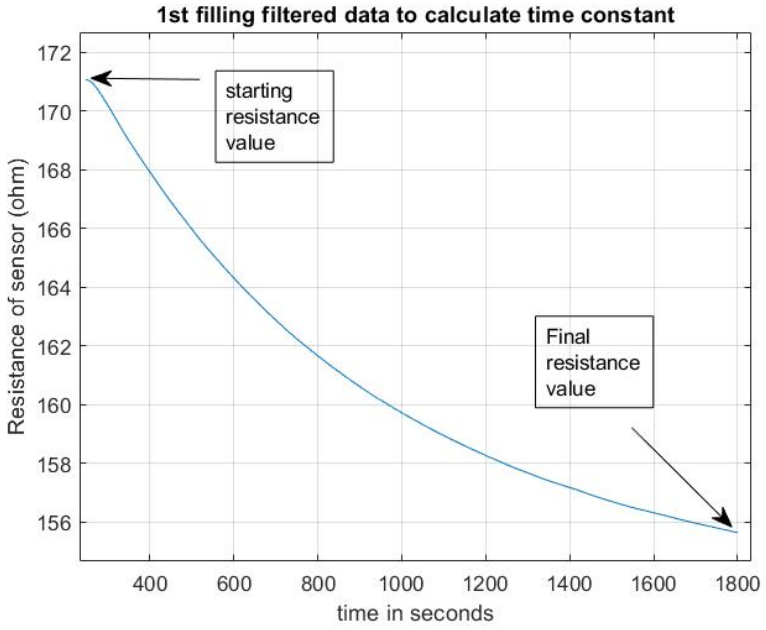



*Figure 59 : Filtered data to calculate time constant*

To calculate the thermal time constant starting maximum resistance will be subtracted from final resistance and then 63.2 % of the difference would be taken. That percentage value will be subtracted from the starting resistance value to get a resistance value of 63.2 %. The time taken to

reach that value will be thermal time constant. The mathematics for calculating the thermal time constant is stated below

- The starting resistance value is 171.0508 Ω
- The final resistance value is 155.6481 Ω
- The difference is 171.0508 Ω – 155.6481 Ω =15.4027 Ω
- The 63.2 % of the difference is 15.4027 Ω × 0.632 = 9.7345 Ω
- So the resistance at 63.2 % is 171.0508 Ω – 9.7345 Ω = 161.3163 Ω
- So time at 174.6780 Ω is calculated from the following graph is 831-248=**583 sec**

### Time constant during 2nd filling:-

Figure 60 is the graph that shows the resistance value of the sensor during 2nd stage filling.

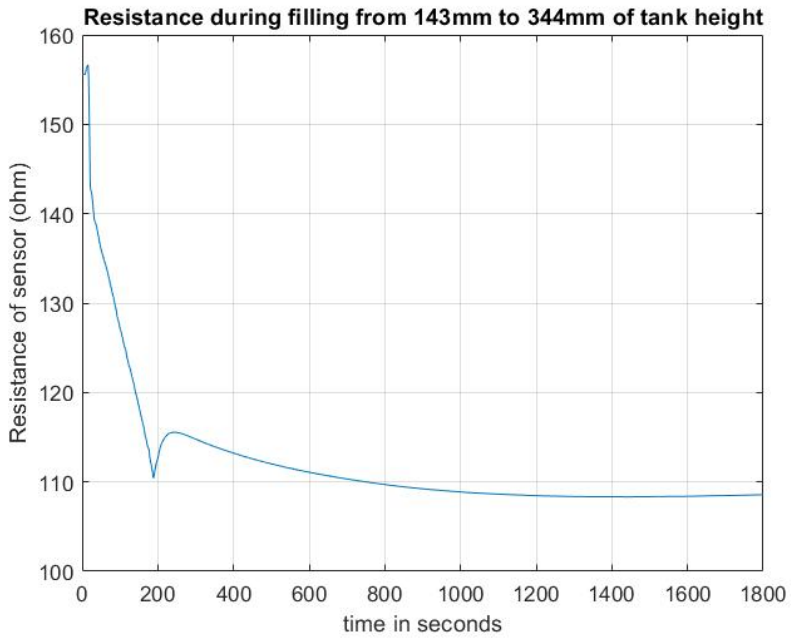



*Figure 60 : Second filling graph*

The mathematics for calculating the thermal time constant is stated below

- The starting resistance value is 115.4593 Ω
- The final resistance value is 108.5727 Ω
- The difference is 115.4593 Ω – 108.5727 Ω =6.8866 Ω
- The 63.2 % of the difference is 6.8866 Ω × 0.632 = 4.3523 Ω
- So the resistance at 63.2 % is 115.4593 Ω – 4.53523 Ω = 111.1070 Ω
- So time at 111.1070 Ω is calculated from the following graph is 597 – 232=**365 sec**

### Time constant during 3rd filling:-

Figure 61 is the graph that shows the resistance value of the sensor during 3rd stage filling.

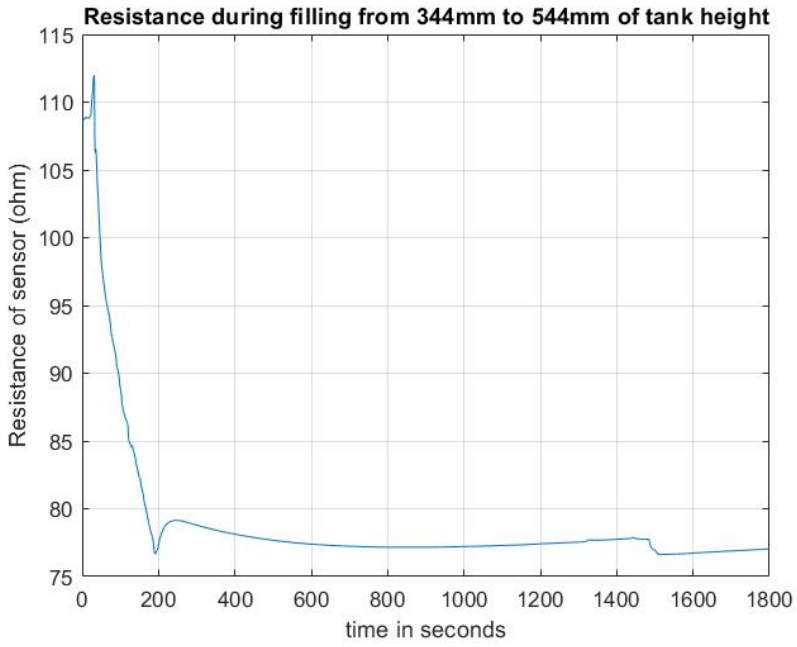



*Figure 61: Third filling graph*

The mathematics for calculating the thermal time constant is stated below

- The starting resistance value is 79.1418 Ω
- The final resistance value is 77.0423 Ω
- The difference is 79.1058 Ω – 77.0423 Ω =2.0995 Ω
- The 63.2 % of the difference is 2.0635 Ω × 0.632 = 1.3268 Ω
- So, the resistance at 63.2 % is 79.1418 Ω – 1.3268 Ω = 77.8150 Ω
- So, time at 77.8008 Ω is calculated from the following graph is 467 – 246=**221 sec**

### Time constant during 4th filling: -

Figure 62 the graph that shows the resistance value of the sensor during 4th stage filling.

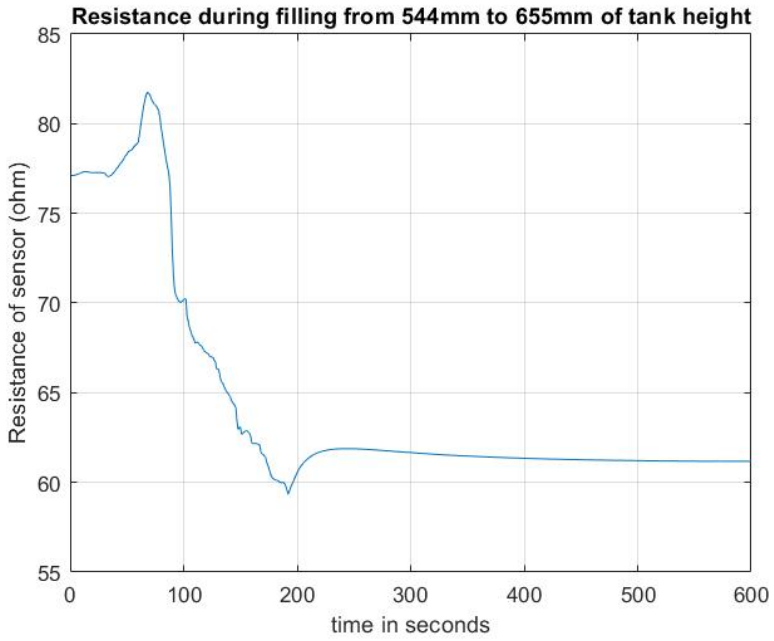



*Figure 62: Fourth filling graph*

The mathematics for calculating the thermal time constant is stated below

- The starting resistance value is 61.8817 **Ω**
- The final resistance value is 61.1765 **Ω**
- The difference is 61.8817 **Ω** – 61.1765 **Ω** =0.7052 **Ω**
- The 63.2 % of the difference is 0.7052 **Ω** × 0.632 = 0.4456 **Ω**
- So, the resistance at 63.2 % is 61.8817 **Ω** – 0.4456 **Ω** = 61.4361 **Ω**
- So, time at 61.4361 **Ω** is calculated from the following graph is 362 – 244=**118 sec**

### The time constant during 5th filling:-

Figure 63 is the graph that shows the resistance value of the sensor during 5$^{th}$ stage filling.

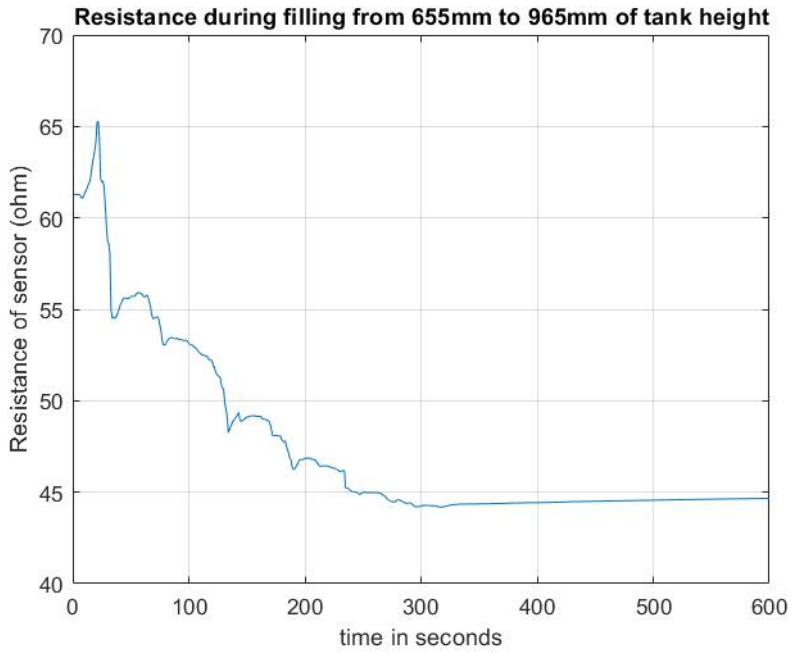



*Figure 63: Fifth filling graph*

The mathematics for calculating the thermal time constant is stated below

- The starting resistance value is 46.7669 Ω
- The final resistance value is 44.6651 Ω
- The difference is 46.7669 Ω – 44.6651 Ω =2.1018 Ω
- The 63.2 % of the difference is 2.1018 Ω × 0.632 = 1.3283 Ω
- So, the resistance at 63.2 % is 46.7669 Ω – 1.3283 Ω = 45.4386 Ω
- So, time at 45.4386 Ω is calculated from the following graph is 236 – 198=**38 sec**

## Summary

Table 12 shows the summary of time constant of all fillings along with mean and standard deviation.

*Table 12 : Time constants summary, mean and standard deviation*

| Filling stage | Change in level (mm) | Time constant(sec) |
|---|---|---|
| 1 | 103 mm | 583 sec |
| 2 | 201 mm | 365 sec |
| 3 | 200 mm | 221 sec |
| 4 | 111 mm | 118 sec |
| 5 | 310 mm | 38 sec |
| Mean | | **265 sec** |
| Standard deviation | | **192.9756 sec** |

### 6.8.3. Test Observations and Recommendations

- The test showed quite a smooth change in resistance with respect to the level of liquid nitrogen.it was observed there were some sharp changes in resistance during the initial phase of filling for the first couple of fillings. But after that sensor response was very smooth and it reached in its steady state in less time. With a precooled cryogenic tank, it is expected that the first couple of filling will also generate a smoother response.
- Another observation was that winding got slipped because sensor got shrunk a bit which can be seen in Figure 64. This problem can be solved by using a glue to wound wire with sensor or to make threads in sensor and wound wire on those threads. But problem in threading is that minimal thread pitch would have to be made. If every turn of wire needs to be in different threads, then lot of threading needs to be done knowing the fact that with threading height and weight of sensor will increase. Like for example separation between two turns (crest) is 0.10 mm, then number of turns multiply with 0.10 mm will result in its increased length. Another aspect to threading is that how thick crest can be made keeping the mechanical and material restrictions in check. The possible solution to this can be increasing pitch of thread such that around 100 turns of wire can be wound in it. But still the resistance behaviour of sensor because of crest of thread must be analysed.


*Figure 64 : Slipage of winding*

- The temperature differential of this cryogenic tank was high as the lid was not closed from top. But if the lid was closed then differential temperature would be less and it can be expected that resistance response would be much flatter then as it can be seen in Figure 55.
- Overall the test was of success as it passed the test specifications, as there was no deformation of sensor and neither any electromagnetic distortion hampered with values. Some more tests are certainly required under different condition such as an insulated cryogenic tank to understand the dynamics of sensor more clearly. Figure 65 shows the sensor after experimentation.

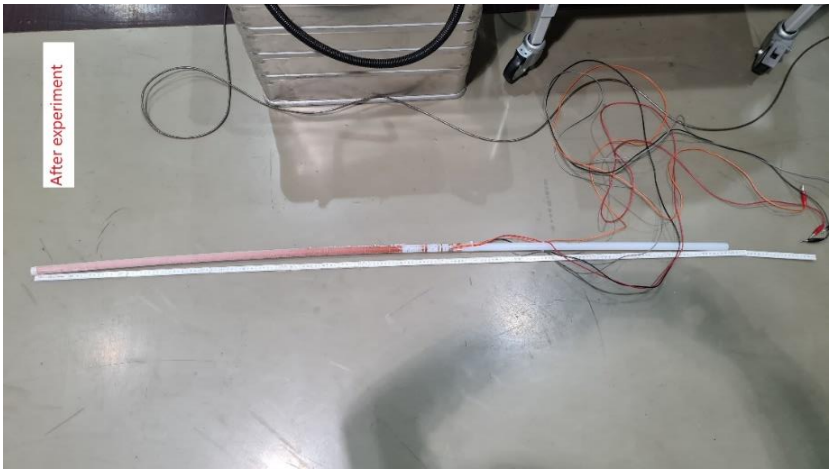



*Figure 65 : Sensor after experiment*

- From Table 12 it can be seen that with each successive fillings it is taking less time for sensor to reach steady state.
- As the cryogenic tank had no lid, and tank was filled by inserting pipe into the tank. For filling the tank pipe was inserted from the top because of which cooled air inside the tank came out causing inflow of environmental air into the tank hence increasing the resistance a bit. Similarly, when the filling was stopped, pipe was removed from tank, which again caused inflow of air hence again causing increase in resistance before reaching to steady state. These behaviours can be clearly seen in Figure 66.

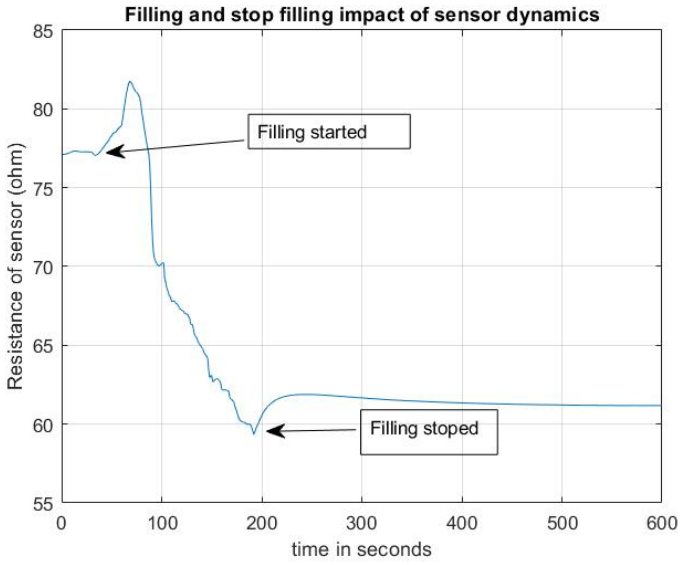



*Figure 66: Sensor dynamics during filling and filling stop stage*

- Values obtained from table 10 and 11 of resistance of sensor during filling and evaporation of liquid nitrogen is plotted in Figure 67. It can be seen that the behavior of sensor is very much similar during both phases.Since table 10 and 11 are far discrete points, and also the fact resitance of sensor during same level in two phases was not recorded so its difficult to compare point by point and do error analysis.The small difference in resistance that can be observed in both stages(filling and evaporation) can also be because of slippage of winging that happened in filling stage.In future resitance of sensor should be recorded in at same level for both evaporation and filling phase. Apart from this the sharp dip of resistance during initial phase of filling can be avoided if precooled cryogenic tank is used for experimentation.

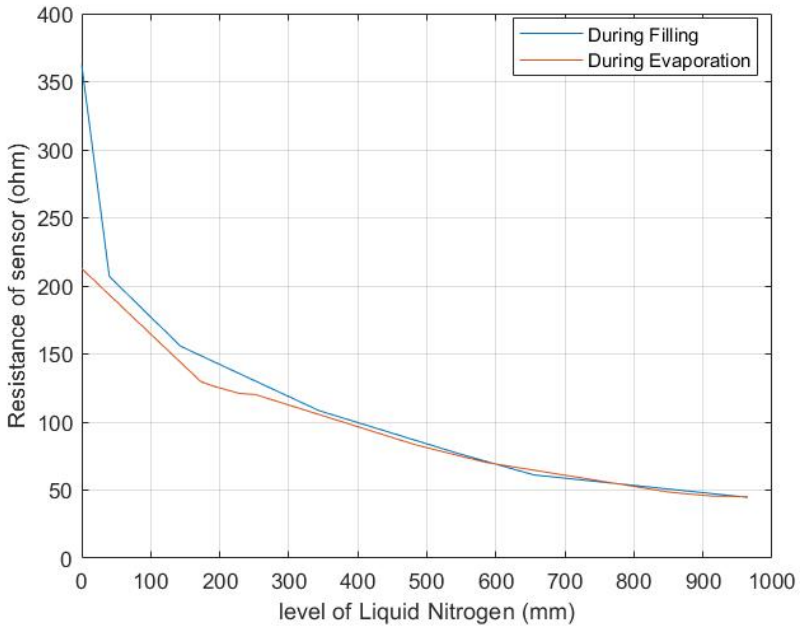



*Figure 67 : Resistance of sensor during filling and evaporation of liquid nitrogen*

## 6.9 Test Plan at Hochschule Bremerhaven

- Sensor 1 is used for experimental purposes in Hochschule Bremerhaven.
- The sensor is connected with MAX31865 in a four-wire configuration. For four wire configuration jumper J2 is taken off such that point 1 2 3 and 4 are not connected with each other.
- Water tank needs to be grounded to avoid any electromagnetic interference.
- Note the water temperature and room temperature before starting temperature.
- Sensor is then put in water tank as shown in Figure 68.

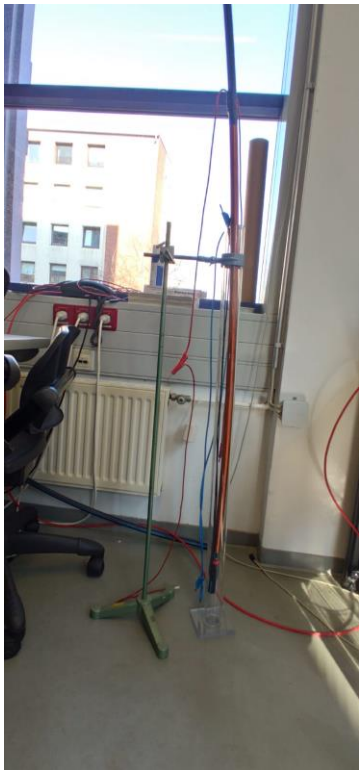


*Figure 68: Experimental setup at Hochschule Bremerhaven*

- After that Sensor is clamped with some assembly. Special attention needs to be paid that sensor is perpendicular to the base of tank i.e it's not slanted.
- After that ZedBoard is turned on.
- Then the program is run at Vitis after which system is ready to record resistance values at different water levels.
- Water is filled in the tank gradually.
- As the water is getting filled, its level and corresponding resistance from MAX31865 or some other measuring device is noted.

### 6.9.1. Test Case 1

Before the start of experiment water temperature and room temperature were noted as below. This experiment was conducted on 03rd of March 2022 in Hochschule Bremerhaven.

Water temperature: 25.79 °C

Room Temperature: 24.2 °C

After that water inflow was started into the water tank. The resistive sensor measurement were taken from MAX31865.After recording the values of water level and its corresponding resistance following graph was obtained which is shown in Figure 69.

.

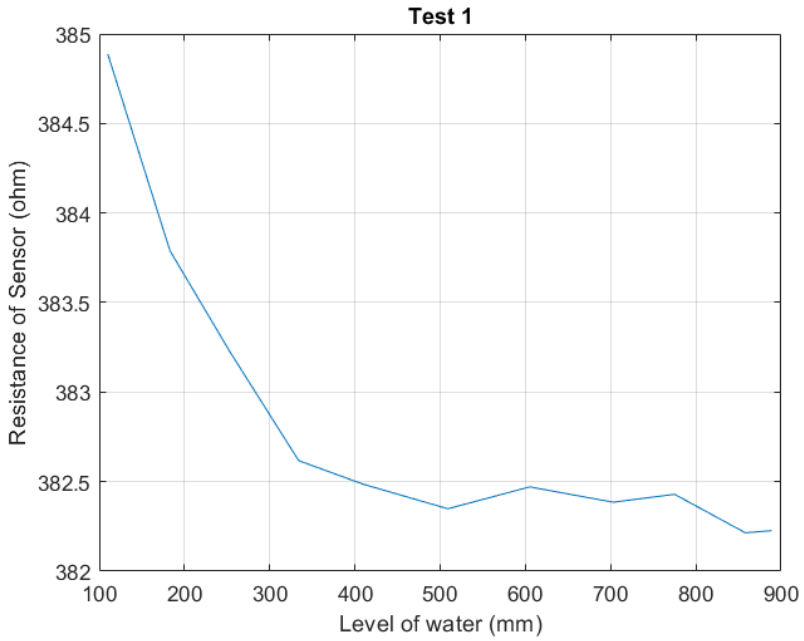



*Figure 69: Test 1 result graph*

It can be seen that with rising water level the resistance of sensor is dropping.

### 6.9.2. Test Case 2

Before the start of experiment water temperature and room temperature were noted as below. This experiment was conducted on 03rd of March, 2022 in Hochschule Bremerhaven.

Water temperature: 17.2 °C

Room Temperature: 22.5 °C

After that water inflow was started into the water tank. The resistive sensor measurement was taken from MAX31865. After recording the values of water level and its corresponding resistance following graph was obtained.

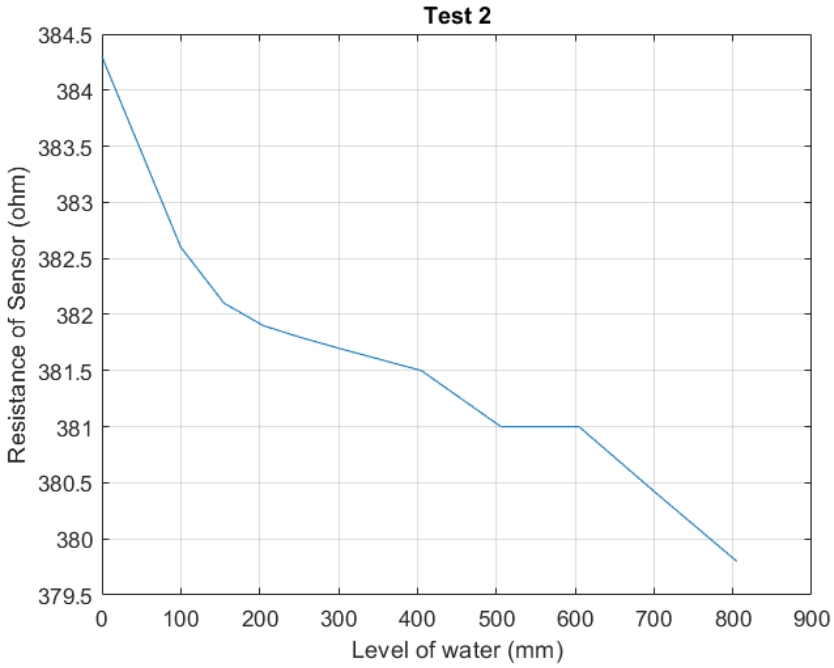



*Figure 70: Test 2 result graph*

It can be seen that with rising water level the resistance of sensor is dropping.

### 6.9.3. Hochschule Bremerhaven Test Remarks

- Tests showed decrease in resistance as the water level was raised.
- Tests were not very precise due to limited availability of proper resources. The water tank was very narrow i.e. while pouring water it was getting in contact with sensor from top as well. Apart from that one fourth of sensor was not fully submerged because of less height of tank. During both experiments room and water temperature were not equal.

# 7. Development of TMNG application and process flow

## 7.1 Main TMNG module

This part of our application can be considered as top design file which handles TCP/IP connectivity and different design threads. It includes all the necessary libraries associated with Linux based applications. It starts with declaration of *pthread_mutex_t*, *mode* (char array to store auto or manual command), *data_fifo* (a 2D data FIFO of size 100x50), and *SensParam*_struct (structure of parameters used for choosing sensor and setting sample counts, sample time, and sampling unit). This part of application is also creating thread IDs.

While initialization, the clock_gettime () function gets the current time of the clock specified by clock ID, and puts it into the buffer pointed to by init_*time*. The only supported clock ID is CLOCK_REALTIME.

The *init_time* parameter points to a structure containing at least the following members:

- **time_t tv_sec:** The number of seconds since 1970.
- **time_t tv_nsec:** The number of nanoseconds expired in the current second. This value increases by some multiple of nanoseconds, based on the system clock's resolution.

Moreover, there are special set of functions which play an important role in this project:

- **sighandler:** This function is used to catch **Cltr+C** to terminate the whole application
- **pthread_cond_signal ():** This call unblocks at least one of the threads that are blocked on the specified condition variable *cond* (if any threads are blocked on *cond*)
- **sched_param:** This function is used to define the priority of the current thread.

Once declaration and definition of above variable and functions is done, the hardware device (*meascdd*) is opened. After that the socket file descriptor (*sockfd*) is created and configured mainly with socket option, the Port number and IP address. After this, socket checks for binding and starts to listen for any request from remote device or client. Upon receiving request from valid client, the server accepts the connection.

After getting connected to client or any remote device, the main program moves to launching threads. It launches following set of threads:

- Resistive sensor thread (*res_sensor*)
- User interfacing thread (*uimodule*)
- CANOpen Sensor thread (*CANOpen_thread*)
- Data Transfer thread (*data_transfer*)
- Push buttons thread (*PB_transfer*)
- Switches thread (*SW_transfer*)

Once termination of program is requested by user, the main program checks for threads to join and terminate them safely. After all is done, the *sockfd* is closed and program is exited.

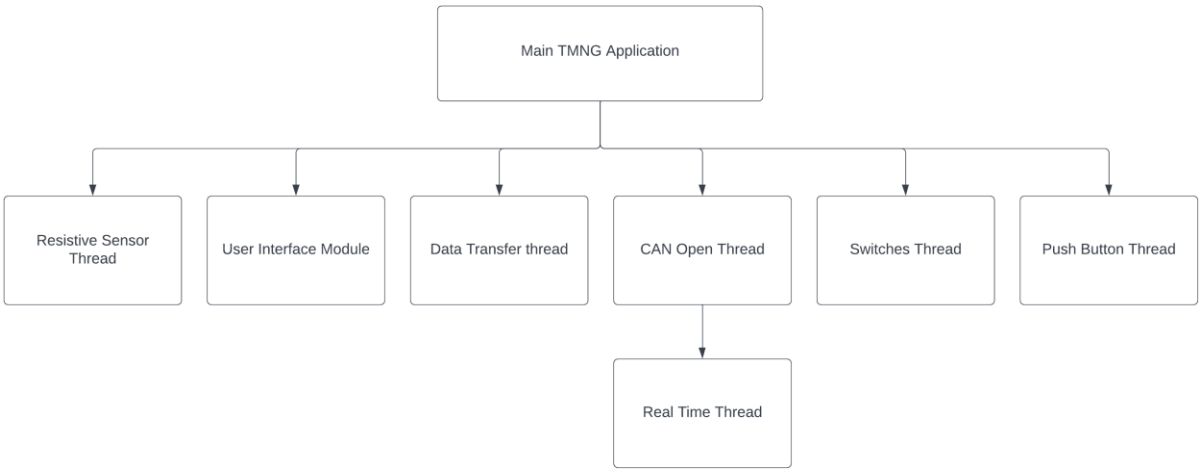



*Figure 71: TMNG main module flow chart*

## 7.2 Resistor Sensor Thread

The resistor sensor thread is responsible to measure the resistance of the temperature sensor connected. A MAX31685 device is used as ADC converter and measured ADC code is used to calculate the resistance/temperature.

Since multithreading is employed, a separate thread is launched from main for resistance measurement. The steps for measurement are already explained in the design components section. This section contains the explanation of logic that is used for implementation.

The thread powers the MAX31865 device prior to start of a measurement. It is required that the device need to be powered on for some time before the measurement, for an accurate conversion. A global flag "*terminate*" is used to check whether the thread need to be terminated or not. An infinite loop is executed based on this flag. In order to avoid CPU usage during this infinite loop, a thread conditional variable is used to wait for the resistance measurement request. Once the request for measurement is received, the "*terminate*" flag is checked again. This is to avoid the thread conditional variable missing the flag while in wait condition.

After reception of measurement request and the count of samples required, another loop is started for sample counting. The time is captured at this point for time stamping the resistance value. A one-shot conversion of 50Hz filter mode is then requested to the MAX31865 device, and thread waits for time of a 63ms. This time is defined in the data sheet of MAX31865 device, 62.5ms in 50Hz filter mode. Due to this limitation of wait time, the sampling rate supported for the resistance measurement is always above 63ms. Even if some value less than 63ms is entered by the user, the sampling period will be always set to 63ms by the thread.

Once the conversion is finished, the registers 1H and 2H are read from the MAX31865 device to obtain the LSB and MSB values of resistance data. Using these data, the resistance/temperature values can be calculated. To compute the time stamp, the difference between measurement time

and initial time (which is measured in the main program at the execution start of main thread) is found. This was done to decrease the number of bits required to store the time stamp value. Once the timestamp and sample value are ready, it is copied to the global data FIFO. The data transfer thread picks the data to be transmitted to the TCP client from this FIFO. The thread then clears the file pointer created for the 63ms wait and then loops over the sample count to obtain that much count of data. Once the sample count loop ends it waits for another measurement request.

Once the "*terminate*" flag is set, the thread comes out of the infinite loop, powers down the MAX31865 device and exits.

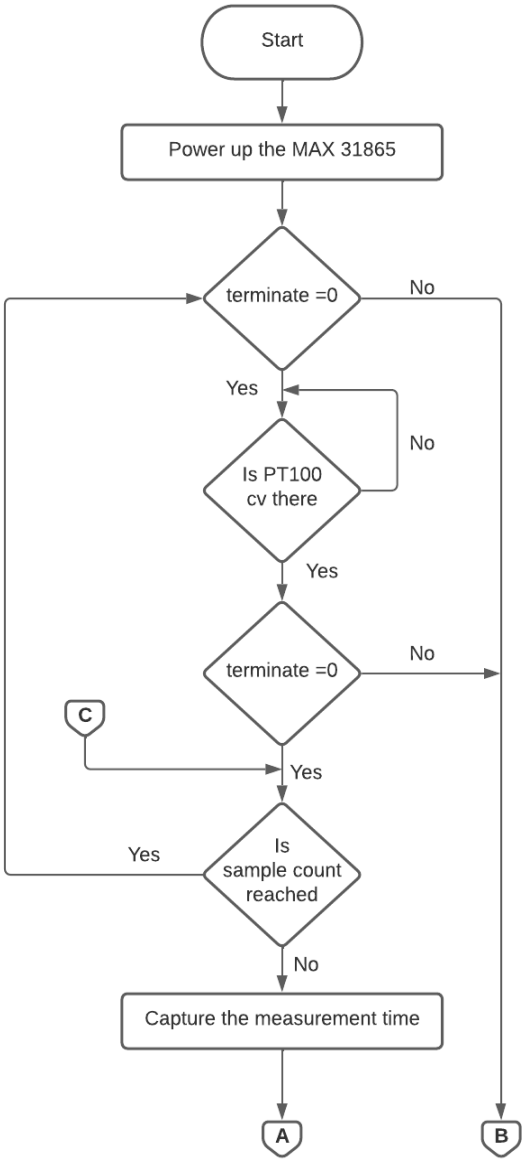



*Figure 72: Flow chart of resistor sensor thread part 1*

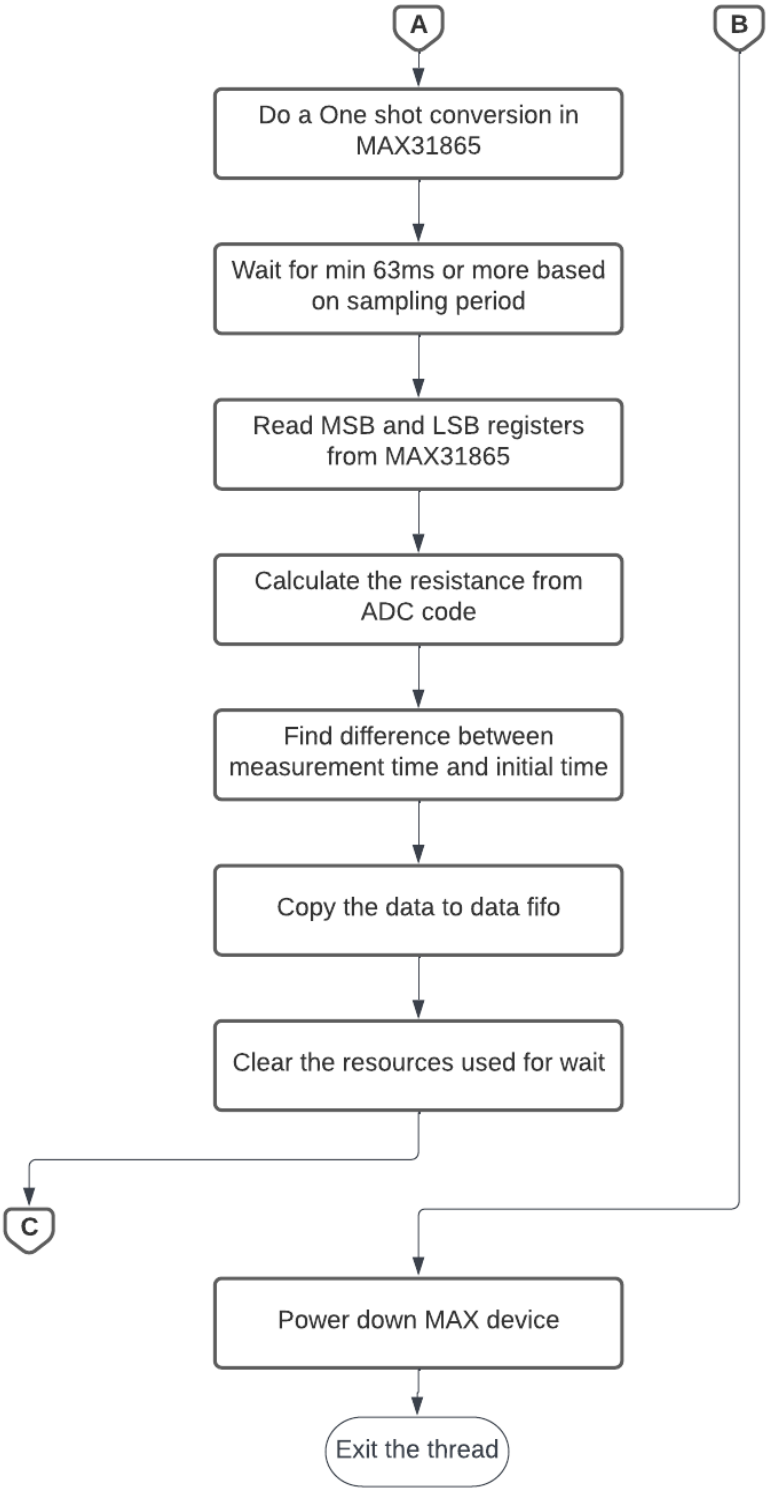



*Figure 73: Flow chart of resistor sensor thread part 2*

## 7.3 CAN sensor thread

The CAN sensor thread deals with initialization of CAN network and CANopen device. In this part, the open source CANopenNode Stack libraries are used to build CANopen objects and launching of different CANopen protocols as described in Literature part of CAN. As the CAN is implemented over a Linux so its driver files and gateway libraries are taken from CAN-Linux which is also an open-source platform about CAN implementation over the Linux OS [31] [32]. The libraries are added in following order:

- **301/** - CANopen application layer and communication profile.
  - **CO_config.h** - Configuration macros for CANopenNode.
  - **CO_driver.h** - Interface between CAN hardware and CANopenNode.
  - **CO_ODinterface.h/.c** - CANopen Object Dictionary interface.
  - **CO_Emergency.h/.c** - CANopen Emergency protocol.
  - **CO_HBconsumer.h/.c** - CANopen Heartbeat consumer protocol.
  - **CO_NMT_Heartbeat.h/.c** - CANopen Network management and Heartbeat producer protocol.
  - **CO_PDO.h/.c** - CANopen Process Data Object protocol.
  - **CO_SDOclient.h/.c** - CANopen Service Data Object - client protocol (master functionality).
  - **CO_SDOserver.h/.c** - CANopen Service Data Object - server protocol.
  - **CO_SYNC.h/.c** - CANopen Synchronisation protocol (producer and consumer).
  - **CO_TIME.h/.c** - CANopen Time-stamp protocol.
  - **CO_fifo.h/.c** - Fifo buffer for SDO and gateway data transfer.
  - **crc16-ccitt.h/.c** - Calculation of CRC 16 CCITT polynomial.
- **303/** - CANopen Recommendation
  - **CO_LEDs.h/.c** - CANopen LED Indicators
- **304/** - CANopen Safety.
  - **CO_SRDO.h/.c** - CANopen Safety-relevant Data Object protocol.
  - **CO_GFC.h/.c** - CANopen Global Failsafe Command (producer and consumer).
- **305/** - CANopen layer setting services (LSS) and protocols.
  - **CO_LSS.h** - CANopen Layer Setting Services protocol (common).

    - **CO_LSSmaster.h/.c** - CANopen Layer Setting Service - master protocol.
    - **CO_LSSslave.h/.c** - CANopen Layer Setting Service - slave protocol.
- **309/** - CANopen access from other networks.
    - **CO_gateway_ascii.h/.c** - Ascii mapping: NMT master, LSS master, SDO client.
- **storage/**
    - **CO_storage.h/.c** - CANopen data storage base object.
- **extra/**
    - **CO_trace.h/.c** - CANopen trace object for recording variables over time.
- **CANopen.h/.c** - Initialization and processing of CANopen objects, suitable for common configurations.
- **CO_driver_target.h** - Example hardware definitions for CANopenNode.
- **CO_driver.c** – driver files for interface for CANopenNode linux
- **CO_storageLinux.h/.c** - for data storage to non-volatile memory.
- **OD.h/.c** - CANopen Object dictionary source files, automatically generated from DS301_profile.xpd.
- **CO_application.h** - Application interface for CANopenNode
- **CO_epoll_interface.c/.h** - Helper functions for Linux epoll interface to CANopenNode.
- **CO_error.c/.h** - CAN module object for Linux socketCAN Error handling
- **CO_error_msgs.h** -Definitions for CANopenNode Linux socketCAN Error handling
- **cansensor.c/.h** – Main thread/ CANopen Sensor thread file

Using above CANopenNode Linux and CANopenNode Stack files, CANopen device is initialized and set to its operational mode after a set of initial setups. Firstly, the CANopen object and gateway object are created, and some important functions, variable, and callback functions are declared and defined. After that, the initialization process of CAN protocol is started involves extraction of CAN interface ID, Epoll memory allocation, and creating of main and real time Epoll functions, assignment of real time file descriptor ID to CAN pointer file descriptor ID, and creation of main gateway function. Following this, the initialization loop for CANopen device starts which involves invoking of following sub routines:

- CAN Configuration
- CAN module initialization
- LSS initialization
- Assignment of pending node ID to an active node ID

- CANopen device initialization
- CANopen PDO initialization

After this, the Realtime thread is created for the first the run of the application for PDO and SYNC communication. It uses the epoll functions which execute in parallel with RealTime thread for handling the PDO and SYNC messages.

At last, when a *LastRun* flag is set, RealTime thread in terminated, epoll objects are deleted, CAN object is deleted, memory is released, and is safely exited from the CANopen main/CAN sensor thread. The process is can easily understood from following flow chart.

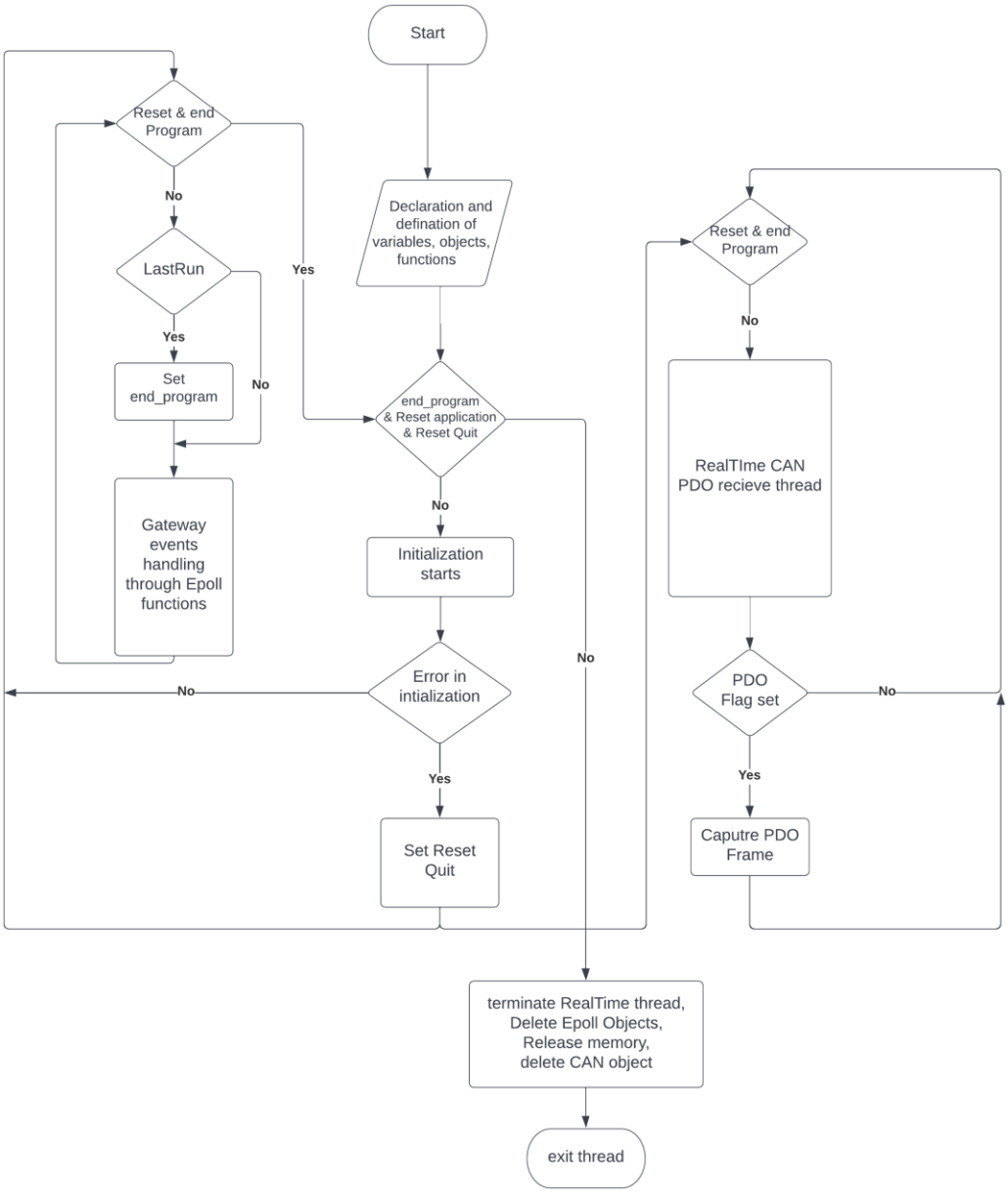



*Figure 74: CAN Sensor thread flow chart*

## 7.4 Pushbutton and Switches thread

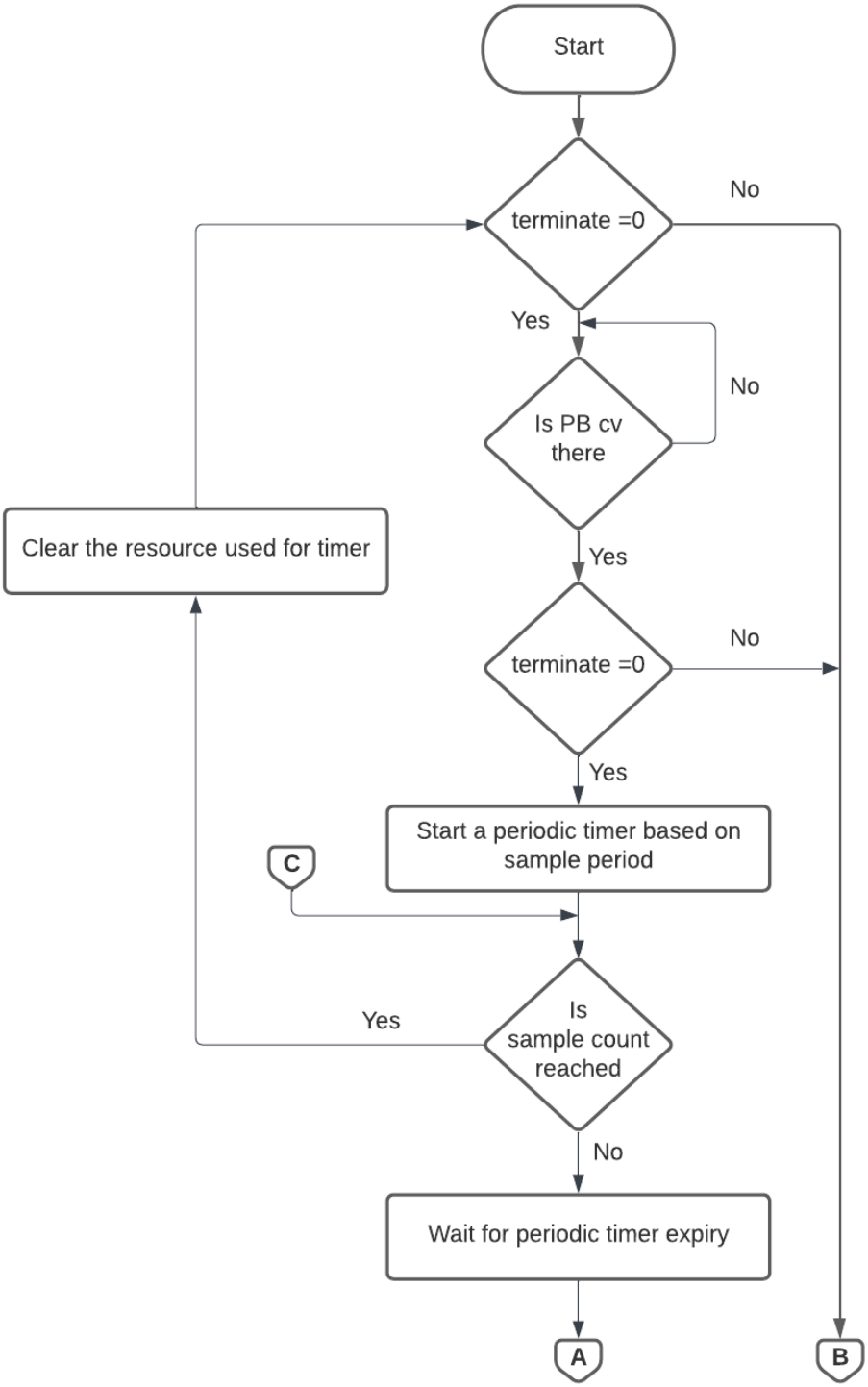



*Figure 75: Flow chart of push button and switches thread part 1*

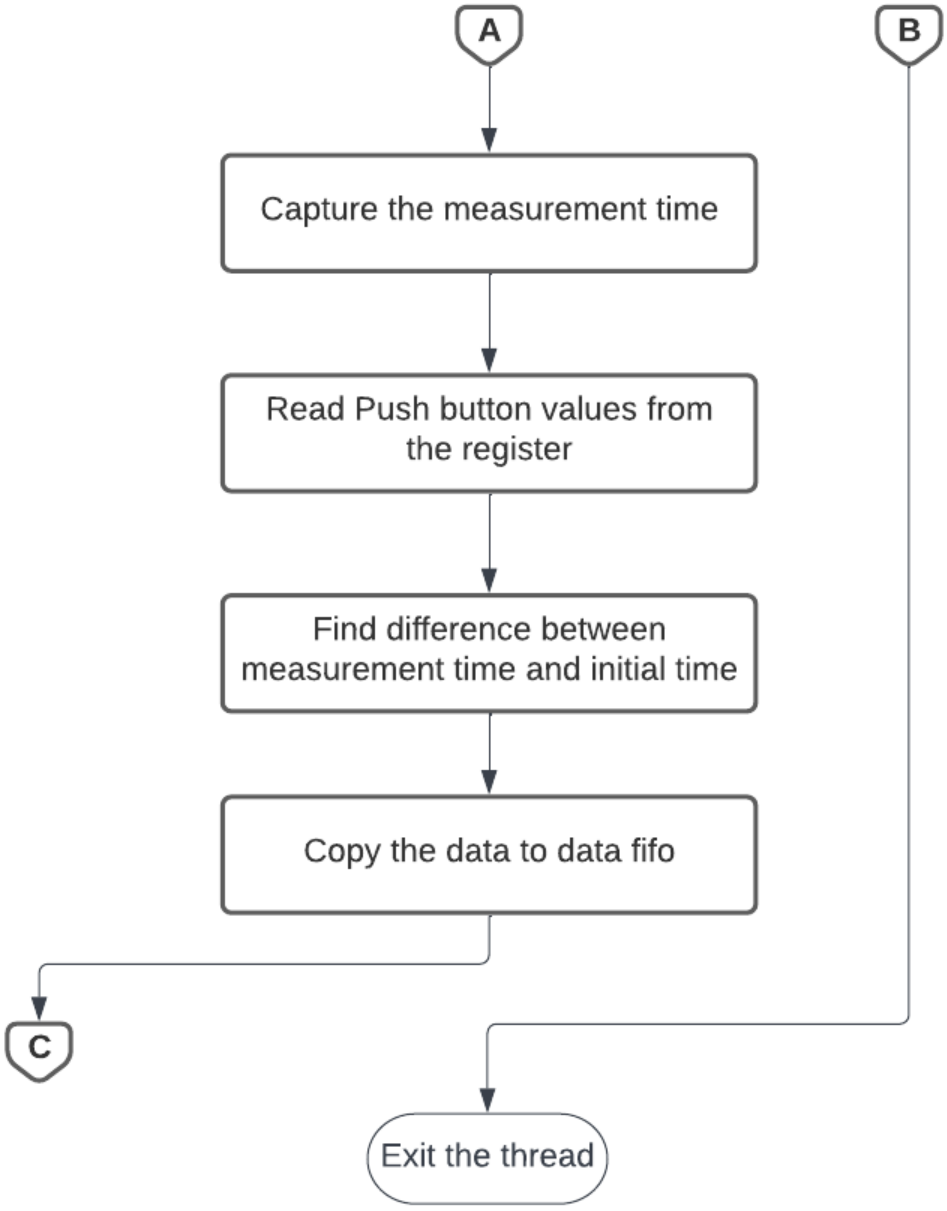



*Figure 76: Flow chart of push button and switches thread part 2*

Considering push buttons present on the Zynq board as a digital sensor, a thread has been employed for data acquisition. The logical implementation is similar to the resistance sensor thread.

Once the thread starts execution, it uses the global flag "*terminate*" to check whether the thread need to be terminated or not. And an infinite loop is executed based on this flag. In order to avoid CPU usage during this infinite loop, a thread conditional variable is used to wait for the measurement request. Once the request for measurement is received, the "*terminate*" flag is checked again. This is to avoid the thread conditional variable missing the flag while in wait condition.

A periodic timer is created in this thread based on the sampling rate the user has requested. There is no limitation for the sampling rate, like resistance measurement. A loop based on the sample count requested is started and the thread waits for the time expiry inside the loop. Once the timer expires the current time is captured and the register stored with the push button values is read. Here also the measurement time captured is subtracted with initial time to reduce the number of bits. When both sample value and time stamp is available, the data is copied to the data FIFO for transmission. Once the sample count loop ends the file pointer for timer creation is closed and thread waits for another measurement request. When the thread receives a termination request, it exits the thread safely.

The thread for data acquisition of switches uses exactly same flow with different variables. If any further scope of expansion is required with a greater number of sensors, the user can reuse same thread for data acquisition.

## 7.5 Data transfer thread

The data transfer thread is responsible for transmission of the measurement data to the client. It runs a periodic timer which expires at every 1ms. It uses the global flag "*terminate*" to check whether the thread need to be terminated or not. And an infinite loop is executed based on this flag. The thread waits for the 1ms timer expiry inside the loop. Once the timer expires, it checks whether any data is there to be transmitted. Once the data present is completely transmitted, it resets the data pointer and waits for another expiry of timer. If terminate flag is one, then the thread exits after clearing the resources used for timer.

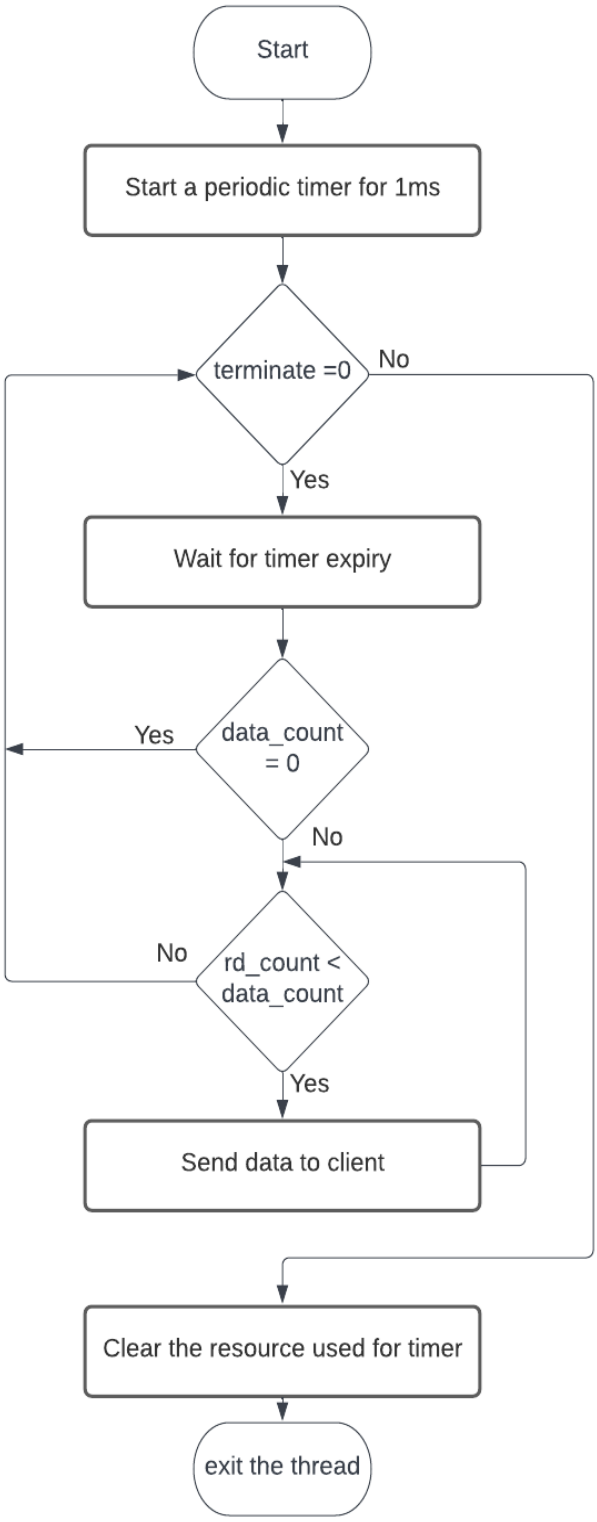



*Figure 77: Flow chart of data transfer thread*

## 7.6 User Interface Thread

The User Interface (UI) module thread is subject to interact with user using the temperature management gateway application (TMNG). The user at client end of the application has control over selecting sensor and choosing the sampling rate.

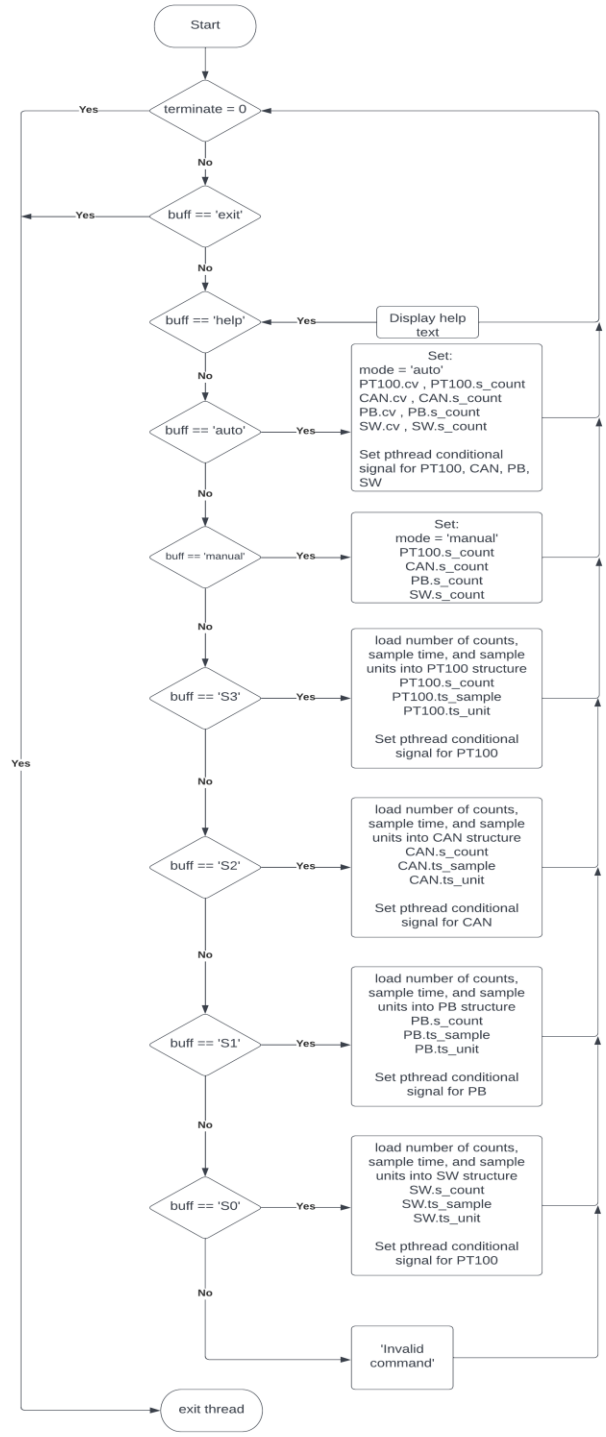



*Figure 78: User Interface (UI) module flow chart*

There are four following set of sensors that our application supports:

- Switches (Digital Sensor) – **S0**
- Push button (Digital Sensor) – **S1**
- CAN Sensor – **S2**
- Resistive MAX31685 Pmod Module (Custom developed sensor/ PT100) – **S3**

Additionally, this thread also gives control over the mode of application. The application provides user to switch between **Auto** and **Manual** mode. In Auto mode, the application display values to a pre-defined sampling rate. In Manual mode, it gives flexibility to set number of counts, sampling time and even unit to sampling rate of individual sensor. Below is the flow chart explaining how UI module is implemented inside the TMNG application.

# 8. GUI Development at Client Side

The GUI is a Java based application made for the client to control and connect with the server. The GUI is used as a client side for the TCP/IP protocol as discussed earlier. The server is a device that is listening for the client requests to come in. In this project, the TCP client has been implemented as a GUI application for graphical representation of the received measurement values from the Network Gateway. The client sends in the required commands to the server end to connect with the Network Gateway and allow the particular sensors to operate and return back the measurement values to be displayed in the GUI in form of a graph.

## 8.1. GUI in Java

Java is a platform independent, and only the operating system specific JVM (Java Virtual Machine) needs to be installed, for the byte code (program code) to execute. Here, the application has used Linux's JVM.

The GUI has been built using Window Builder editor. The editor is built as a plug-in to Eclipse, it is composed of SWT Designer and Swing Designer and makes it very easy to create such interfaces.

The following major components of the editor are the two different views of an application.

- Design view: The user can drag and drop components onto the design canvas. The placement and size, within the constraints of the container's layout manager, of the component can be manipulated. The palette contains components, such as panels, buttons, text fields, etc. There is a hierarchal list, basically a tree-like representation of all the GUI elements that are currently being used in the design. Containers such as panels and frames are shown as parent nodes to the components that they contain.

- Source view: This contains the auto – generated Java code of the design view, the designer has the flexibility to change the orientation of the blocks by manipulating the code or can add features, functionality and back-end data processing which is not possible in design view.

## 8.2. Design of the GUI

The MNGwy GUI consists of 13 elements used in a hierarchal order for the visual display. All the components are as following.

**JFrame:** The *javax.swing.JFrame* class is a type of container which inherits the *java.awt.Frame* class. JFrame works like the main window where components like labels, buttons, textfields are added to create a GUI.

a) **getContentPane():** A container has several layers in it. You can think of a layer as a transparent film that overlays the container. In Java Swing, the layer that is used to hold objects is called the content pane. Objects are added to the content pane layer of the container. The *getContentPane()* method retrieves the content pane layer so that you can add an object to it. The content pane is an object created by the Java run time environment. You do not have to know the name of the content pane to use it. When you use *getContentPane()*, the content pane object then is substituted there so that you can apply a method to it.

i. **TextField_cmd:** A lightweight component that allows the editing of a single line of text. *CommandHandler(textField_cmd.gerText())* is the function added in as an event handler used to get any text written in the *TextField_cmd*. To be used as a command from the client to be sent to the server.

ii. **ScrollPane:** A container class which implements automatic horizontal and/or vertical scrolling for a single child component.

I. **TextPane:** JTextPane is a subclass of JEditorPane class. JTextPane is used for styled document with embedded images and components. It is text component that can be marked up with attributes that are represented graphically. Used to print all the incoming text from the server.

iii. **panel_canv:** JPanel, a part of the Java Swing package, is a container that can store a group of components. The main task of JPanel is to organize components, various layouts can be set in JPanel which provide better organization of components. In our project *panel_canv* is inherited with the *CanvasW* class, which is a separate java file. It shows that if the client is connected to the server or not, also it contains the visual display of the all the sensors connected according to the client's sent commands to the server. All the sensors last updated value, timestamp and the sampling period at which the sensor is sending the data is displayed on the *panel_canv* as well, giving a complete visual of all the sensors connected in the Network Gateway.

iv. **panel_SW:** JPanel, as stated in *panel_canv*. It is inherited with *org.knowm.xchart.XChartPanel* class. XChart is a light-weight and convenient library for plotting data designed to go from data to chart in the least amount of time possible and to take the guess-work out of customizing the chart style. In the project *panel_SW* is used to plot the value of switches set on the Zed Board represented as Sensor 0 (S0) at the particular time stamp on the graph.

v. **panel_PB:** As explained in *panel_SW*, in the project it used to plot the Push Button pressed on the Zed Board represented as Sensor 1 (S1) at the particular time stamp on the graph.

vi. **panel_CAN:** As explained in *panel_SW*, in the project it used to plot the measurement values sent by the CAN device represented as Sensor 2 (S2) at the particular time stamp on the graph.

vii. **panel_RTD:** As explained in *panel_SW*, in the project it used to plot the measurement values sent by the MAX31865 device represented as Sensor 3 (S3) at the particular time stamp on the graph.

b) **menuBar:** JMenuBar is a part of Java Swing package inherited with the *javax.swing.JMenuBar* class . JMenuBar is an implementation of menu bar. It contains one or more JMenu objects, when the JMenu objects are selected they display a popup showing one or more JMenuItems.

i. **mnFile:** An implementation of a menu named *'File'* in the project - a popup window containing JMenuItem's that is displayed when the user selects an item on the JMenuBar.

I. **mntmExit:** An implementation of an item in a menu named *'Exit'* in the project. A menu item is essentially a button sitting in a list. When the user selects the "button", the action associated with the menu item is performed. A JMenuItem contained in a JPopupMenu performs exactly that function. In the project the event handler added with the 'Exit' button is to terminate the GUI application with command *System.exit(0)*.

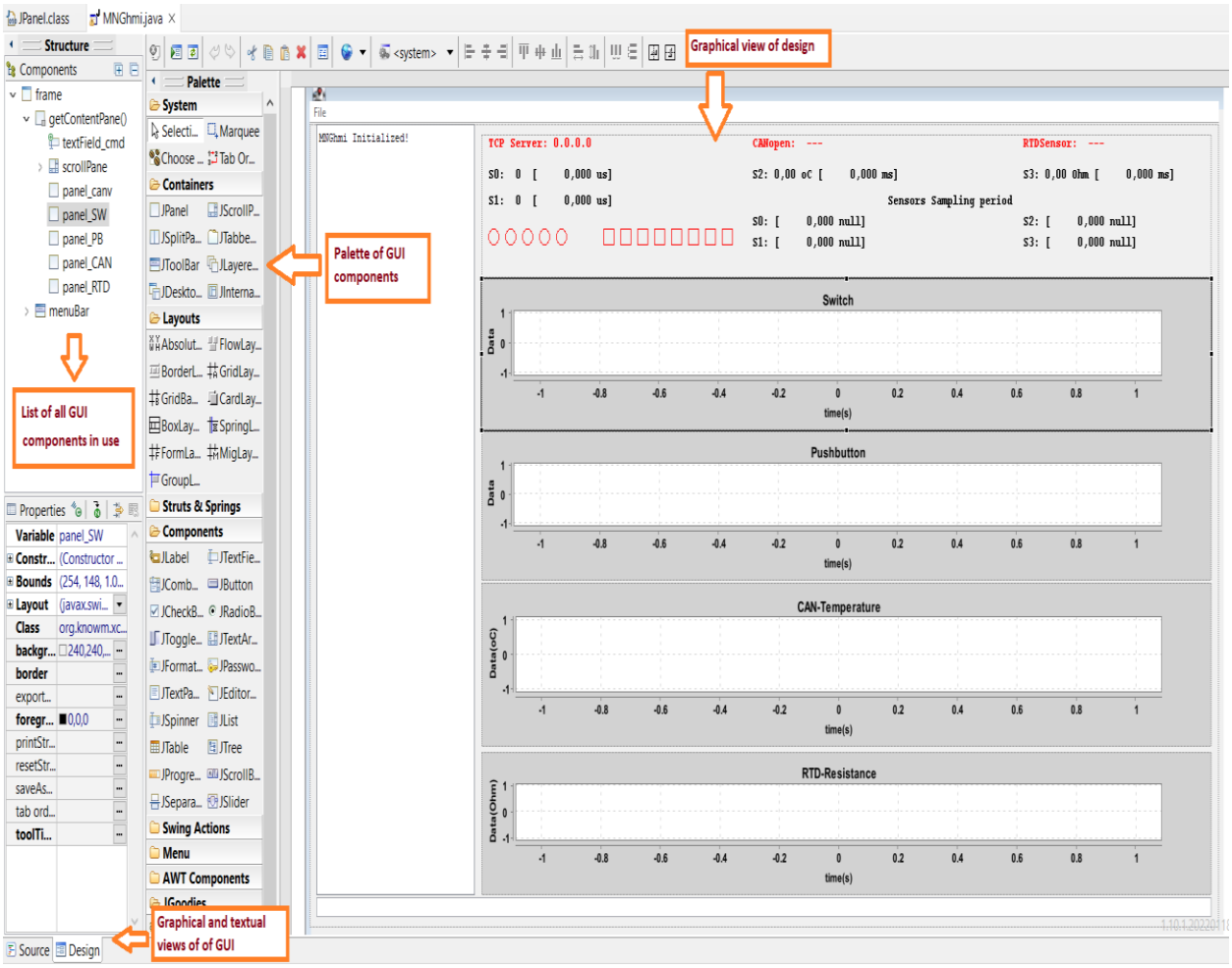
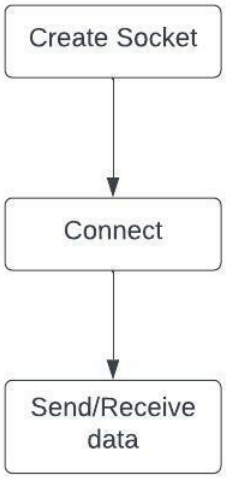



*Figure 79: Design view of Measurement Network Gateway GUI*

## 8.3. GUI Client Application

The GUI application is built to act as a client end for the TCP/IP protocol. The TCP client provides a way for connecting, sending, and receiving stream data over a network. The complete explanation is provided in the TCP/IP section.

The three basic stages the client performs to connect with the server and send/receive data are as follows.




The TCP/IP protocol is designed such that each computer or device in a network has a unique "IP Address" (Internet Protocol Address) and each IP address can open and communicate over up to 65535 different “ports” for sending and receiving data to or from any other network device. The IP Address uniquely identifies the computer or device on the network and a "Port Number" identifies a specific connection between one computer or device and another (i.e between two IP Addresses). A TCP/IP "port" can be thought of as a private two-way communications line where the port number is used to identify a unique connection between two devices. The concept is very similar to any other type of port on your PC (serial, parallel, etc) except that instead of having a physical connection, the TCP/IP protocol creates a "virtual IP port" and the network hardware and software is responsible for routing data in and out of each virtual IP port.

In the project the IP Address for the device is given as a hostname using the string variable *Host_Name = "172.20.48.31"*. The port used for sending and receiving the data is given as *Port_Num = 50012*.

The client creates a socket, the code used to perform the following function is as follows

```
Socket socket = new Socket(Host_Name, Port_Num)
```

If the IP Address is as of the device which is **“172.20.48.31”** and the Port Number on both the server and client end is same that is **“50012”** the connection is established and the client can start to send and receive data.

In case the server is not found and connection is not established or I/O exception has occurred the error message is printed by the exception handler. The code for the exception is as follows.

```
catch (UnknownHostException ex) {
        System.out.println("Server not found: " + ex.getMessage());
        }
catch (IOException ex) {
System.out.println("I/O error: " + ex.getMessage());
```

```
}
```

Once the connection is established with server the client can start to read and write the data, for this input and output streams are used. Input stream is used to read the data form the server and output stream is used to write the data to the server.

```
/* parameter to write to the server */
output = socket.getOutputStream();
writer = new PrintWriter(output, true);
```

The *getOutputStream()* method of Java Socket class returns an output stream for the given socket. If you close the returned OutputStream then it will close the linked socket. It Returns an output stream for writing bytes to this socket and it throws an I/O Exception if an I/O error occurs when creating the output stream or if the socket is not connected. *PrintWriter* prints formatted representations of objects to a text-output stream. This class implements all of the print methods found in *PrintStream*. It does not contain methods for writing raw bytes, for which a program should use un-encoded byte streams. In our case we have used it to use the print method of *println*.

```
/* parameter to receive the string from the server */
input = socket.getInputStream();
reader = new BufferedReader(new InputStreamReader(input));
```

The *getInputStream()* method of Java Socket class returns an input stream for the given socket. If you close the returned *InputStream*, then it will close the linked socket. It return the *getInputStream()* returns an input stream for reading bytes from this socket and it throws an I/O Exception if an I/O error occurs while creating the input stream or if this socket is closed or the given socket is not connected. Input is an *InputStream*, *InputStreamReader* is created to read the bytes from the input and then it is wrapped in *BufferedReader* used to implement the function of *readLine()* in the project. The data received from the server stored in the reader is transferred in a ring buffer. The ring buffer is sent to the GUI using the *PopMessage()* function.

## 8.4. GUI Functional Explanation

The GUI performs multiple functions to be able to show the graphical display of the Network Gateway. Once the application has started, a connection is formed between the client and the server end as discussed previously. To be able to form this connection a command of “tcps” is entered in the *textField_cmd* section of the GUI which allows it to invoke the event handler and use the *CommandHandler(String cmds)* function. In this function it checked that if the correct command is entered to establish the connection. If it is true the following code allows it to establish the connection.

```
hostname = "172.20.48.31";
TcpOBJ.tcp_conn(hostname);
```

*TcpOBJ.tcp_conn(hostname)* is a function in the TCPClient java file which establishes the connection between client and server and prints an error if it is not connected as discussed in the previous section. The complete flow chart of the GUI application is shown in Figure 80.

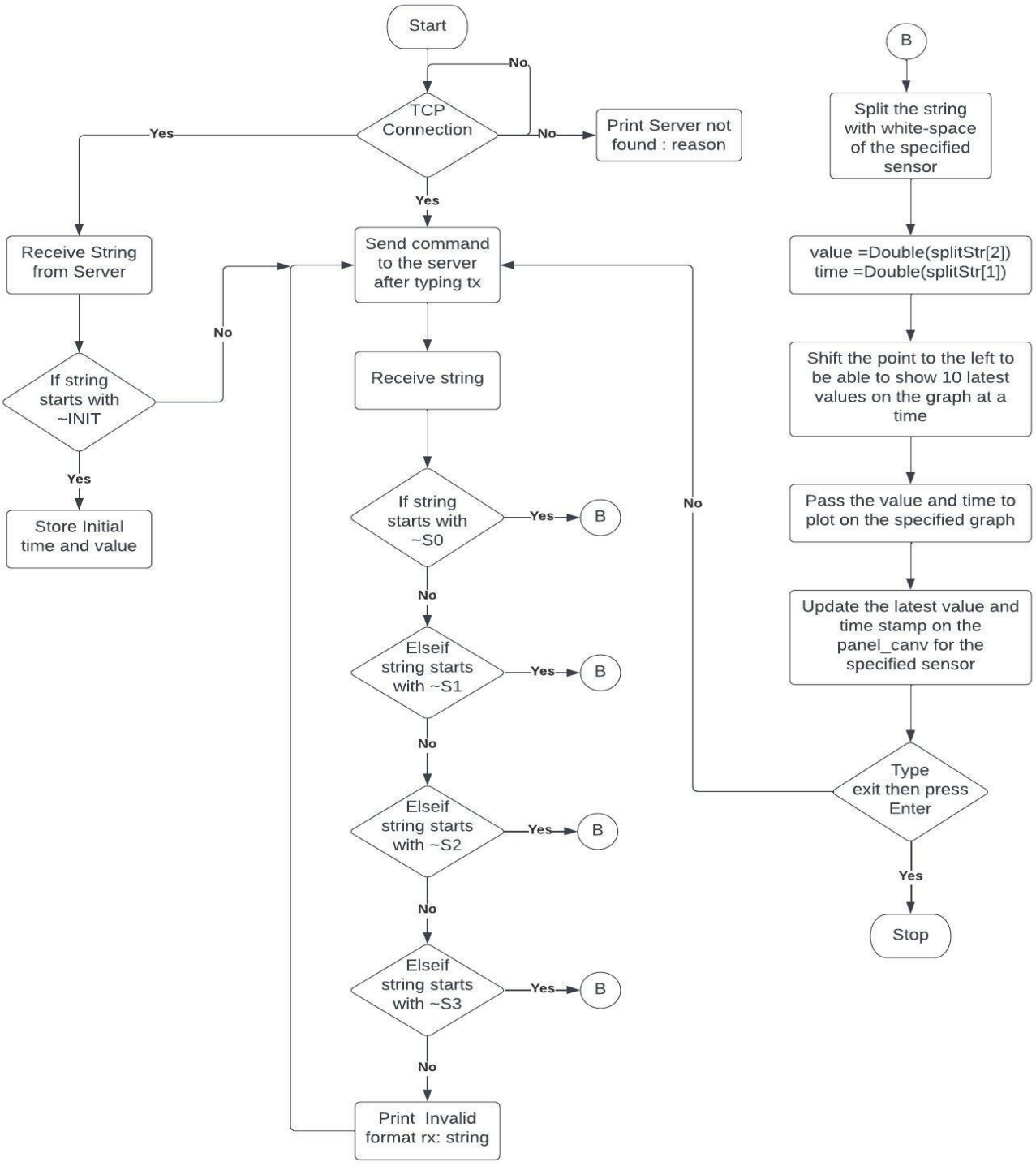



*Figure 80: Flow Chart of the GUI Application*

A ring buffer string array *SBuff* of the size 33 is used to receive any string from the server. Once the connection is established the server sends an initial time and the initial value which is zero, the initial time is the actual time the Gateway application as started. The client application just stores the value in a variable if it is needed for any processing.

All the command that can be entered in the GUI for the server uses the *TcpOBJ.TcpSend(txstr)* function to send the command to the server. The commands are only sent if the they are written after **"tx".** The following commands can be sent to the server.

- Sensor_type No_iterations Sampling_rate Unit (Example: S0 10 100 ms)
  S0: Switches, S1: PushButtons, S2: CAN-Temperature, S3: RTD-Resistance
- auto (to invoke all the sensors and continuously send the data)
- manual (to stop the auto mode)
- exit (to close the server end)

The **"help"** can be entered to show all the command formats as shown in Figure 81.

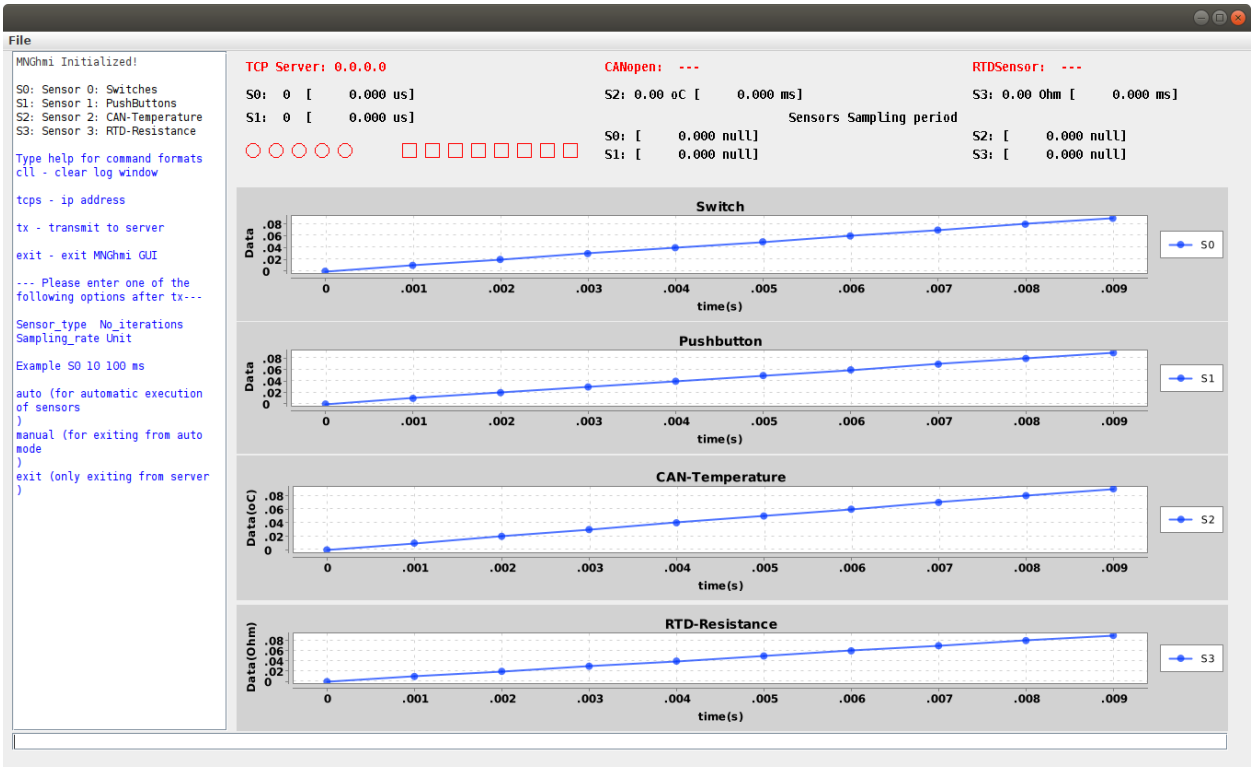



*Figure 81: GUI (help command)*

Once the correct command is entered the regarding the server, it sends the string in a specific format. The string is stored in the ring buffer, in the main program a timer *DispUpdate_Timer* is executed which calls the *UpdateDynamic()* function again and again after a certain period of time.

The *UpdateDynamic()* calls the *PopMessage()* function from the TCPClient and gets the string sent from the server. The string is stored in *rx_str*, the *rx_str.startsWith("any character")* function is used to check for the type of sensor the data is coming for, the string always comes in a specific

format. The string format consists of "~Sensor_type: Time_stamp Value" i.e. "~S0: 38164307.459 145.000". The string stored in rx_str is splitted into an array of string with a whitespace using the following code.

```
//splitting of string at whitespace
String[] splitStr = rx_str.split(" ");
```

This is done to separate out the time stamp and the value sent of the sensor. The string is converted into double by using the following code.

```
// converting string into Double
mval[i] = Double.parseDouble(splitStr[2]);
mtime[i] = Double.parseDouble(splitStr[1]);
```

A check is put in place just in case a value at an old time stamp is sent by the server at a later moment. The code is repeated in all the sensors value processing separately for data showing on the graph with different sensor *latest_time*. One example of the code used is as follows.

```
// allowing only the new timestamp and data to show on the graph
if(latest_time_SW > mtime[i])
{
continue;
}
else
{
latest_time_SW = mtime[i];
}
```

The data coming in as a time stamp for x-axis and value for the y-axis is to be shown as a point on the graph. As the data is coming in periodically from the server for different sensors. The point is shifted 10 times to the left so that 10 most recent values for the particular sensor can be shown at a time on the graph.

The values are then sent to the particular sensor graph and it can be plotted in a form of series. The code used for plotting the series is as follows.

```
/*Add the sensor data and the time-stamp on the respective graphs*/
SW_Chart.addSeries("S0", Stime[0], Sdata[0]);
PB_Chart.addSeries("S1", Stime[1], Sdata[1]);
CAN_Chart.addSeries("S2", Stime[2], Sdata[2]);
RTD_Chart.addSeries("S3", Stime[3], Sdata[3]);
```

The new values are updated into the series and plotted of the respective graphs by the following code.

```
SW_Chart.updateXYSeries(S, Stime[i], Sdata[i], null);

PB_Chart.updateXYSeries(S, Stime[i], Sdata[i], null);

CAN_Chart.updateXYSeries(S, Stime[i], Sdata[i], null);

RTD_Chart.updateXYSeries(S, Stime[i], Sdata[i], null);
```

*repaint()* function needs to be used after every update in any particular panel for showing the update in the GUI. The *UpdateCANServer* and *UpdateRTDServer* functions are used to show if the particular sensor is connected, it updates the status on the canvas panel at the top. The *UpdateCAN*, *UpdateRTD* functions are used to update the last value and time stamp sent by the CAN or RTD sensor to be shown on the canvas panel. *Update* function is used to update the last value and time stamp sent by the Switches or PushButton to be shown on the canvas panel. *Update_tperiod* function is used to show the sampling period the particular sensor is being sampled on the canvas panel.

## 8.5. GUI Figures for Verification

### 8.5.1. Switches (Sensor 0) Verification

As shown in Figure 82 when the connection is established between server and the client the TCP Server shows the IP Address of the server and the color changes to green.

- The command “S0 10 10 us” is sent to the server shown as sending [S0 10 10 us] on the GUI.
- The last value (shown in hex) and time stamp sent by server for S0 is shown on the top.
- The 8 square boxes glow according to the binary conversion of the value sent by the server for the Sensor 0.
- The Switch graph shows the points according to the value and time stamp sent by the server.
- 145 decimal value hex conversion is 91 and binary conversion is 10010001 as shown in Figure 82.
- The sampling period value with the unit is updated below the Sensors Sampling period on the canvas panel.


*Figure 82: GUI (Switches – Sensor 0)*

### 8.5.2. Push-Buttons (Sensor 1) Verification

As shown in Figure 83 when the connection is established between server and the client the TCP Server shows the IP Address of the server and the color changes to green.

- The command “S1 10 10 us” is sent to the server shown as sending [S0 10 10 us] on the GUI.
- The last value (shown in hex) and time stamp sent by server for S1 is shown on the canvas panel.
- The 5 ovals glows according value associated to the button pressed on the Zed Board sent by the server for the Sensor 1.
- The Pushbutton graph shows the points according to the value and time stamp sent by the server.
- 2 decimal value hex conversion is 2 and binary conversion is 00010 as shown in Figure 8383.
- The sampling period value with the unit is updated below the Sensors Sampling period on the canvas panel.

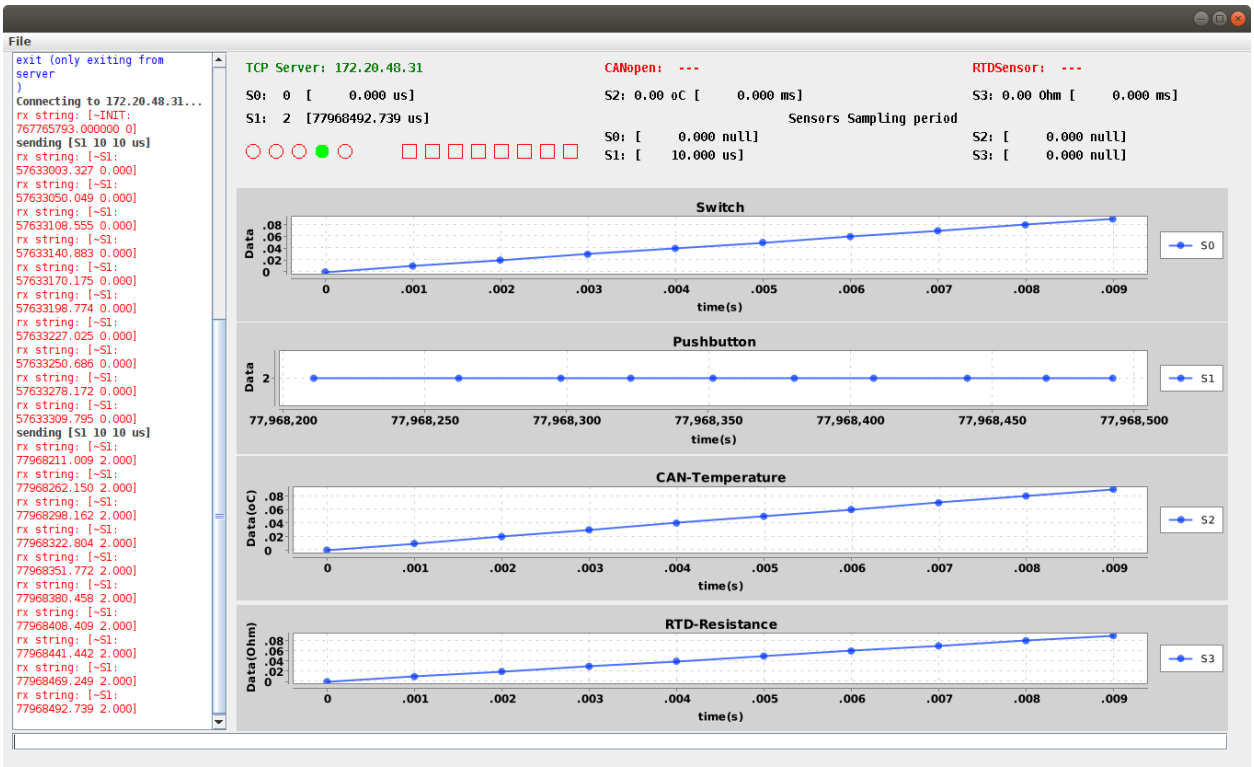



*Figure 83: GUI (Push Buttons - Sensor 1)*

### 8.5.3. MAX31865 RTD (Sensor 3) Verification

As shown in Figure 84 when the connection is established between server and the client the TCP Server shows the IP Address of the server and the color changes to green.

- The RTD Sensor is shown as connected in green color.

- The command “S3 10 100 ms” is sent to the server shown as sending [S3 10 100 ms] on the GUI.
- The last value and time stamp sent by server for S3 is shown on the canvas panel.
- The RTD-Resistance graph shows the points according to the value and time stamp sent by the server.
- The sampling period value with the unit is updated below the Sensors Sampling period on the canvas panel.

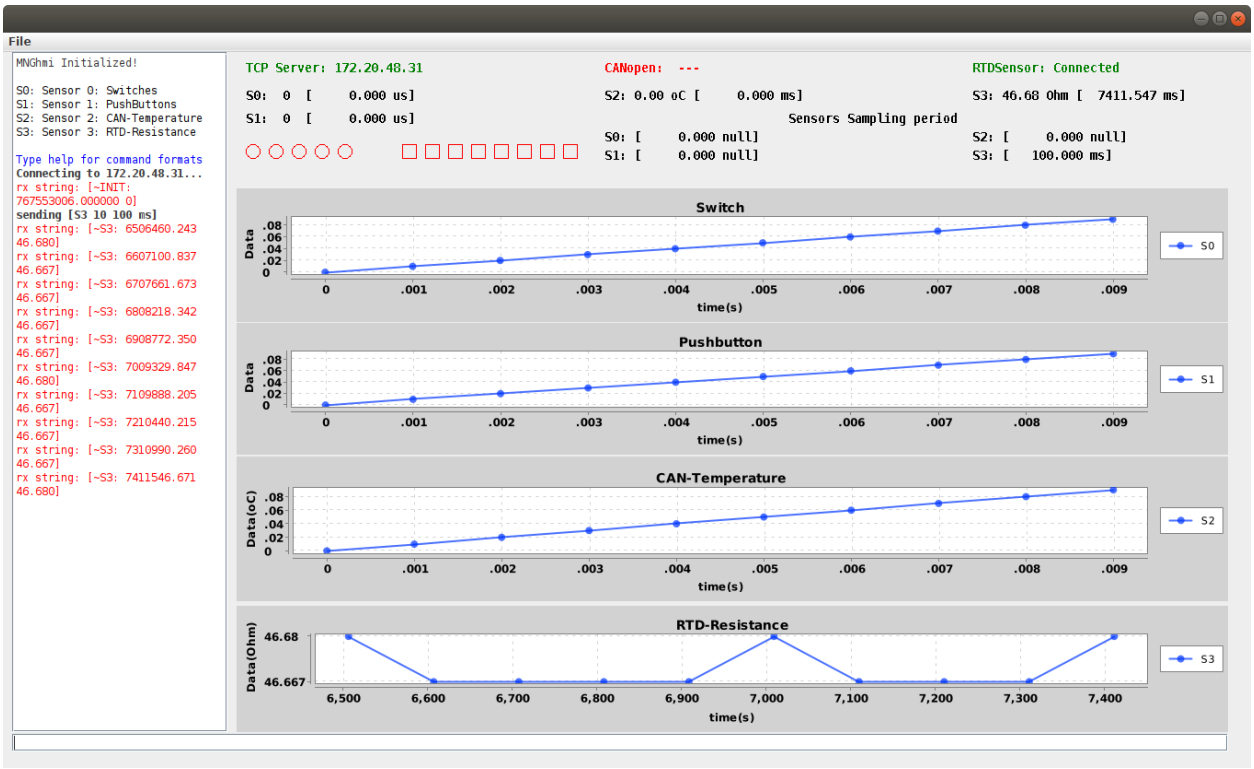



*Figure 84: GUI (RTD Sensor - Sensor 3)*

### 8.5.4. Auto Mode Verification

As shown in Figure 85 when the connection is established between server and the client the TCP Server shows the IP Address of the server and the color changes to green. First all the sensor commands have to execute once to get the sampling period for each sensor. Once it is done by sending the auto command the server starts to send the data from all the sensors at the same time, which inherently shows the multithreading performed in the project.

- The RTD Sensor is shown as connected in green color.
- The command “auto” is sent to the server to execute the auto mode.
- The last value and time stamp sent by the server for the respective sensor is shown on the canvas panel.
- All the graphs show the points according to the value and time stamp of respective sensor sent by the server.

- The sampling period value with the unit is updated below the Sensors Sampling period on the canvas panel.


*Figure 85 : GUI (Auto Mode)*

# 9. Performance Analysis and Results

The validation of the application can be done through the GUI. But for the performance analysis, another approach need to be used since the graphs present in the GUI cannot visualize high speed data. Each of the sample sent from the server contain the sensor name, sensor data and sensor time stamp.

## 9.1. High speed TCP client evaluation for Sampling period jitter analysis

The real time performance analysis is done through a TCP client program which writes the received data into a file. And the file is then processed by C program to find jitter of the sampling period. The difference of time stamp between two adjacent samples are calculated and summarized for jitter.

The below mentioned steps are used for data capture and analysis

1. Launch the temperature measurement gateway, for the TCP server to be active. The data transmission is performed every 1ms by the application.
2. Launch the TCP client program, to receive the data from the server and capture it into a file for post processing. The TCP client requests which sensor data to be sent and at what rate it should be sent. Sample count of 500 is used for all sensors. The data is captured for

sampling rates 200µs, 400µs, 600µs, 800µs and 1ms (except for resistor sensor for which it is 100ms).

3. Post processing is done for the captured file using C, and difference between timestamp of adjacent samples are found out and listed into a table based on comparison with different tolerance levels.

A TCP client is the receiving end in this mode instead of GUI. The analysis is performed for push button, switches and resistor sensor. The experiment is repeated for some iterations to know about the consistency of behavior as well. The capture is repeated for 3 times in the experiment.

The lowest possible sampling rate need to be programmed for this test. The sampling rate to start the measurement is chosen to be 200µs. The reason is because, once we go below a sampling rate of 500µs some unwanted ASCII characters are being introduced to the file. These need to be replaced and then only the file can be processed. And below 200µs sampling rates was not providing any acceptable accuracy based on tolerances. This is a limitation of the timers that are used to configure the sampling rate. The data is captured for 200µs, 400µs, 600µs, 800µs and 1ms.

The C code used to post process consist of opening the file in read binary mode and extract the lines of the file one by one. In this case, the data of interest is the time stamp of sample data. The time stamp is extracted from each line and the difference with the previous sample's time stamp is found. This difference is then compared to the tolerance values mentioned in the table (5%, 2% and 1 %). It is given as sampling period range as well in the table. If time difference value found, is falling in the sampling period range, then it is considered to be an accurate sample. A simple if else statement is used to get the total count of these accurate values according to the tolerances. Then percentage is found with respect to 500 samples. The maximum accuracy that can be achieved is 99.8 since the 1$^{st}$ sample cannot be used.

The table of sensors with sampling period accuracy analysis is given below.

**For 200µs**

*Table 13: Performance analysis for 200µs*

| Number of attempts | Type of the sensor | Expected Sample Period | Tolerance of Sampling Period | Accuracy | Sampling period Range |
|---|---|---|---|---|---|
| 1 | Pushbutton | 200µs | 5% | 96.4% | 200-210 µs |
| | | | 2% | 85.6% | 200-205 µs |
| | | | 1% | 60.8% | 200-202 µs |
| 2 | Pushbutton | 200µs | 5% | 96.2% | 200-210 µs |
| | | | 2% | 92.8% | 200-205 µs |
| | | | 1% | 80.6% | 200-202 µs |
| 3 | Pushbutton | 200µs | 5% | 95.0% | 200-210 µs |
| | | | 2% | 87.4% | 200-205 µs |
| | | | 1% | 76.4% | 200-202 µs |
| **Number of attempts** | **Type of the sensor** | **Expected Sample Period** | **Tolerance of Sampling Period** | **Accuracy** | **Sampling period Range** |
| 1 | Switches | 200µs | 5% | 98.0% | 200-210 µs |
| | | | 2% | 95.6% | 200-205 µs |
| | | | 1% | 90.4% | 200-202 µs |
| 2 | Switches | 200µs | 5% | 99.4% | 200-210 µs |
| | | | 2% | 93.4% | 200-205 µs |
| | | | 1% | 88.8% | 200-202 µs |
| 3 | Switches | 200µs | 5% | 99.4% | 200-210 µs |
| | | | 2% | 98.6% | 200-205 µs |
| | | | 1% | 95.2% | 200-202 µs |

**For 400µs**

*Table 14: Performance analysis for 400µs*

| Number of attempts | Type of the sensor | Expected Sample Period | Tolerance of Sampling Period | Accuracy | Sampling period Range |
|---|---|---|---|---|---|
| 1 | Pushbutton | 400µs | 5% | 99.6% | 400-420 µs |
| | | | 2% | 98.0% | 400-410 µs |
| | | | 1% | 88.2% | 400-404 µs |
| 2 | Pushbutton | 400µs | 5% | 99.8% | 400-420 µs |
| | | | 2% | 99.4% | 400-410 µs |
| | | | 1% | 96.8% | 400-404 µs |
| 3 | Pushbutton | 400µs | 5% | 99.6% | 400-420 µs |
| | | | 2% | 96.8% | 400-410 µs |
| | | | 1% | 69.0% | 400-404 µs |

| Number of attempts | Type of the sensor | Expected Sample Period | Tolerance of Sampling Period | Accuracy | Sampling period Range |
|---|---|---|---|---|---|
| 1 | Switches | 400µs | 5% | 99.4% | 400-420 µs |
| | | | 2% | 99.0% | 400-410 µs |
| | | | 1% | 91.4% | 400-404 µs |
| 2 | Switches | 400µs | 5% | 99.8% | 400-420 µs |
| | | | 2% | 98.4% | 400-410 µs |
| | | | 1% | 94.4% | 400-404 µs |
| 3 | Switches | 400µs | 5% | 99.8% | 400-420 µs |
| | | | 2% | 99.6% | 400-410 µs |
| | | | 1% | 96.0% | 400-404 µs |

**For 600µs**

*Table 15: Performance analysis for 600µs*

| Number of attempts | Type of the sensor | Expected Sample Period | Tolerance of Sampling Period | Accuracy | Sampling period Range |
|---|---|---|---|---|---|
| 1 | Pushbutton | 600µs | 5% | 99.6% | 600-630 µs |
| | | | 2% | 99.6% | 600-612 µs |
| | | | 1% | 98.2% | 600-606 µs |
| 2 | Pushbutton | 600µs | 5% | 99.0% | 600-630 µs |
| | | | 2% | 99.0% | 600-612 µs |
| | | | 1% | 97.2% | 600-606 µs |
| 3 | Pushbutton | 600µs | 5% | 97.8% | 600-630 µs |
| | | | 2% | 95.0% | 600-612 µs |
| | | | 1% | 93.6% | 600-606 µs |
| **Number of attempts** | **Type of the sensor** | **Expected Sample Period** | **Tolerance of Sampling Period** | **Accuracy** | **Sampling period Range** |
| 1 | Switches | 600µs | 5% | 99.2% | 600-630 µs |
| | | | 2% | 99.2% | 600-612 µs |
| | | | 1% | 98.4% | 600-606 µs |
| 2 | Switches | 600µs | 5% | 99.6% | 600-630 µs |
| | | | 2% | 99.0% | 600-612 µs |
| | | | 1% | 95.8% | 600-606 µs |
| 3 | Switches | 600µs | 5% | 99.6% | 600-630 µs |
| | | | 2% | 99.4% | 600-612 µs |
| | | | 1% | 97.8% | 600-606 µs |

**For 800µs**

*Table 16: Performance analysis for 800µs*

| Number of attempts | Type of the sensor | Expected Sample Period | Tolerance of Sampling Period | Accuracy | Sampling period Range |
|---|---|---|---|---|---|
| 1 | Pushbutton | 800µs | 5% | 99.6% | 800-840 µs |
| | | | 2% | 97.2% | 800-816 µs |
| | | | 1% | 97.2% | 800-808 µs |
| 2 | Pushbutton | 800µs | 5% | 99.6% | 800-840 µs |
| | | | 2% | 99.6% | 800-816 µs |
| | | | 1% | 99.6% | 800-808 µs |
| 3 | Pushbutton | 800µs | 5% | 99.6% | 800-840 µs |
| | | | 2% | 99.2% | 800-816 µs |
| | | | 1% | 99.2% | 800-808 µs |
| **Number of attempts** | **Type of the sensor** | **Expected Sample Period** | **Tolerance of Sampling Period** | **Accuracy** | **Sampling period Range** |
| 1 | Switches | 800µs | 5% | 99.6% | 800-840 µs |
| | | | 2% | 97.4% | 800-816 µs |
| | | | 1% | 97.2% | 800-808 µs |
| 2 | Switches | 800µs | 5% | 99.4% | 800-840 µs |
| | | | 2% | 95.4% | 800-816 µs |
| | | | 1% | 94.2% | 800-808 µs |
| 3 | Switches | 800µs | 5% | 99.6% | 800-840 µs |
| | | | 2% | 99.6% | 800-816 µs |
| | | | 1% | 99.2% | 800-808 µs |

**For 1ms**

*Table 17: Performance analysis for 1ms*

| Number of attempts | Type of the sensor | Expected Sample Period | Tolerance of Sampling Period | Accuracy | Sampling period Range |
|---|---|---|---|---|---|
| 1 | Pushbutton | 1ms | 5% | 99.6% | 1000-1050 µs |
| | | | 2% | 99.2% | 1000-1020 µs |
| | | | 1% | 99.2% | 1000-1010 µs |
| 2 | Pushbutton | 1ms | 5% | 99.6% | 1000-1050 µs |
| | | | 2% | 99.6% | 1000-1020 µs |
| | | | 1% | 99.6% | 1000-1010 µs |
| 3 | Pushbutton | 1ms | 5% | 99.6% | 1000-1050 µs |
| | | | 2% | 99.2% | 1000-1020 µs |
| | | | 1% | 99.2% | 1000-1010 µs |

| Number of attempts | Type of the sensor | Expected Sample Period | Tolerance of Sampling Period | Accuracy | Sampling period Range |
|---|---|---|---|---|---|
| 1 | Switches | 1ms | 5% | 99.6% | 1000-1050 µs |
| | | | 2% | 99.6% | 1000-1020 µs |
| | | | 1% | 99.2% | 1000-1010 µs |
| 2 | Switches | 1ms | 5% | 99.6% | 1000-1050 µs |
| | | | 2% | 99.4% | 1000-1020 µs |
| | | | 1% | 99.4% | 1000-1010 µs |
| 3 | Switches | 1ms | 5% | 99.6% | 1000-1050 µs |
| | | | 2% | 99.6% | 1000-1020 µs |
| | | | 1% | 99.6% | 1000-1010 µs |

**For 100ms**

*Table 18: Performance analysis for 100ms*

| Number of attempts | Type of the sensor | Expected Sample Period | Tolerance of Sampling Period | Accuracy | Sampling period Range |
|---|---|---|---|---|---|
| 1 | Resistor sensor | 100ms | 5% | 99.6% | 100-110 ms |
| | | | 2% | 99.6% | 100-102 ms |
| | | | 1% | 99.6% | 100-101 ms |
| 2 | Resistor sensor | 100ms | 5% | 99.6% | 100-110 ms |
| | | | 2% | 99.6% | 100-102 ms |
| | | | 1% | 99.6% | 100-101 ms |
| 3 | Resistor sensor | 100ms | 5% | 99.6% | 100-110 ms |
| | | | 2% | 99.6% | 100-102 ms |
| | | | 1% | 99.6% | 100-101 ms |

The tables indicate that for a minimum sample period of 200µs, the application is able to achieve an accuracy above 90%. But this is not a consistent result, the accuracy is fluctuating unpredictably for different tolerances. This is the case for 400µs, 600µs and 800µs as well. This is an issue of making use of timers in application. The normal scheduling does not give real time accuracy, it might decide to wake up the timer even 10µs later, because it had some other work to do at that time. If it is required to wake up every 200µs exactly, then task need to be run as real time. Then the kernel will wake up every 200µs without fail. Unless a higher priority real time task is busy doing stuff. But it can be observed that once we reaches the millisecond granularity the accuracy improved significantly even for the lower tolerance rates.

## 10. Limitations of Application

1. The first limitation of the application is with respect to the timer logic used. These timers only allow accuracy over millisecond granularity as seen from the analysis. When the sensor samples need to be collected over microsecond sampling rate, the jitter increases. This makes the sample collection time unpredictable. One way to overcome this issue is to make the timers operate at real time. But this is a challenge since the timers are present in threads. By accident if a thread is assigned higher priority, all other threads including the

main thread will wait on the higher priority thread. And this stops the execution of the program.

2. Limitation in the number of sensors supported. The application right now does data acquisition for 3 sensors. The application is designed in such a way that additional sensors can be added as a separate thread, so that it doesn't disturb the current functionality. Although technically speaking there is no limit in the threads supported by a process, this will be determined by the memory capability of the system. The Zynq board has a limited amount of memory, hence there will be a limit for the sensors that can be supported by the system.

3. Data transmission rate and overflow of samples at the server side. In current design the data transmission thread is repeated at every 1ms. With this value in mind, it requires at least 30 data buffers for holding the data if the sampling rate is 100µs and three sensors are present. Otherwise, the sampled data will overflow and overwrite the data FIFO used to store data. If the data transmission frequency is lower, the data FIFO size need to be increased and vice versa. Since the Zynq 7000 has limited amount of memory, there should be a proper balance between the data transmission frequency and the minimum sampling rate required by the user.

## 11. Conclusion

The Measurement Network Gateway device is successfully implemented and validated by means GUI program. The performance analysis is performed with TCP client program. Almost all the requirement of the project has been completed. Although CAN sensor was not successfully implemented, the other sensors has been interfaced. The sensors are sampled at the given sampling rate for the given sample counts. Using various sampling rates, the application is evaluated to find the jitter or uncertainty in the sampling periods. Various types of sensors were made and tested for temperature and level measurement of nitrogen tanks. They are tested and calibrated with different test setups.

# 12. References

[1] NXP, "NXP Gateway," [Online]. Available: https://www.nxp.com/docs/en/white-paper/AUTOGWDEVWPUS.pdf.

[2] DGILENT, "ZedBoard," [Online]. Available: https://digilent.com/reference/programmable-logic/zedboard/start.

[3] "CANopen device model," [Online]. Available: https://www.canopensolutions.com/english/about_canopen/CANopen-device-model.shtml#

[4] JUMO, "JUMO CANtrans T," [Online]. Available: https://www.jumo.de/web/products/apps/productdetailpage?pdpId=902910.

[6] A. Xilinx, "AMD Xilinx," [Online]. Available: https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html

[5] https://datasheets.maximintegrated.com/en/ds/MAX31865.pdf

[7] CAN [online], Available: https://copperhilltech.com/a-brief-introduction-to-controller-area-network/

[8] Introduction to CAN [online], Available: https://www.ti.com/lit/an/sloa101b/sloa101b.pdf

[9] History of CAN [online], Available:https://en.wikipedia.org/wiki/CANopen

[10]CANopenObjectdictionary[online], Available:https://canopennode.github.io/CANopenNode/md_doc_objectDictionary.html

[11]CANopenDevicearchitecture [online],

Available: https://www.can-cia.org/can-knowledge/canopen/device-architecture/

[12]CANopen[online], Available:https://cache.industry.siemens.com/dl/files/771/109479771/att_993267/v1/109479771_CANopen_Tutorial_V20_en.pdf

[13] CANopen communication objects [online], Available: https://doc.ingeniamc.com/emcl2/command-reference-manual/communications/canopen-protocol/canopen-objects/nmt

[14] NMThearbeatprotocol[online],

Available: https://www.canopensolutions.com/english/about_canopen/Heartbeat-service.shtml

[15] Special function Protocol [online], Available:https://www.can-cia.org/can-knowledge/canopen/special-function-protocols/

[16] PDO protocol [online], Available: https://www.can-cia.org/can-knowledge/canopen/pdo-protocol

[17] SYNC Protocol [online], Available :https://www.can-cia.org/can-knowledge/canopen-fd/sync-protocol

[18] SPI Protocol [online], Available:https://en.wikipedia.org/wiki/Serial_Peripheral_Interface

[19] SPI Protocol [online], Available:https://www.circuitbasics.com/basics-of-the-spi-communication-protocol/

[20] S. Roberts, "KE2 Therm Solutions," January 2011. [Online]. Available: https://ke2therm.com/wp-content/uploads/2015/08/w-5-4_Understand_Ethernet_Protocol.pdf.

[21] "Digital Guide IONOS," 2020. [Online]. Available: https://www.ionos.de/digitalguide/server/knowhow/tcp-vorgestellt/.

[22] "Steve's Internet Guide," 2021. [Online]. Available: http://www.steves-internet-guide.com/tcpip-ports-sockets/.

[23] "simplest coding," 2010. [Online]. Available: http://simplestcodings.blogspot.com/2010/07/tcp-server-client.html.

[24] M. Kerrich, "Man7," Jambit Gmbh, 2022. [Online]. Available: https://man7.org/linux/man-pages/man2/socket.2.html.

[25] "Beep Volume," 2020. [Online]. Available: http://beepvolume.com/async-programming/2020/multithreading/.

[26] M. Kerrich, "Man7," Jambit Gmbh, 2022. [Online]. Available: https://man7.org/linux/man-pages/man2/bind.2.html.

[27] M. Kerrich, "Man7," Jambit Gmbh, 2022. [Online]. Available: https://man7.org/linux/man-pages/man2/listen.2.html.

[28] https://github.com/knowm/XChart

[29] https://www.tabnine.com/code/java/methods/org.knowm.xchart.XChartPanel/%3Cinit%3E

[30] https://www.taltech.com/datacollection/articles/a_brief_overview_of_tcp_ip_communications

[31] https://www.javatpoint.com/java-socket-getinputstream-method

[32] https://www.javatpoint.com/java-socket-getoutputstream-method

[33] https://github.com/CANopenNode/CANopenNode

[34] https://github.com/CANopenNode/CANopenLinux

[35] EMCY Protocol [online], Available: https://www.can-cia.org/can-knowledge/canopen-fd/emcy-write-protocol/#:~:text=The%20EMCY%20write%20protocol%20is,only%20once%20per%20error%20event.

[36] SDO protocol [online], Available: Service Data Object (SDO) Protocol

[37] NMT protocol [online], Available: Guarding function

[38] Image [online] Retrived from: https://directory.eoportal.org/web/eoportal/satellite-missions/b/beesat-1

[39] CAN [online] Available: https://images.app.goo.gl/hsYtVm3KFbVZUZhC9

# 13. Appendix

## SOURCE CODE

```
/*---------------------------------------------------------------------------
---
 Name        : TMNG_Main.c
 Authors     : Anees Ahmed Zuberi ,Khizar Akhtar , Nikhil Narayanan
 Version     :
 Copyright   :
 Description : Main file
---------------------------------------------------------------------------
*/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <fcntl.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
#include <arpa/inet.h>
#include <signal.h>


#include "res_sensor.h"
#include "can_sensor.h"
#include "uimodule.h"
#include "data_transfer.h"


#define PORT 50012


unsigned int terminate;
pthread_mutex_t data_fifo_lock;
pthread_mutex_t zynq_lock;
unsigned int data_wr_count;
char data_fifo[100][50];
struct timespec init_time;
SensParam_struct PT100,CAN,PB,SW;
pthread_mutex_t board_lock;
char mode[7]="\0";

int connfd,fd;

void sighandler(int);

//The below function will be used to catch Ctrl+C and terminate safely
```

```
void sighandler(int signum){
   printf("\nCaught Ctrl+C signal %d, Program exit.\n", signum);
   terminate = 1;
   lastRun = 1;
   pthread_cond_signal(&PT100.cv);
   pthread_cond_signal(&CAN.cv);
   pthread_cond_signal(&PB.cv);
   pthread_cond_signal(&SW.cv);
   exit(1);
}

int main(){

    pthread_t res_sensor_thread_tID;
    pthread_t uimodule_thread_tID;
    pthread_t CANOpen_thread_tID;
    pthread_t data_transfer_thread_tID;
    pthread_t PB_transfer_thread_tID;
    pthread_t SW_transfer_thread_tID;

    int rc,sockfd,true;
    unsigned int len,cli;
    struct sockaddr_in servaddr;
    void *status;
    double start_time;
    terminate = 0;
    data_wr_count = 0;
    struct sched_param schedparm_main;

    memset(&schedparm_main, 0, sizeof(schedparm_main));
    schedparm_main.sched_priority = 1; // lowest rt priority
    sched_setscheduler(0, SCHED_FIFO, &schedparm_main);

    signal(SIGINT, sighandler);


    fd = open("/dev/measCdd", O_RDWR);
    if(fd < 0){
      perror("Error: Failed to open the device");
    }

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if(sockfd == -1) {
      printf("Error: Socket creation failed\n");
      exit(0);
    } else {
      printf("Info: Socket creation successful\n");
    }

    true = 1;
    if (setsockopt(sockfd,SOL_SOCKET,SO_REUSEADDR,&true,sizeof(int)) == -1)
    {
      printf("Error: Setting socket options failed\n");
      exit(0);
    }
```

```
    printf("Info: TcpServerF: %d\n", sockfd);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = inet_addr("172.20.48.31");
    servaddr.sin_port = htons(PORT);

    // Binding newly created socket to given IP and verification
    if((bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr))) != 0) {
      printf("Error: Socket bind failed\n");
      exit(0);

    } else {
      printf("Info: Socket bind successful\n");
    }

    // Now server is ready to listen and verification
    if((listen(sockfd, 5)) != 0) {
      printf("Error: Listen failed\n");
      exit(0);
    } else {
      printf("Info: Server listening\n");
    }

    len = sizeof(cli);
    connfd = accept(sockfd, (struct sockaddr *)&cli, &len);

    if(connfd < 0) {
      printf("Error: Server connection failed\n");
      exit(1);
    } else {
      printf("Info: Server connected to the client\n");
    }

    clock_gettime(CLOCK_REALTIME, &init_time);
    start_time = (double)(init_time.tv_sec * 1000) +
(double)(init_time.tv_nsec/1000000);
    pthread_mutex_lock(&data_fifo_lock);
    sprintf(data_fifo[data_wr_count], "~%s: %f %s\n", "INIT", start_time,
"0");
    data_wr_count++;
    pthread_mutex_unlock(&data_fifo_lock);

    /*-------------------------Creating Threads-------------------------*/
    //Launch resistive sensor data thread
    rc = pthread_create(&res_sensor_thread_tID, NULL, res_sensor,NULL);
    if(rc!= 0) {
    printf("Error: Failed to launch  resistive sensor thread\n");
    exit(-1);
    }

    //Launch uimodule thread
    rc = pthread_create(&uimodule_thread_tID, NULL, uimodule, &connfd);
    if(rc != 0) {
```

```
    printf("Error: Failed to launch command sorter pthread_create(): %d\n",
rc);
    exit(-1);
    }

    //Launch the CANOpen thread
     rc = pthread_create(&CANOpen_thread_tID, NULL, CANOpen_thread,NULL);
     if (rc != 0) {
     printf("Failed to launch CANOpen pthread_create(): %d\n", rc);
     exit(-1);
     }

    //Launch the data transfer threads
    rc = pthread_create(&data_transfer_thread_tID, NULL, data_transfer,
NULL);
    if(rc != 0) {
      printf("ERROR; return code from data transfer is %d\n", rc);
      exit(-1);
    }

    //Launch the data transfer threads
    rc = pthread_create(&PB_transfer_thread_tID, NULL, PB_transfer, NULL);
    if(rc != 0) {
      printf("ERROR; return code from data transfer is %d\n", rc);
      exit(-1);
    }

    //Launch the data transfer threads
    rc = pthread_create(&SW_transfer_thread_tID, NULL, SW_transfer, NULL);
    if(rc != 0) {
      printf("ERROR; return code from data transfer is %d\n", rc);
      exit(-1);
    }

    /*----------Check for threads to join and terminate safely----------*/


    rc = pthread_join(CANOpen_thread_tID, &status);
    if (rc != 0) {
    printf("ERROR: return code from CANOpen pthread_join(): %d\n", rc);
    exit(-1);
         }


    rc = pthread_join(res_sensor_thread_tID, &status);
    if(rc != 0) {
    printf("ERROR: return code from resistive pthread_join(): %d\n", rc);
    exit(-1);
    }

    rc = pthread_join(uimodule_thread_tID, &status);
    if(rc != 0) {
    printf("ERROR: return code from command sorter pthread_join(): %d\n",
rc);
    exit(-1);
    }
```

```
    rc = pthread_join(data_transfer_thread_tID, &status);
    if(rc != 0) {
      printf("ERROR: return code from cyclic pthread_join() is %d\n", rc);
      exit(-1);
    }

    rc = pthread_join(PB_transfer_thread_tID, &status);
    if(rc != 0) {
      printf("ERROR: return code from cyclic pthread_join() is %d\n", rc);
      exit(-1);
    }
    rc = pthread_join(SW_transfer_thread_tID, &status);
    if(rc != 0) {
      printf("ERROR: return code from cyclic pthread_join() is %d\n", rc);
      exit(-1);
    }

    /* Close TCP server socket */
    close(sockfd);
    printf("Info: TMNG program shutdown complete\n");
    return EXIT_SUCCESS;
}
/*---------------------------------------------------------------------------
-
Name        :  res_sensor.h
Authors     :  Nikhil Narayanan
Version     :
Copyright   :
Description :  Header file for res_sensor.c module
---------------------------------------------------------------------------
*/

#ifndef res_sensor_H_
#define res_sensor_H_

// IO data structure for board
typedef struct {
    unsigned int rnum;
    unsigned int rvalue;
}  MeasObj_struct;
// Data structure for temperature measurement
typedef struct {
    char sname[3];
    double tvalue;
    double tv_msec;
}  TempV_struct;

typedef struct {
    int s_count;
    int ts_sample;
    pthread_cond_t cv;
    pthread_mutex_t lock;
    char ts_unit[2];
}  SensParam_struct;
```

```
#define MEASOBJ_SIZE sizeof(MeasObj_struct)
#define REGNUM_ID 0x0FF
#define NREGS 4


extern unsigned int terminate;
extern int fd;
extern unsigned int data_wr_count;
extern char data_fifo[100][50];
extern pthread_mutex_t data_fifo_lock;
extern struct timespec init_time;
extern SensParam_struct PT100,CAN,PB,SW;
extern char mode[7];
extern pthread_mutex_t zynq_lock;


int reg_write(int reg,int value);
int reg_read(int reg);
int max_write(int reg,int value);
int max_read(int reg);

void* res_sensor();
void* PB_transfer();
void* SW_transfer();

#endif


/*---------------------------------------------------------------------------
-
Name        :  res_sensor.c
Authors     :  Nikhil Narayanan
Version     :
Copyright   :
Description :  Module for resistive sensor measurement
---------------------------------------------------------------------------
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/timerfd.h>
#include <time.h>
#include <pthread.h>
#include <sched.h>

#include "uimodule.h"
#include "res_sensor.h"

//Debug prints enables(Only added for function calls)
```

```
#define DBG_PRINT_SUB 0
#define MAX31685_DELAY 63 //In milliseconds
#define SPI_DELAY 100 //In micro seconds

//Function to write the slave registers in IP
int reg_write(int reg,int value)
{
    int retc;
    static MeasObj_struct TF_Obj_Snd;
    TF_Obj_Snd.rnum = reg;
    TF_Obj_Snd.rvalue = value;
    retc = write(fd,&TF_Obj_Snd, MEASOBJ_SIZE);
    if(retc < 0){
      perror("Error: Failed to write message to device.\n");
      return errno;
    }
    return 0;
}

//Function to read the slave registers in IP
int reg_read(int reg)
{
    int retc;
    static MeasObj_struct TF_Obj_Snd, TF_Obj_Rcv;
    TF_Obj_Snd.rnum = REGNUM_ID;
    TF_Obj_Snd.rvalue = reg;
    retc = write(fd,&TF_Obj_Snd, MEASOBJ_SIZE);
    if(retc < 0){
      perror("Error: Failed to write message to device.\n");
      return errno;
    }
    retc = read(fd,&TF_Obj_Rcv, MEASOBJ_SIZE);
    if(retc < 0){
      perror("Error: Failed to read message from device.\n");
      return errno;
    }
    return TF_Obj_Rcv.rvalue;
}

//Function to write the registers in MAX device

int max_write(int reg,int value)
{
    unsigned int spi_data,status;
    struct itimerspec itval;
    unsigned long long missed;
    int tim_fd;

    spi_data = (0x80 | reg)<<8 | (value & 0xff);
    pthread_mutex_lock(&zynq_lock);
    reg_write(1,spi_data);
    pthread_mutex_unlock(&zynq_lock);
    //Create the timer
    tim_fd = timerfd_create(CLOCK_MONOTONIC,0);
    if(tim_fd == -1){
      perror("Error: creating timer.");
```

```
    }
    //Make the timer To execute after SPI_DELAY
    itval.it_interval.tv_sec = 0;
    itval.it_interval.tv_nsec = 0;
    itval.it_value.tv_sec = 0;
    itval.it_value.tv_nsec = SPI_DELAY * 1000;
    timerfd_settime (tim_fd, 0, &itval, NULL);
    read(tim_fd, &missed, sizeof (missed));
    pthread_mutex_lock(&zynq_lock);
    status = (reg_read(1)) & 0x01;
    pthread_mutex_unlock(&zynq_lock);
    close(tim_fd);
    if(status == 0){
      perror("Error: SPI not completed");
    }
    return 0;
}

int max_read(int reg)
{
    unsigned int spi_data,status;
    struct itimerspec itval;
    unsigned long long missed;
    int tim_fd;

    spi_data = reg<<8;
    pthread_mutex_lock(&zynq_lock);
    reg_write(1,spi_data);
    pthread_mutex_unlock(&zynq_lock);
    tim_fd = timerfd_create(CLOCK_MONOTONIC,0);
    if(tim_fd == -1){
      perror("Error: creating timer.");
    }
    //Make the timer To execute after SPI_DELAY
    itval.it_interval.tv_sec = 0;
    itval.it_interval.tv_nsec = 0;
    itval.it_value.tv_sec = 0;
    itval.it_value.tv_nsec = SPI_DELAY * 1000;
    timerfd_settime (tim_fd, 0, &itval, NULL);
    read(tim_fd, &missed, sizeof (missed));
    pthread_mutex_lock(&zynq_lock);
    status = (reg_read(1)) & 0x01;
    pthread_mutex_unlock(&zynq_lock);
    close(tim_fd);
    if(status == 0){
      perror("SPI not completed");
      return 0xffff;
    }else{
      pthread_mutex_lock(&zynq_lock);
      status = reg_read(2) & 0xff;
      pthread_mutex_unlock(&zynq_lock);
      return(status);
    }
}

// Function to read the registers in MAX device and calculate the temperature
```

```
void* res_sensor()
{
  int i,r_lsb,r_msb,r_reg_bit0,ADC_code,status,tim_fd;
  struct timespec meas_time;
  struct itimerspec itval;
  unsigned long long missed;
  TempV_struct res_meas;
  struct sched_param schedparm_resist;

  memset(&schedparm_resist, 0, sizeof(schedparm_resist));
  schedparm_resist.sched_priority = 1; // lowest rt priority
  sched_setscheduler(0, SCHED_FIFO, &schedparm_resist);

  max_write(0,0x081); //Power up the MAX31685
  while(terminate == 0){
    pthread_mutex_lock(&PT100.lock);
    pthread_cond_wait(&PT100.cv, &PT100.lock);
    pthread_mutex_unlock(&PT100.lock);
    if(terminate == 1){
      break;
    }
    for(i=1;i <= PT100.s_count;i++){
      if(strcmp(mode,"auto") == 0){
         i--;
      }
      clock_gettime(CLOCK_REALTIME, &meas_time);
      max_write(0,0x0a1);//Do a one shot conversion in MAX31685
      //Create the timer
      tim_fd = timerfd_create(CLOCK_MONOTONIC,0);
      if(tim_fd == -1){
        perror("Error: creating timer.");
      }
      //Make the timer To execute after 63ms
      itval.it_interval.tv_sec = 0;
      itval.it_interval.tv_nsec = 0;

      if(strcmp(PT100.ts_unit,"s") == 0){
        itval.it_value.tv_sec = PT100.ts_sample;
        itval.it_value.tv_nsec = 0;
      }else if(strcmp(PT100.ts_unit,"ms") == 0){
        itval.it_value.tv_sec = 0;
        if(PT100.ts_sample < (MAX31685_DELAY)){
          itval.it_value.tv_nsec = (long int)(MAX31685_DELAY) * 1000000;
        }else{
          itval.it_value.tv_nsec = (long int)(PT100.ts_sample) * 1000000;
        }
      }
      timerfd_settime (tim_fd, 0, &itval, NULL);
      read(tim_fd, &missed, sizeof (missed));
      pthread_mutex_lock(&zynq_lock);
      status = (reg_read(1)) & 0x03;
      pthread_mutex_unlock(&zynq_lock);
      if(status != 1){
        perror("Error: Conversion is not finished");
      }
```

```
      r_msb = max_read(1);
      if(r_msb == 0xffff){
        perror("Error: Wait for MAX31865 timed out\n");
        continue;
      }
      r_msb &= 0xff;
      if(DBG_PRINT_SUB == 1){
        printf("Info: Value read from register 0x01 of MAX device (MSB) =
%x\n",r_msb);
      }
      r_lsb = max_read(2);
      if(r_msb == 0xffff){
        perror("Error: Wait for MAX31865 timed out\n");
        continue;
      }
      r_lsb &= 0xff;
      if(DBG_PRINT_SUB == 1){
        printf("Info: Value read from register 0x02 of MAX device (LSB) =
%x\n",r_lsb);
      }
      r_reg_bit0 = r_lsb & 0x01;
      if(r_reg_bit0 == 1){
        perror("Error: Faulty value read.\n");
      }else{
        ADC_code =  r_msb << 7 | r_lsb >> 1;
        pthread_mutex_lock(&data_fifo_lock);
        strcpy(res_meas.sname,"S3");
        //res_meas.tvalue = (ADC_code / 32.0) - 256.0;
        res_meas.tvalue = (ADC_code * 0.01220703125);
        res_meas.tv_msec = (double)(meas_time.tv_sec - init_time.tv_sec) *
1000000 + (double)(meas_time.tv_nsec - init_time.tv_nsec)/1000;
        sprintf(data_fifo[data_wr_count], "~%s: %.3f %.3f\n",
res_meas.sname,res_meas.tv_msec,res_meas.tvalue);
        data_wr_count++;
        pthread_mutex_unlock(&data_fifo_lock);
      }
      close(tim_fd);
    }

  }
  max_write(0, 0x001);
  printf("Info: MAX31865 Power down\n");
  printf("Info: Resistive Sensor thread dead\n");
  pthread_exit(NULL);
  return NULL;

}

void *PB_transfer (void *arg)
{
    int i,tim_PB_fd;
    struct itimerspec itval;
    unsigned long long missed;
    unsigned int PB_value ;
    struct timespec meas_time;
    TempV_struct PB_meas;
```

```
    struct sched_param schedparm_PB;

    memset(&schedparm_PB, 0, sizeof(schedparm_PB));
    schedparm_PB.sched_priority = 1; // lowest rt priority
    sched_setscheduler(0, SCHED_FIFO, &schedparm_PB);

    while (terminate == 0)
    {
      pthread_mutex_lock(&PB.lock);
      pthread_cond_wait(&PB.cv, &PB.lock);
      pthread_mutex_unlock(&PB.lock);
      if(terminate == 1){
        break;
      }
      /* Create the timer */
      tim_PB_fd = timerfd_create (CLOCK_MONOTONIC, 0);
      if(tim_PB_fd == -1){
        perror("Error: creating timer.");
      }
      /* Make the timer periodic */
      if(strcmp(PB.ts_unit,"s") == 0){
        itval.it_interval.tv_sec = PB.ts_sample;
        itval.it_interval.tv_nsec = 0;
      }else if(strcmp(PB.ts_unit,"ms") == 0){
        itval.it_interval.tv_sec = 0;
        itval.it_interval.tv_nsec = (long int)PB.ts_sample * 1000000;
      }else if(strcmp(PB.ts_unit,"us") == 0){
        itval.it_interval.tv_sec = 0;
        itval.it_interval.tv_nsec = (long int)PB.ts_sample * 1000;
      }
      itval.it_value.tv_sec = 1;
      itval.it_value.tv_nsec = 0;
      timerfd_settime (tim_PB_fd, 0, &itval, NULL);

      for(i=1;i<=PB.s_count;i++){
        if(strcmp(mode,"auto") == 0){
           i--;
        }
        read(tim_PB_fd, &missed, sizeof (missed));
        clock_gettime(CLOCK_REALTIME, &meas_time);
        pthread_mutex_lock(&zynq_lock);
        PB_value = (reg_read(0) & 0x1f00) >> 8;
        pthread_mutex_unlock(&zynq_lock);
        //Lock the data fifo here
        pthread_mutex_lock(&data_fifo_lock);
        strcpy(PB_meas.sname,"S1");
        PB_meas.tvalue = PB_value;
        PB_meas.tv_msec = (double)(meas_time.tv_sec - init_time.tv_sec) *
1000000 + (double)(meas_time.tv_nsec - init_time.tv_nsec)/1000;
        sprintf(data_fifo[data_wr_count], "~%s: %.3f %.3f\n", PB_meas.sname,
PB_meas.tv_msec,PB_meas.tvalue);
        data_wr_count++;
        //Unlock the data fifo here
        pthread_mutex_unlock(&data_fifo_lock);
      }
      close(tim_PB_fd);
```

```
    }
    printf("Info: Push button thread dead.\n");
    pthread_exit(NULL);
    return NULL;
}

void *SW_transfer (void *arg)
{
    int i,tim_SW_fd;
    struct itimerspec itval;
    unsigned long long missed;
    unsigned int SW_value ;
    struct timespec meas_time;
    TempV_struct SW_meas;
    struct sched_param schedparm_SW;

    memset(&schedparm_SW, 0, sizeof(schedparm_SW));
    schedparm_SW.sched_priority = 1; // lowest rt priority
    sched_setscheduler(0, SCHED_FIFO, &schedparm_SW);

    while (terminate == 0)
    {
      pthread_mutex_lock(&SW.lock);
      pthread_cond_wait(&SW.cv, &SW.lock);
      pthread_mutex_unlock(&SW.lock);
      if(terminate == 1){
        break;
      }
      /* Create the timer */
      tim_SW_fd = timerfd_create (CLOCK_MONOTONIC, 0);
      if(tim_SW_fd == -1){
        perror("Error: Creating timer failed.");
      }
      /* Make the timer periodic */
      if(strcmp(SW.ts_unit,"s") == 0){
        itval.it_interval.tv_sec = SW.ts_sample;
        itval.it_interval.tv_nsec = 0;
      }else if(strcmp(SW.ts_unit,"ms") == 0){
        itval.it_interval.tv_sec = 0;
        itval.it_interval.tv_nsec = (long int)SW.ts_sample * 1000000;
      }else if(strcmp(SW.ts_unit,"us") == 0){
        itval.it_interval.tv_sec = 0;
        itval.it_interval.tv_nsec = (long int)SW.ts_sample * 1000;
      }
      itval.it_value.tv_sec = 1;
      itval.it_value.tv_nsec = 0;
      timerfd_settime (tim_SW_fd, 0, &itval, NULL);

      for(i = 1;i <= SW.s_count;i++){
        if(strcmp(mode,"auto") == 0){
           i--;
        }
        read(tim_SW_fd, &missed, sizeof (missed));
        clock_gettime(CLOCK_REALTIME, &meas_time);
        pthread_mutex_lock(&zynq_lock);
        SW_value = reg_read(0) & 0xff;
```

```
        pthread_mutex_unlock(&zynq_lock);
        //Lock the data fifo here
        pthread_mutex_lock(&data_fifo_lock);
        strcpy(SW_meas.sname,"S0");
        SW_meas.tvalue = SW_value;
        SW_meas.tv_msec = (double)(meas_time.tv_sec - init_time.tv_sec) *
1000000 + (double)(meas_time.tv_nsec - init_time.tv_nsec)/1000;
        sprintf(data_fifo[data_wr_count], "~%s: %.3f %.3f\n",
SW_meas.sname,SW_meas.tv_msec,SW_meas.tvalue);
        data_wr_count++;
        //Unlock the data fifo here
        pthread_mutex_unlock(&data_fifo_lock);
      }
      close(tim_SW_fd);
    }
    printf("Info: Switches thread dead.\n");
    pthread_exit(NULL);
    return NULL;
}

/*---------------------------------------------------------------------------
-
Name        :  uimodule.h
Authors     :  Khizar Akhtar,Nikhil Narayanan
Version     :
Copyright   :
Description :  header file for uimodule.c
---------------------------------------------------------------------------
*/


#ifndef uimodule_H_
#define uimodule_H_

extern int connfd;
extern char mode[7];
void* uimodule();

#endif
/*---------------------------------------------------------------------------
-
Name        :  uimodule.c
Authors     :  Khizar Akhtar,Nikhil Narayanan
Version     :
Copyright   :
Description :  User interface thread
---------------------------------------------------------------------------
*/

#include <stddef.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
```

```
#include "res_sensor.h"
#include "uimodule.h"
#include "can_sensor.h"

#define BUFFER_SIZE 80

void* uimodule()

{
    char buff[BUFFER_SIZE],text_inp[BUFFER_SIZE],arg3[2];
    int arg1,arg2;
    struct sched_param schedparm_ui;

    memset(&schedparm_ui, 0, sizeof(schedparm_ui));
    schedparm_ui.sched_priority = 1; // lowest rt priority
    sched_setscheduler(0, SCHED_FIFO, &schedparm_ui);

    while (terminate == 0)
    {
      strcpy(text_inp,"\0");
      arg1 = 0;
      arg2 = 0;
      strcpy(arg3,"\0");

      terminate = 0;
      bzero(buff, BUFFER_SIZE);
      read(connfd, buff, sizeof(buff));
      //printf("\n %s",buff);
      sscanf(buff, "%s %d %d %s", text_inp, &arg1, &arg2, arg3);
      if(strcmp(text_inp,"exit") == 0){
        terminate = 1;
        lastRun = 1;
        pthread_cond_signal(&PT100.cv);
        pthread_cond_signal(&CAN.cv);
        pthread_cond_signal(&PB.cv);
        pthread_cond_signal(&SW.cv);
        break;
      }else if(strcmp(text_inp,"help") == 0){
        printf("--- Please sent one of the following options ---\n");
        printf("--- Sensor ID - iterations - sampling period  ---\n");
        printf("--- To exit - exit ---\n");
      }
      else if(strcmp(text_inp,"auto")==0){
        printf("Auto mode enabled.\n");
        strcpy(mode,"auto");
        PT100.s_count = 1;
        CAN.s_count = 1;
        PB.s_count = 1;
        SW.s_count = 1;
        pthread_cond_signal(&PT100.cv);
        pthread_cond_signal(&CAN.cv);
        pthread_cond_signal(&SW.cv);
        pthread_cond_signal(&PB.cv);
      }
      else if(strcmp(text_inp,"manual")==0){
        printf("Auto mode stopped.\n");
```

```
        strcpy(mode,"manual");
        PT100.s_count = 0;
        CAN.s_count = 0;
        PB.s_count = 0;
        SW.s_count = 0;
      }
      else if(strcmp(text_inp,"S3") == 0){
        if(sscanf(buff, "%s %d %d %s", text_inp, &arg1, &arg2, arg3) != 4){
            printf("Error: Please enter a valid count for iterations or
sampling period and unit and try again.\n");
          printf("Info: Values %s %d %d %s\n" ,text_inp, arg1,arg2,arg3 );
        }else if((arg1 == 0)||(arg2 == 0)){
          printf("Error: Iterations and sampling rate should be non
zero.\n");
        }else{
          PT100.s_count = arg1;
          PT100.ts_sample = arg2;
          strcpy(PT100.ts_unit,arg3);
          pthread_cond_signal(&PT100.cv);
          printf("Info: S3 Sample Count: %d Sampling Rate:
%d\n",PT100.s_count,PT100.ts_sample);
        }
      }else if(strcmp(text_inp,"S2") == 0){
        if(sscanf(buff, "%s %d %d %s", text_inp, &arg1, &arg2, arg3) != 4){
            printf("Error: Please enter a valid count for iterations or
sampling period and unit and try again.\n");
          printf("Info: Values %s %d %d %s\n" ,text_inp, arg1,arg2,arg3 );

        }else if((arg1 == 0)||(arg2 == 0)){
            printf("Error: Iterations and sampling rate should be non
zero.\n");
        }else{
          CAN.s_count = arg1;
          CAN.ts_sample = arg2;
          strcpy(CAN.ts_unit,arg3);
          pthread_cond_signal(&CAN.cv);
          printf("Info: S2 Sample Count: %d Sampling Rate: %d\n"
,CAN.s_count, CAN.ts_sample );
        }
      }else if(strcmp(text_inp,"S1") == 0){
        if(sscanf(buff, "%s %d %d %s", text_inp, &arg1, &arg2, arg3) != 4){
            printf("Error: Please enter a valid count for iterations or
sampling period and unit and try again.\n");
          printf("Info: Values %s %d %d %s\n" ,text_inp, arg1,arg2,arg3 );

        }else if((arg1 == 0)||(arg2 == 0)){
            printf("Error: Iterations and sampling rate should be non
zero.\n");
        }else{
          PB.s_count = arg1;
          PB.ts_sample = arg2;
          strcpy(PB.ts_unit,arg3);
          pthread_cond_signal(&PB.cv);
          printf("Info: S1 Sample Count: %d Sampling Rate: %d \n"
,PB.s_count, PB.ts_sample);
        }
```

```
      }else if(strcmp(text_inp,"S0") == 0){
        if(sscanf(buff, "%s %d %d %s", text_inp, &arg1, &arg2, arg3) != 4){
          printf("Error: Please enter a valid count for iterations or
sampling period and unit and try again.\n");
          printf("Info: Values %s %d %d %s\n" ,text_inp, arg1,arg2,arg3 );

        }else if((arg1 == 0)||(arg2 == 0)){
            printf("Error: Iterations and sampling rate should be non
zero.\n");
        }else{
          SW.s_count = arg1;
          SW.ts_sample = arg2;
          strcpy(SW.ts_unit,arg3);
          pthread_cond_signal(&SW.cv);
          printf("Info: S0 Sample Count: %d Sampling Rate: %d\n" ,SW.s_count,
SW.ts_sample );
        }
      }else{
        printf("Error: Invalid command.\n");
      }
    }
    printf("Info: UImodule thread dead\n");
    pthread_exit(NULL);
    return NULL;
}
/*---------------------------------------------------------------------------
-
Name        :  data_transfer.h
Authors     :  Nikhil Narayanan
Version     :
Copyright   :
Description :  Header file for data_transfer.c module
---------------------------------------------------------------------------
*/

#ifndef data_transfer_H_
#define data_transfer_H_

extern int connfd;
void* data_transfer();

#endif
/*---------------------------------------------------------------------------
-
Name        :  data_transfer.c
Authors     :  Nikhil Narayanan
Version     :
Copyright   :
Description :  Cyclic thread for ethernet transmission, PB and SW
---------------------------------------------------------------------------
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
```

```c
#include <pthread.h>
#include <stdint.h>
#include <unistd.h>
#include <sys/timerfd.h>
#include <time.h>

#include "res_sensor.h"
#include "data_transfer.h"


#define BUFFER_SIZE 80
#define DATA_INTERVAL 1 //In milliseconds

void *data_transfer (void *arg)
{
    int rd_count;
    int tim_fd;
    struct itimerspec itval;
    unsigned long long missed;
    struct sched_param schedparm_data;

    memset(&schedparm_data, 0, sizeof(schedparm_data));
    schedparm_data.sched_priority = 1; // lowest rt priority
    sched_setscheduler(0, SCHED_FIFO, &schedparm_data);

    /* Create the timer */
    tim_fd = timerfd_create (CLOCK_MONOTONIC, 0);
    if(tim_fd == -1){
      perror("Error: creating timer.");
    }
    /* Make the timer periodic */
    itval.it_interval.tv_sec = 0;
    itval.it_interval.tv_nsec = DATA_INTERVAL * 1000000;
    itval.it_value.tv_sec = 1;
    itval.it_value.tv_nsec = 0;
    timerfd_settime (tim_fd, 0, &itval, NULL);

    while (terminate == 0)
    {
      read (tim_fd, &missed, sizeof (missed));
      if(data_wr_count > 0){
        //Lock the data fifo here
        pthread_mutex_lock(&data_fifo_lock);
        for(rd_count=0;rd_count < data_wr_count;rd_count++){
          write(connfd, data_fifo[rd_count], strlen(data_fifo[rd_count]));
        }
        data_wr_count = 0;
        //Unlock the data fifo here
        pthread_mutex_unlock(&data_fifo_lock);
      }
    }
    close(tim_fd);
    printf("Info: Data transfer thread dead.\n");
    pthread_exit(NULL);
    return NULL;
}
```

```
/*
 * measdev.h
 *
 *  Created on: Oct 08, 2020
 *      Author: mueller
 */

#ifndef MEASDEV_H_
#define MEASDEV_H_

// MEAScdd IO data
typedef struct {
    unsigned int rnum;
    unsigned int rvalue;
}  MeasObj_struct;

typedef struct {
    float tvalue;
    int tv_sec;
    int tv_nsec;
}  TempV_struct;

#define MEASOBJ_SIZE sizeof(MeasObj_struct)
#define REGNUM_ID 0x0FF
#define NREGS 4

#define TIMER100ms  0x009896ff
#define MBINT_ENABLE 0x80000000
#define MBINT_ACKN 0x40000000

#endif /* MEASDEV_H_ */

/*
============================================================================
=
Name : cansensor.h
Authors : Muhammad Hussain, Maddipatla Srujana
Version :
Description :

============================================================================
*/

#ifndef can_sensor_H_
#define can_sensor_H_


extern unsigned int terminate;
extern unsigned int sensID;
extern unsigned int lastRun;
void* CANOpen_thread();

#endif
```

```
/*
============================================================================
=
Name : cansensor.c
Authors : Muhammad Hussain, Maddipatla Srujana
Version :
Description :

============================================================================
*/
#include <sched.h>
#include <errno.h>
#include <stdarg.h>
#include <syslog.h>
#include <sys/epoll.h>
#include <net/if.h>
#include <linux/reboot.h>
#include <sys/reboot.h>


#include "CANopen.h"
#include "OD.h"
#include "301/CO_driver.h"
#include "309/CO_gateway_ascii.h"
#include "CO_error.h"
#include "CO_epoll_interface.h"
#include "CO_storageLinux.h"

#include "can_sensor.h"
#include "res_sensor.h"
#include "uimodule.h"

/* Interval of main-line and real-time thread in microseconds */
#ifndef MAIN_THREAD_INTERVAL_US
#define MAIN_THREAD_INTERVAL_US 100000
#endif

#ifndef TMR_THREAD_INTERVAL_US
#define TMR_THREAD_INTERVAL_US 1000
#endif

/* default values for CO_CANopenInit() */
#ifndef NMT_CONTROL
#define NMT_CONTROL \
            CO_NMT_STARTUP_TO_OPERATIONAL \
          | CO_NMT_ERR_ON_ERR_REG \
          | CO_ERR_REG_GENERIC_ERR \
          | CO_ERR_REG_COMMUNICATION
#endif

#ifndef FIRST_HB_TIME
#define FIRST_HB_TIME 500
#endif

#ifndef SDO_SRV_TIMEOUT_TIME
#define SDO_SRV_TIMEOUT_TIME 1000
```

```
#endif

#ifndef SDO_CLI_TIMEOUT_TIME
#define SDO_CLI_TIMEOUT_TIME 500
#endif

#ifndef SDO_CLI_BLOCK
#define SDO_CLI_BLOCK false
#endif

#ifndef OD_STATUS_BITS
#define OD_STATUS_BITS NULL
#endif

/* CANopen gateway enable switch for CO_epoll_processMain() */
#ifndef GATEWAY_ENABLE
#define GATEWAY_ENABLE true
#endif
/* Interval for time stamp message in milliseconds */
#ifndef TIME_STAMP_INTERVAL_MS
#define TIME_STAMP_INTERVAL_MS 10000
#endif

/* Definitions for application specific data storage objects */
#ifndef CO_STORAGE_APPLICATION
#define CO_STORAGE_APPLICATION
#endif
/* Interval for automatic data storage in microseconds */
#ifndef CO_STORAGE_AUTO_INTERVAL
#define CO_STORAGE_AUTO_INTERVAL 60000000
#endif

/* CANopen object */
CO_t *CO = NULL;
uint16_t pendingBitRate = 500;
// Pending CANopen NodeId, can be set by argument or LSS slave.
uint8_t pendingNodeId = 1;
static uint8_t CO_activeNodeId;

/*real-time thread handler*/
static void* rt_thread(void* arg);

/*Real-time thread ID*/
pthread_t rt_thread_id;

/*Separate EPOLL objects for main-line, real-time and gateway functions*/
CO_epoll_t epMain, epRT;
CO_epoll_gtw_t epGtw;
CO_ReturnError_t err;
CO_NMT_reset_cmd_t reset = CO_RESET_NOT;


const char *CANbus = "can0";
long int CANopen_timestamp;
unsigned int CANopen_RPDO_val;
unsigned int end_program = 0;
```

```
/*Enabling interaction capability through console/terminal*/
int32_t commandInterface = CO_COMMAND_IF_STDIO;
//int32_t commandInterface = CO_COMMAND_IF_DISABLED;

//CANOpen Object Pointer to be referenced for every protocol and CAN bus
CO_CANptrSocketCan_t CANptr = {0};
/*Disabling device control over socket*/
char *localSocketPath = NULL;
uint32_t socketTimeout_ms = 0;

/*Flags to control the flow of the application*/
unsigned int PDOFlag = 0;
unsigned lastRun = 0;
/*---------------------------------------*/
/* Message logging function */
void log_printf(int priority, const char *format, ...) {
 va_list ap;
 va_start(ap, format);
 vsyslog(priority, format, ap);
 va_end(ap);
#if (CO_CONFIG_GTW) & CO_CONFIG_GTW_ASCII_LOG
    if (CO != NULL) {
        char buf[200];
        time_t timer;
        struct tm* tm_info;
        size_t len;

        timer = time(NULL);
        tm_info = localtime(&timer);
        len = strftime(buf, sizeof(buf), "%Y-%m-%d %H:%M:%S: ", tm_info);

        va_start(ap, format);
        vsnprintf(buf + len, sizeof(buf) - len - 2, format, ap);
        va_end(ap);
        strcat(buf, "\r\n");
        CO_GTWA_log_print(CO->gtwa, buf);
    }
#endif
}

#if (CO_CONFIG_EM) & CO_CONFIG_EM_CONSUMER
/* Callback for Emergency Messages */
static void EmergencyRxCallback(const uint16_t ident,
 const uint16_t errorCode,
 const uint8_t errorRegister,
 const uint8_t errorBit,
 const uint32_t infoCode)
{
 int16_t nodeIdRx = ident ? (ident&0x7F) : CO_activeNodeId;
 log_printf(LOG_NOTICE, DBG_EMERGENCY_RX, nodeIdRx, errorCode,
 errorRegister, errorBit, infoCode);
}
#endif

#if ((CO_CONFIG_NMT) & CO_CONFIG_NMT_CALLBACK_CHANGE) \
```

```
|| ((CO_CONFIG_HB_CONS) & CO_CONFIG_HB_CONS_CALLBACK_CHANGE)
/* Return string description of NMT state. */
static char *NmtState2Str(CO_NMT_internalState_t state)
{
 switch(state) {
 case CO_NMT_INITIALIZING: return "initializing";
 case CO_NMT_PRE_OPERATIONAL: return "pre-operational";
 case CO_NMT_OPERATIONAL: return "operational";
 case CO_NMT_STOPPED: return "stopped";
 default: return "unknown";
 }
}
#endif

#if (CO_CONFIG_NMT) & CO_CONFIG_NMT_CALLBACK_CHANGE
/* Callback for NMT change messages */
static void NmtChangedCallback(CO_NMT_internalState_t state)
{
 log_printf(LOG_NOTICE, DBG_NMT_CHANGE, NmtState2Str(state), state);
}
#endif


#if (CO_CONFIG_HB_CONS) & CO_CONFIG_HB_CONS_CALLBACK_CHANGE
/* Callback for monitoring Heart-beat remote NMT state change */
static void HeartbeatNmtChangedCallback(uint8_t nodeId, uint8_t idx,
 CO_NMT_internalState_t state,
void *object)
{
 (void)object;
 log_printf(LOG_NOTICE, DBG_HB_CONS_NMT_CHANGE,
 nodeId, idx, NmtState2Str(state), state);
}

#endif


/******************************************************************************
*****
* Main-line thread
******************************************************************************
***/
void* CANOpen_thread() {

 bool_t firstRun = true;
 reset = CO_RESET_NOT;

 /* To extract CAN bus index from linux system */
CANptr.can_ifindex = if_nametoindex(CANbus);
    if (CANptr.can_ifindex == 0) {
    printf("CAN Interface %s inactive in linux system\n\r", CANbus);
    reset = CO_RESET_QUIT;
        }

 /* Allocate memory for CANopen objects */
 uint32_t heapMemoryUsed = 0;
```

```
 CO_config_t *config_ptr = NULL;

 CO = CO_new(config_ptr, &heapMemoryUsed);
 //CO = CO_new(NULL,0);
  if (CO == NULL) { //if(CO == !CO_ERROR_NO){
        printf("Can memory allocation failed.\n\r");
        reset = CO_RESET_QUIT;
  }

 /* Epoll function creation */
 err = CO_epoll_create(&epMain, MAIN_THREAD_INTERVAL_US);
 if(err != CO_ERROR_NO) {
 printf("Error: CO_epoll_create(main) failed\n\r");
 reset = CO_RESET_QUIT;
     }

 err = CO_epoll_create(&epRT, TMR_THREAD_INTERVAL_US);
 if(err != CO_ERROR_NO) {
 printf("Error: CO_epoll_create(RT) failed\n\r");
 reset = CO_RESET_QUIT;
     }
 // Assigning real-time epoll file descriptor as the default file
CANptr.epoll_fd = epRT.epoll_fd;

#if (CO_CONFIG_GTW) & CO_CONFIG_GTW_ASCII

    err = CO_epoll_createGtw(&epGtw, epMain.epoll_fd, commandInterface,
                             socketTimeout_ms, localSocketPath);
    if(err != CO_ERROR_NO) {
         printf("Error: CO_epoll_createGtw failed\n\r");
         reset = CO_RESET_QUIT;
    }

#endif

 /* Initialization loop
 * First run and during CANopen Device Communication Reset
 * check for any failure in Initialization*/
 while(reset != CO_RESET_APP && reset != CO_RESET_QUIT && end_program == 0) {
 uint32_t errInfo;
 reset = CO_RESET_NOT;


/* Wait rt_thread. */
 if(!firstRun) {
     CO_LOCK_OD(CO->CANmodule);
     CO->CANmodule->CANnormal = false;
     CO_UNLOCK_OD(CO->CANmodule);
 }


/* Enter CAN configuration. */
    CO_CANsetConfigurationMode((void *)&CANptr);
    CO_CANmodule_disable(CO->CANmodule);
```

```
 /* Initialize CAN Module, LSS and CANOpen_Node */
 err = CO_CANinit(CO, (void *)&CANptr, 0 /* bit rate not used */);
 if (err != CO_ERROR_NO) {
 printf("Error: CAN Module initialization failed\n\r");
 reset = CO_RESET_QUIT;

}
 CO_LSS_address_t lssAddress = {.identity = {
             .vendorID = OD_PERSIST_COMM.x1018_identity.vendor_ID,
             .productCode = OD_PERSIST_COMM.x1018_identity.productCode,
             .revisionNumber = OD_PERSIST_COMM.x1018_identity.revisionNumber,
             .serialNumber = OD_PERSIST_COMM.x1018_identity.serialNumber
         }};


 err = CO_LSSinit(CO, &lssAddress,
                  &pendingNodeId, &pendingBitRate);
 if(err != CO_ERROR_NO) {
     log_printf(LOG_CRIT, DBG_CAN_OPEN, "CO_LSSinit()", err);
     printf("LSS Initialization failed.\n\r");
     reset = CO_RESET_QUIT;
 }

 CO_activeNodeId = pendingNodeId;
 errInfo = 0;

 err = CO_CANopenInit(CO,                // CANopen object
                      NULL,              // alternate NMT
                      NULL,              // alternate em
                      OD,                // Object dictionary
                      OD_STATUS_BITS,    // Optional OD_statusBits
                      NMT_CONTROL,       // CO_NMT_control_t
                      FIRST_HB_TIME,     // firstHBTime_ms
                      SDO_SRV_TIMEOUT_TIME, // SDOserverTimeoutTime_ms
                      SDO_CLI_TIMEOUT_TIME, // SDOclientTimeoutTime_ms
                      SDO_CLI_BLOCK,     // SDOclientBlockTransfer
                      CO_activeNodeId,
                      &errInfo);
 if(err != CO_ERROR_NO && err != CO_ERROR_NODE_ID_UNCONFIGURED_LSS) {
     if (err == CO_ERROR_OD_PARAMETERS) {
         log_printf(LOG_CRIT, DBG_OD_ENTRY, errInfo);
     }
     else {
         log_printf(LOG_CRIT, DBG_CAN_OPEN, "CO_CANopenInit()", err);
     }
     reset = CO_RESET_QUIT;
 }


 /* initialize part of threadMain and callbacks */
 CO_epoll_initCANopenMain(&epMain, CO);
// ------------------------------------------------
#if (CO_CONFIG_GTW) & CO_CONFIG_GTW_ASCII
        CO_epoll_initCANopenGtw(&epGtw, CO);
#endif
```

```
if(!CO->nodeIdUnconfigured) {
    if(errInfo != 0) {
        CO_errorReport(CO->em, CO_EM_INCONSISTENT_OBJECT_DICT,
                       CO_EMC_DATA_SET, errInfo);
    }

#if (CO_CONFIG_EM) & CO_CONFIG_EM_CONSUMER
            CO_EM_initCallbackRx(CO->em, EmergencyRxCallback);
#endif
#if (CO_CONFIG_NMT) & CO_CONFIG_NMT_CALLBACK_CHANGE
            CO_NMT_initCallbackChanged(CO->NMT, NmtChangedCallback);
#endif
#if (CO_CONFIG_HB_CONS) & CO_CONFIG_HB_CONS_CALLBACK_CHANGE
          CO_HBconsumer_initCallbackNmtChanged(CO->HBcons, 0, NULL,
                                               HeartbeatNmtChangedCallback);

#endif

        log_printf(LOG_INFO, DBG_CAN_OPEN_INFO, CO_activeNodeId,
"communication reset");
        }

        else {

            log_printf(LOG_INFO, DBG_CAN_OPEN_INFO, CO_activeNodeId, "node-id
not initialized");
            printf("CANopen device node-id: %d Initialization
failed.\n\r",CO_activeNodeId);
            reset = CO_RESET_QUIT;
        }

        /* First time only initialization. */
        if(firstRun) {
            firstRun = false;

            if(pthread_create(&rt_thread_id, NULL, rt_thread, NULL) != 0) {
                                    log_printf(LOG_CRIT, DBG_ERRNO,
"pthread_create(rt_thread)");
                                    printf("Real time thread creation
failed.\n\r");
                                    reset = CO_RESET_QUIT;
                                }

        } /* if(firstRun) */

                errInfo = 0;
                err = CO_CANopenInitPDO(CO,             /* CANopen object */
                                        CO->em,         /* emergency object
*/
                                        OD,             /* Object dictionary
*/
                                        CO_activeNodeId,
                                        &errInfo);

                if(err != CO_ERROR_NO ) {
```

```
                    printf("PDO initialization failed.\n\r");

                    reset = CO_RESET_QUIT;
                }

                /* start CAN */
                CO_CANsetNormalMode(CO->CANmodule);
                reset = CO_RESET_NOT;
                printf("Info: CAN is running \n\r");


 while(reset == CO_RESET_NOT && end_program == 0) {
 /* loop for normal program execution
******************************************/

    if(lastRun == 1)
    {
        end_program = 1;
        break;
    }

     CO_epoll_wait(&epMain);
     CO_epoll_processGtw(&epGtw, CO, &epMain);
     CO_epoll_processMain(&epMain, CO, GATEWAY_ENABLE, &reset);
     CO_epoll_processLast(&epMain);
 }

 }

        /* program exit , join threads */


    if (pthread_join(rt_thread_id, NULL) != 0) {
        printf("RT thread joining failed.\n\r");
    }

        /* delete epoll objects */
        CO_epoll_close(&epRT);
        CO_epoll_close(&epMain);
        CO_epoll_closeGtw(&epGtw);
        /* delete CAN object memory */
        CO_CANsetConfigurationMode((void *)&CANptr);
        //CO_delete((void *)&CANptr);
        CO_delete(CO);
        printf("Info: CANopen thread dead\n\r");
    return NULL;
}


/*****************************************************************************
***
 * Real time thread for CAN receive and threadTmr

*****************************************************************************
*/
static void* rt_thread(void* arg) {
```

```
    (void)arg;
    while(reset == CO_RESET_NOT && end_program == 0) {

        CO_epoll_wait(&epRT);
        CO_epoll_processRT(&epRT, CO, true);
        CO_epoll_processLast(&epRT);

        if(CO->RPDO->CANrxNew[0]) {
        /* Set flag every time new PDO is received at the buffer */
            printf("PDO flag is set \n");

        CANopen_timestamp =
        (CO->CANrx->timestamp.tv_nsec/1000) +
        (CO->CANrx->timestamp.tv_sec*1000000);
        //(CO->CANmodule->rxArray[4].timestamp.tv_nsec/1000) +
        //(CO->CANmodule->rxArray[4].timestamp.tv_sec*1000000);

        CANopen_RPDO_val = (int)((CO->RPDO->CANrxData[0][1] << 8) | CO->RPDO-
>CANrxData[0][0]);
        PDOFlag = 1;
       }
    }

    return NULL;
}
```

```
   ============================================================================


/*

============================================================================
==
 Name        : TcpClient.java
 Authors     : Asher Ali
 Version     : 3
 Copyright   : Asher Ali , 03-March-2022
 Description : Measurement Network Gateway main GUI application

============================================================================
==
 */
package ngui;

import java.awt.Color;
import java.awt.EventQueue;

import javax.swing.JFrame;
import javax.swing.JMenuBar;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import javax.swing.SpringLayout;
import javax.swing.Timer;
import javax.swing.text.BadLocationException;
import javax.swing.text.Document;
```

```
import javax.swing.text.SimpleAttributeSet;
import javax.swing.text.StyleConstants;
import javax.swing.JTextField;
import javax.swing.JScrollPane;
import javax.swing.JTextPane;
import javax.swing.JPanel;

import org.knowm.xchart.XChartPanel;
import org.knowm.xchart.XYChart;
import org.knowm.xchart.XYChartBuilder;

import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.FlowLayout;
import javax.swing.BoxLayout;
import java.awt.BorderLayout;
import java.awt.Panel;


public class MNGhmi {

    private JFrame frame;
    private JTextField textField_cmd;

    /* Launch the application*/
    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                try {
                    MNGhmi window = new MNGhmi();
                    window.frame.setVisible(true);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });


    }

    /*Create the application*/
    public MNGhmi() {
        initialize();
    }

    /*Initialize the contents of the frame*/
    private void initialize() {
        frame = new JFrame();
        //frame.setBounds(100, 100, 1310, 822);
        frame.setBounds(100, 100, 1357, 828);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JMenuBar menuBar = new JMenuBar();
        frame.setJMenuBar(menuBar);
```

```
        JMenu mnFile = new JMenu("File");
        menuBar.add(mnFile);

        JMenuItem mntmExit = new JMenuItem("Exit");
        mntmExit.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });
        mnFile.add(mntmExit);


            InitCharts();

        /*Add the sensor data and the time-stamp on the respective graphs*/
        SW_Chart.addSeries("S0", Stime[0], Sdata[0]);
        PB_Chart.addSeries("S1", Stime[1], Sdata[1]);
        CAN_Chart.addSeries("S2", Stime[2], Sdata[2]);
        RTD_Chart.addSeries("S3", Stime[3], Sdata[3]);

        /*Executes when command is entered on the GUI text pane */
        textField_cmd = new JTextField();
        textField_cmd.setFont(new Font("Monospaced", Font.PLAIN, 12));
        textField_cmd.setBounds(10, 740, 1322, 19);
        textField_cmd.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                CommandHandler(textField_cmd.getText());
                textField_cmd.setText("");
            }
        });
        frame.getContentPane().setLayout(null);
        frame.getContentPane().setLayout(null);
        frame.getContentPane().setLayout(null);
        frame.getContentPane().add(textField_cmd);
        textField_cmd.setColumns(10);

        JScrollPane scrollPane = new JScrollPane();
        scrollPane.setFont(new Font("Monospaced", Font.PLAIN, 12));;
        scrollPane.setBounds(10, 0, 234, 738);
        frame.getContentPane().add(scrollPane);

        textPane_log = new JTextPane();
        textPane_log.setFont(new Font("Monospaced", Font.PLAIN, 12));
        textPane_log.setText("MNGhmi Initialized! \n");
        PrintTxtWin("\nS0: Sensor 0: Switches\nS1: Sensor 1: PushButtons\n"
                + "S2: Sensor 2: CAN-Temperature\n"+ "S3: Sensor 3: RTD-
Resistance\n",2,true);
        PrintTxtWin("Type help for command formats",1,true);
        textPane_log.setEditable(false);
        scrollPane.setViewportView(textPane_log);

        // Panels creation to be shown on GUI

        panel_canv = new CanvasW();
        panel_canv.setBounds(254, 10, 1078, 137);
```

```
        frame.getContentPane().add(panel_canv);
        panel_canv.setLayout(new SpringLayout());

        panel_SW = new XChartPanel<XYChart>(SW_Chart);
        panel_SW.setBounds(254, 148, 1078, 145);
        frame.getContentPane().add(panel_SW);
        panel_SW.setLayout(new SpringLayout());

        panel_PB = new XChartPanel<XYChart>(PB_Chart);
        panel_PB.setBounds(254, 294, 1078, 143);
        frame.getContentPane().add(panel_PB);
        panel_PB.setLayout(new SpringLayout());

        panel_CAN = new XChartPanel<XYChart>(CAN_Chart);
        panel_CAN.setBounds(254, 439, 1078, 157);
        frame.getContentPane().add(panel_CAN);
        panel_CAN.setLayout(new SpringLayout());

        panel_RTD = new XChartPanel<XYChart>(RTD_Chart);
        panel_RTD.setFont(new Font("Monospaced", Font.PLAIN, 12));
        panel_RTD.setBounds(254, 601, 1078, 137);
        frame.getContentPane().add(panel_RTD);
        panel_RTD.setLayout(new SpringLayout());

        /*Timer for Update Dynamic (Data Frame processing) function*/
        DispUpdate_Timer = new Timer(100, new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                UpdateDynamic();
                DispUpdate_Timer.restart();
            }
        });
        DispUpdate_Timer.start();



        TcpOBJ = new TcpClient();
    }



    private void PrintTxtWin(String twstr, int twstyle, boolean newline) {
        try {
            Document doc = textPane_log.getStyledDocument();
            StyleConstants.setItalic(TextSet, false);
            StyleConstants.setBold(TextSet, false);
            StyleConstants.setForeground(TextSet, Color.BLACK);
            switch (twstyle) {
                case 0:
                    StyleConstants.setBold(TextSet, true);
                    StyleConstants.setForeground(TextSet, Color.DARK_GRAY);
                    break;
                case 1: StyleConstants.setForeground(TextSet, Color.BLUE);
                    break;
                case 2: StyleConstants.setForeground(TextSet, Color.BLACK);
                    break;
```

```
                case 3: StyleConstants.setForeground(TextSet, Color.RED);
                    break;
                case 4: StyleConstants.setForeground(TextSet, Color.GREEN);
                    break;
                default:
                    doc.remove(0, doc.getLength());
            }
            if (twstyle >= 0) {
                textPane_log.setCharacterAttributes(TextSet, true);
                if (newline) {
                    doc.insertString(doc.getLength(), twstr+"\n", TextSet);
                } else {
                    doc.insertString(doc.getLength(), twstr, TextSet);
                }
            }
        } catch (BadLocationException ex) {
            System.out.println(ex.toString());
        }
    }
//// Function to get the commands
    private void CommandHandler(String cmds) {
        String txstr, hostname;

        if (cmds.equals("help")) {
            PrintTxtWin("cll - clear log window\n", 1, true);
            PrintTxtWin("tcps - ip address\n", 1, true);
            PrintTxtWin("tx - transmit to server\n", 1, true);
            PrintTxtWin("exit - exit MNGhmi GUI\n", 1, true);
            // PrintTxtWin("tcps {ip address} - \nconnect to TCP server\n",
            // 1, true);
            PrintTxtWin("--- Please enter one of the following options after
tx---\n", 1, true);
            PrintTxtWin("Sensor_type  No_iterations  Sampling_rate Unit \n",
1, true);
            PrintTxtWin("Example S0 10 100 ms\n", 1, true);
            PrintTxtWin("auto (for automatic execution of sensors \n)", 1,
true);
            PrintTxtWin("manual (for exiting from auto mode \n)", 1, true);
            PrintTxtWin("exit (only exiting from server \n)", 1, true);

        } else if (cmds.equals("cll")) {
            PrintTxtWin("", -1, false);
        } else if (cmds.equals("tcps")) {
            PrintTxtWin("Connecting to 172.20.48.31...", 0, true);
            hostname = "172.20.48.31";
            TcpOBJ.tcp_conn(hostname);
            ((CanvasW) panel_canv).UpdateHostname(hostname);
            panel_canv.repaint();

        } else if (cmds.startsWith("tx ")) {
            txstr = cmds.substring(3);
            if (txstr.startsWith("S0")) {
                PrintTxtWin(String.format("sending [%s]", txstr), 0, true);
                TcpOBJ.TcpSend(txstr);
                ((CanvasW) panel_canv).Update_tperiod(txstr);
            }
```

```
            else if (txstr.startsWith("S1")) {
                PrintTxtWin(String.format("sending [%s]", txstr), 0, true);
                TcpOBJ.TcpSend(txstr);
                ((CanvasW) panel_canv).Update_tperiod(txstr);

            }
            else if (txstr.startsWith("S2")) {
                PrintTxtWin(String.format("sending [%s]", txstr), 0, true);
                TcpOBJ.TcpSend(txstr);

                ((CanvasW) panel_canv).Update_tperiod(txstr);


            }
            else if (txstr.startsWith("S3")) {
                PrintTxtWin(String.format("sending [%s]", txstr), 0, true);
                TcpOBJ.TcpSend(txstr);

                ((CanvasW) panel_canv).Update_tperiod(txstr);

            }
            else if (txstr.startsWith("auto")) {
                PrintTxtWin(String.format("sending [%s]", txstr), 0, true);
                TcpOBJ.TcpSend(txstr);

            }
            else if (txstr.startsWith("manual")) {
                PrintTxtWin(String.format("sending [%s]", txstr), 0, true);
                TcpOBJ.TcpSend(txstr);


            }
            else if (txstr.startsWith("exit")) {
                PrintTxtWin(String.format("sending [%s]", txstr), 0, true);
                TcpOBJ.TcpSend(txstr);


            }

            else {
                PrintTxtWin("Write the valid command", 0, true);

            }

        } else if (cmds.equals("exit")) {
            txstr = "exit";
            PrintTxtWin("Communication stopped", 1, true);
            TcpOBJ.TcpSend(txstr);
            System.exit(0);
        } else {
            PrintTxtWin(" *** unknown cmd ***", 3, true);
        }
    }

    /*Initialization of XY data on graph */
    private void InitCharts()
```

```
    {
        int k;
        for (i = 0; i < NumSensors ;i++) {
            for (k = 0; k < CHART_POINTS; k++) {
                Sdata[i][k] = 0.01* k;
                Stime[i][k] = 0.001 * k;
            }
        }
    }


/*************************************************************************/
    /* Function: Data Frame processing
     * Comments: Extracting, converting and passing the
     * sensor data to the graph */

/*************************************************************************/
    private void UpdateDynamic() {
        String rx_str;
        double[] mtime = new double [NumSensors];
        double[] mval = new double [NumSensors];
        double init_time;
        double init_value;

        int k;
        double latest_time_SW = 0 ;
        double latest_time_PB = 0 ;
        double latest_time_CAN = 0 ;
        double latest_time_RTD = 0 ;


        if (TcpOBJ.TcpConnected()) {
            do {
                rx_str = TcpOBJ.PopMessage();
                if (!rx_str.isEmpty()) {

                    // Storing the initial time and value
                 if (rx_str.startsWith("~INIT")) {
                        PrintTxtWin(String.format("rx string: [%s]",
rx_str), 3, true);
                        try {
                                String[] splitStr = rx_str.split(" ");

                                init_value = Double.parseDouble(splitStr[2]);

                                init_time = Double.parseDouble(splitStr[1]);


                        } catch (NumberFormatException e)
{PrintTxtWin(e.toString(), 3, true);}

                    }


                 else if (rx_str.startsWith("~S0")) {
```

```
                        PrintTxtWin(String.format("rx string: [%s]",
rx_str), 3, true);
                        try {

                            i=0;

                            //splitting of string at whitespace
                            String[] splitStr = rx_str.split(" ");


                            // converting string into Double
                            mval[i] = Double.parseDouble(splitStr[2]);
                            mtime[i] = Double.parseDouble(splitStr[1]);


                            // allowing only the new timestamp and data to
show on the graph

                            if(latest_time_SW > mtime[i])
                            {
                                continue;
                            }
                            else
                            {
                                latest_time_SW = mtime[i];
                            }

                        // Shifting the data coming from the sensor left to
show on the graph

                        for (k=0; k<CHART_POINTS-1; k++)
                        {
                            Stime[i][k] = Stime[i][k+1];
                            Sdata[i][k] = Sdata[i][k+1];
                        }

                        Stime[i][CHART_POINTS-1] = mtime[i];
                        Sdata[i][CHART_POINTS-1] = mval[i];
                        S = String.format("S%d", i);

                        Sdata[i][CHART_POINTS-1]= Sdata[i][CHART_POINTS - 1];

                        SW_Chart.updateXYSeries(S, Stime[i], Sdata[i], null);
                        panel_SW.repaint();

                        ((CanvasW) panel_canv).Update(mtime, mval,i);
                        panel_canv.repaint();
                        } catch (NumberFormatException e)
{PrintTxtWin(e.toString(), 3, true);}
                    }

                    // Reading and displaying of data for Push Button

                    else if (rx_str.startsWith("~S1")) {
```

```
                        PrintTxtWin(String.format("rx string: [%s]",
rx_str), 3, true);
                        try {

                            i=1;


                             //splitting of string at whitespace
                            String[] splitStr = rx_str.split(" ");


                            // converting string into Double
                            mval[i] = Double.parseDouble(splitStr[2]);
                            mtime[i] = Double.parseDouble(splitStr[1]);

                            // allowing only the new timestamp and data to
show on the graph

                            if(latest_time_PB > mtime[i])
                            {
                                continue;
                            }
                            else
                            {
                                latest_time_PB = mtime[i];
                            }

                        // Shifting the data coming from the sensor left to
show on the graph

                        for (k=0; k<CHART_POINTS-1; k++)
                        {
                            Stime[i][k] = Stime[i][k+1];
                            Sdata[i][k] = Sdata[i][k+1];
                        }

                        Stime[i][CHART_POINTS-1] = mtime[i];
                        Sdata[i][CHART_POINTS-1] = mval[i];
                        S = String.format("S%d", i);

                        Sdata[i][CHART_POINTS-1]= Sdata[i][CHART_POINTS - 1];

                        PB_Chart.updateXYSeries(S, Stime[i], Sdata[i], null);
                        panel_PB.repaint();

                        ((CanvasW) panel_canv).Update(mtime, mval,i);
                        panel_canv.repaint();
                        } catch (NumberFormatException e)
{PrintTxtWin(e.toString(), 3, true);}
                    }

                    // Reading and displaying of data for CAN Sensor

                    else if (rx_str.startsWith("~S2")) {
```

```
                        PrintTxtWin(String.format("rx string: [%s]",
rx_str), 3, true);
                        try {
                            ((CanvasW) panel_canv).UpdateCANServer(rx_str);

                            i=2;


                             //splitting of string at whitespace
                            String[] splitStr = rx_str.split(" ");


                            // converting string into Double
                            mval[i] = Double.parseDouble(splitStr[2]);
                            mtime[i] = Double.parseDouble(splitStr[1])/1000;

                            // allowing only the new timestamp and data to
show on the graph

                            if(latest_time_CAN > mtime[i])
                            {
                                continue;
                            }
                            else
                            {
                                latest_time_CAN = mtime[i];
                            }


                        for (k=0; k<CHART_POINTS-1; k++)
                        {
                            Stime[i][k] = Stime[i][k+1];
                            Sdata[i][k] = Sdata[i][k+1];
                        }

                        Stime[i][CHART_POINTS-1] = mtime[i];
                        Sdata[i][CHART_POINTS-1] = mval[i];
                        S = String.format("S%d", i);

                        Sdata[i][CHART_POINTS-1]= Sdata[i][CHART_POINTS - 1];

                        CAN_Chart.updateXYSeries(S, Stime[i], Sdata[i],
null);
                        panel_CAN.repaint();

                        ((CanvasW) panel_canv).UpdateCAN(mtime, mval,i);
                        panel_canv.repaint();
                        } catch (NumberFormatException e)
{PrintTxtWin(e.toString(), 3, true);}
                    }

                    // Reading and displaying of data for RTD Sensor

                    else if (rx_str.startsWith("~S3")) {
```

```
                        PrintTxtWin(String.format("rx string: [%s]",
rx_str), 3, true);
                        try {

                            ((CanvasW) panel_canv).UpdateRTDServer(rx_str);
                            i=3;


                             //splitting of string at whitespace
                            String[] splitStr = rx_str.split(" ");


                            // converting string into Double

                            mval[i] = Double.parseDouble(splitStr[2]);
                            mtime[i] = Double.parseDouble(splitStr[1])/1000;

                            // allowing only the new timestamp and data to
show on the graph

                            if(latest_time_RTD > mtime[i])
                            {
                                continue;
                            }
                            else
                            {
                                latest_time_RTD = mtime[i];
                            }


                        for (k=0; k<CHART_POINTS-1; k++)
                        {
                            Stime[i][k] = Stime[i][k+1];
                            Sdata[i][k] = Sdata[i][k+1];
                        }

                        Stime[i][CHART_POINTS-1] = mtime[i];
                        Sdata[i][CHART_POINTS-1] = mval[i];


                        S = String.format("S%d", i);

                        Sdata[i][CHART_POINTS-1]= Sdata[i][CHART_POINTS - 1];

                        RTD_Chart.updateXYSeries(S, Stime[i], Sdata[i],
null);
                        panel_RTD.repaint();
                        ((CanvasW) panel_canv).UpdateRTD(mtime,mval,i);
                        panel_canv.repaint();
                        } catch (NumberFormatException e)
{PrintTxtWin(e.toString(), 3, true);}
                    }

                    else {
                        PrintTxtWin(String.format("Invalid format rx: [%s]",
                                    rx_str), 3, true);
```

```
                        }
                }

                }while (!rx_str.isEmpty());
            }
        }
    private JPanel panel_canv;
    private JPanel panel_CAN;
    private JPanel panel_RTD;
    private JPanel panel_PB;
    private JPanel panel_SW;
    private JTextPane textPane_log;
    private Timer DispUpdate_Timer;
    private SimpleAttributeSet TextSet = new SimpleAttributeSet();
    private TcpClient TcpOBJ;
    private XYChart CAN_Chart = new XYChartBuilder().width(400).height(180)
                                .title("CAN-
Temperature").xAxisTitle("time(s)")
                                .yAxisTitle("Data(oC)").build();
    private XYChart RTD_Chart = new XYChartBuilder().width(200).height(90)
             .title("RTD-Resistance").xAxisTitle("time(s)")
             .yAxisTitle("Data(Ohm)").build();
    private XYChart PB_Chart = new XYChartBuilder().width(400).height(180)
                .title("Pushbutton").xAxisTitle("time(s)")
                .yAxisTitle("Data").build();
    private XYChart SW_Chart = new XYChartBuilder().width(400).height(180)
                .title("Switch").xAxisTitle("time(s)")
            .yAxisTitle("Data").build();
    private final int CHART_POINTS = 10;
    private int i,  NumSensors = 4;
    private String S;
    private double[][] Sdata = new double [NumSensors][CHART_POINTS];
    private double[][] Stime = new double [NumSensors][CHART_POINTS];
}
```

```
/*

================================================================================
==
 Name        : TcpClient.java
 Authors     : Asher Ali
 Version     : 3
 Copyright   : Asher Ali , 03-March-2022
 Description : TCP Client connection application

================================================================================
==
 */

package ngui;
```

```
import java.net.*;
import java.io.*;

/**************************************************************************
***
 * Comments: Defines function related to TCP Client connection, receives
*  data from server and store it in ring buffer for data
*  frame process.
**************************************************************************
**/
public class TcpClient {

    public void tcp_conn(String hostname) {
        Host_Name = hostname;

        Thread ConnTHR = new Thread() {
            public void run() {
                boolean do_terminate;
                do_terminate = false;

                try (Socket socket = new Socket(Host_Name, Port_Num)) {

                    /* parameter to write to the server */
                    output = socket.getOutputStream();
                    writer = new PrintWriter(output, true);

                    /* parameter to receive the string from the server */
                    input = socket.getInputStream();
                    reader = new BufferedReader(new
InputStreamReader(input));

                    String line;
                    Is_Connected = true;

                    do {
                        /*Read the received string and store
                         * it in the ring buffer*/
                        if ((line = reader.readLine()) != null) {
                            SBuff[RingB_InP] = line;
                            if (RingB_Len < RINGB_SIZE) {
                                RingB_Len++;
                            }
                            RingB_InP = (RingB_InP + 1) % RINGB_SIZE;

                            do_terminate = line.startsWith("exit");

                        } else {
                            System.out.println("- no data -");
                            do_terminate = true;
                          }
                    } while (!do_terminate);

                } catch (UnknownHostException ex) {
                    System.out.println("Server not found: " +
ex.getMessage());
```

```
                } catch (IOException ex) {
                    System.out.println("I/O error: " + ex.getMessage());
                }
                Is_Connected = false;
            }
        };
        /*The thread is started here after the definition*/
        ConnTHR.start();
    }

    /*Function to write the command string to the server */
    public void TcpSend(String txstr)
    {
        if (Is_Connected) {
            writer.println(txstr);
        }
    }

    /*This function returns the string that is read in the ring buffer */
    public String PopMessage()
    {
        String rline = "";

        if (RingB_Len > 0) {
            rline = SBuff[RingB_OutP];
            RingB_OutP = (RingB_OutP + 1) % RINGB_SIZE;
            RingB_Len--;
        }
        return rline;
    }

    /*Returns TCP connection status*/
    public boolean TcpConnected()
    {
        return Is_Connected;
    }

    public TcpClient()
    {
        Is_Connected = false;
        Host_Name = "172.20.48.31";
        RingB_InP = 0;
        RingB_OutP = 0;
        RingB_Len = 0;
    }

    /*Declaration*/
    private boolean Is_Connected;
    private String Host_Name;

    /*Should be same as the server port*/
    private final int Port_Num = 50012;

    private OutputStream output;
    private PrintWriter writer;
    private InputStream input;
```

```
    private BufferedReader reader;
    private int RingB_InP, RingB_OutP, RingB_Len;
    private final int RINGB_SIZE = 33;
    private String[] SBuff = new String[RINGB_SIZE];

}


/*

================================================================================
==
 Name        : CanvasW.java
 Authors     : Asher Ali
 Version     : 3
 Copyright   : Asher Ali , 03-March-2022
 Description : Display of TCP connection, CANOpen connection, Sensor Data
               and Time Stamp on the Canvas Panel.

================================================================================
==
 */
package ngui;

import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Graphics2D;

import javax.swing.JPanel;

public class CanvasW extends JPanel {

    private static final long serialVersionUID = 1L;

    @Override
    public void paint(Graphics g) {
        Graphics2D grphc2d = (Graphics2D) g;
        String TCP_str, CAN_str, S0_str, S1_str, S2_str,S3_str,S0TP_str,
S1TP_str, S2TP_str,S3TP_str, RTD_str,tperiod_str;
        int  k, bmaskP,bmaskS;

        g.clearRect(0, 0, super.getWidth(), super.getHeight());
        g.setFont(monoFont);
        TCP_str = String.format("TCP Server: %s", TCP_Server);

        /* TCP Server-Client Connection updation */
        if (TCP_Server.equals("0.0.0.0")) {
            grphc2d.setColor(Color.red);
        } else {
            grphc2d.setColor(m_tGreen);
        }
        g.drawString(TCP_str, 10, 12);

        /* CANOpen Connection updation */
        if (CAN_Server == 0) {
```

```
    CAN_str = "CANopen:  ---";
    grphc2d.setColor(Color.red);
} else {
    CAN_str = String.format("CANopen: Connected");
    grphc2d.setColor(m_tGreen);
}
g.drawString(CAN_str, 400, 12);
grphc2d.setColor(Color.black);

/* CANOpen Connection updation */


tperiod_str = String.format("Sensors Sampling period");

g.drawString(tperiod_str, 600, 67);


/* RTDSensor Connection updation */
if (RTD_Server == 0) {
    RTD_str = "RTDSensor:  ---";
    grphc2d.setColor(Color.red);
} else {
    RTD_str = String.format("RTDSensor: Connected");
    grphc2d.setColor(m_tGreen);
}
g.drawString(RTD_str, 800, 12);
grphc2d.setColor(Color.black);

/* To Display Sensor Data Value and Time Stamp */
S0_str = String.format("S0: %2x  [%10.3f us]", S[0], T_S[0]);
g.drawString(S0_str, 10, 42);
S1_str = String.format("S1: %2x  [%10.3f us]", S[1], T_S[1]);
g.drawString(S1_str, 10, 67);
S2_str = String.format("S2: %.2f oC [%10.3f ms]", CAN_S[2], T_S[2]);
g.drawString(S2_str, 400, 42);
S3_str = String.format("S3: %.2f Ohm [%10.3f ms]", RTD_S[3], T_S[3]);
g.drawString(S3_str, 800, 42);

// To display the sampling period of the sensors

S0TP_str = String.format("S0: [%10.3f %s] ", TP_S[0], U_S[0]);
g.drawString(S0TP_str, 400, 87);
S1TP_str = String.format("S1: [%10.3f %s] ", TP_S[1], U_S[1]);
g.drawString(S1TP_str, 400, 107);
S2TP_str = String.format("S2: [%10.3f %s] ", TP_S[2], U_S[2]);
g.drawString(S2TP_str, 800, 87);
S3TP_str = String.format("S3: [%10.3f %s] ", TP_S[3], U_S[3]);
g.drawString(S3TP_str, 800, 107);

/* To set and fill the color into the rectangular shape pushbutton
 *  box according to pushbutton value */
bmaskS = 1 << (N_SW - 1);
for (k = 0; k < N_SW; k++) {
    if ((S[0] & bmaskS) != 0) {
        grphc2d.setColor(Color.green);
        grphc2d.fillRect(180+k*25, 90, 15, 15);
```

```
            } else {
                grphc2d.setColor(Color.red);
                grphc2d.drawRect(180+k*25, 90, 15, 15);
            }
            bmaskS >>= 1;
        }

        /* To set and fill the color into the oval shape switches
         * according to received switch value */
        bmaskP = 1 << (N_PB - 1);
        for (k = 0; k < N_PB; k++) {
            if ((S[1] & bmaskP) != 0) {
                grphc2d.setColor(Color.green);
                grphc2d.fillOval(10+k*25, 90, 15, 15);
            } else {
                grphc2d.setColor(Color.red);
                grphc2d.drawOval(10+k*25, 90, 15, 15);
            }
            bmaskP >>= 1;
        }
    }

    /* To store the sensor data value and time stamp for the enabled sensor
*/
    public void Update(double [] mtime, double [] mval , int i)
    {
        T_S[i]= mtime[i];
        S[i]= (int) mval[i];
    }

    public void Update_tperiod(String txstr_can)
    {
        if (Double.parseDouble(txstr_can.substring(1,2)) == 0)
        {
            //splitting of string at whitespace
            String[] splitStr = txstr_can.split(" ");

            // converting string into Double

            TP_S[0] = Double.parseDouble(splitStr[2]);

            // printing the Unit of sampling period

            U_S[0] = splitStr[3];
        }



        else if (Double.parseDouble(txstr_can.substring(1,2)) == 1)
        {

            //splitting of string at whitespace
            String[] splitStr = txstr_can.split(" ");

            // converting string into Double
```

```
            TP_S[1] = Double.parseDouble(splitStr[2]);

            // printing the Unit of sampling period

            U_S[1] = splitStr[3];
        }
        else if (Double.parseDouble(txstr_can.substring(1,2)) == 2)
        {
            //splitting of string at whitespace
            String[] splitStr = txstr_can.split(" ");

            // converting string into Double

            TP_S[2] = Double.parseDouble(splitStr[2]);

            // printing the Unit of sampling period

            U_S[2] = splitStr[3];
        }
        else if (Double.parseDouble(txstr_can.substring(1,2)) == 3)
        {
            //splitting of string at whitespace
            String[] splitStr = txstr_can.split(" ");

            // converting string into Double

            TP_S[3] = Double.parseDouble(splitStr[2]);

            // printing the Unit of sampling period

            U_S[3] = splitStr[3];
        }

    }

    /* To store the CANOpen sensor data value and time stamp */
    public void UpdateCAN(double [] mtime, double [] mval , int i)
    {
        T_S[i]= mtime[i];


        CAN_S[i] = mval[i];

    }

    /* To store the CANOpen sensor data value and time stamp */
    public void UpdateRTD(double [] mtime, double [] mval , int i)
    {
        T_S[i]= mtime[i];


        RTD_S[i] = mval[i];

    }
```

```
/* To store the IP address of host */
public void UpdateHostname(String newhost)
{
    TCP_Server = newhost;

}

/* Function to check whether CANOpen sensor is enabled and set CAN_Server
 * to display connection status on CANvas  */
public void UpdateCANServer(String txstr_can)
{
    if (Double.parseDouble(txstr_can.substring(2,3)) == 2) {

        if (RTD_Server == 1)
        {
            RTD_Server =0;
        }
        if ((txstr_can.substring(0,3)).equals("~S2")){
                CAN_Server = 1;
                                                    }
            else {
                CAN_Server = 0;
            }
        }
    }

public void UpdateRTDServer(String txstr_rtd)
{
    if (Double.parseDouble(txstr_rtd.substring(2,3)) == 3) {

        if (CAN_Server == 1)
        {
            CAN_Server =0;
        }
        if ((txstr_rtd.substring(0,3)).equals("~S3")){
                RTD_Server = 1;
                                                    }
            else  {
                RTD_Server = 0;
            }
        }
    }

/* Default initialization */
public CanvasW()
{
    TCP_Server = "0.0.0.0";
    CAN_Server = 0;
    RTD_Server = 0;
    for (i=0; i < NumSensors; i++)
    {
        S[i] = 0;
        T_S[i] = 0.0;
    }
}
```

```
    /* Declarations */
    private static Color m_tGreen = new Color(0, 128, 0);
    private static Font monoFont = new Font("Monospaced", Font.BOLD, 14);
    private String TCP_Server;
    private int CAN_Server;
    private int RTD_Server;
    private int i,  NumSensors = 4;
    private final int N_SW = 8;
    private final int N_PB = 5;
    private int[] S = new int[100];
    private double[] CAN_S = new double[3];
    private double[] RTD_S = new double[4];
    private double[] T_S = new double[100];
    private double[] TP_S = new double[100];
    private String [] U_S = new String [100];
}

/*
============================================================================
** Name        : data_process.c
** Author      : Nikhil Narayanan
** Version     :
** Copyright   :
** Description : Measurement data processing
**
============================================================================*/

#include <stdio.h>
#include <math.h>

int main()
{
  FILE *file;
  char line[50],sname[10],filename[50];
  float tvalue, tstime, last_time = 0, diff;
  int i,count5, count2, count10,j;
  for(j=0;j<=3;j++){
    sprintf(filename,"/media/nikhil/SSD/Data_samples/SW_1ms_%d.txt",j);
    file = fopen(filename,"rb");
    count5 = 0;
    count2 = 0;
    count10 = 0;
    last_time = 0;
    while (fgets(line, sizeof(line), file)) {
       //printf("%s", line);
      i++;
      sscanf(line,"%s %f %f",sname, &tstime, &tvalue);
      if(last_time == 0){
        last_time = tstime;
        continue;
      }
      diff = tstime - last_time - 1000.000000;
      if(fabs(diff)<= 50.000000){
        count10++;
      }
      if(fabs(diff)<= 20.000000){
```

```
        count5++;
      }
      if(fabs(diff)<= 10.000000){
        count2++;
      }
      //printf("%d %s %f %f lasttime=%f Diff =%f\n", i,sname, tstime,
tvalue,last_time,diff);
      last_time = tstime;
    }
    fclose(file);
    printf("File = %s\n",filename);
    printf("Count10 = %d, count5 = %d, count2 = %d\n",
count10,count5,count2);

  }
  return 0;
}
```