University of
South-Eastern Norway

Faculty of
Technology, Natural Sciences and Maritime Sciences
(TNM)

Department of
Electrical Engineering, Information Technology and
Cybernetics (EIK)

Introduction to data communication with C# programming

Nils-Olav Skeie

September 27, 2021

Contents

Preface

This document gives an introduction to programming in C# for serial and network communication. This introduction will cover serial data communication, a few serial communication protocols and implementation in C# using a terminal console program. This introduction will also cover basic network communication with the TCP/IP protocol using sockets and implementation of a GUI chat application in C#. This is however not an introduction to neither data communication nor programming, but a combination of these two subjects. It is assumed that the reader has basic understanding of both data communication and programming before reading this document. The versions of this document are:

Revision	**Changes / Extensions**	**Who**	**Date**
0.1	First version for some BSc and MSc courses at USN/Porsgrunn	NOS	AUG−20
0.2	Minor updates and fixes	NOS	SEP-21

Nomenclature

This section gives a list of abbreviations used in this lecture note.

Abbreviation	**Meaning/Explanation**
CAN	Controller Area Network
EOBD	European On-Board Diagnostics
FIFO	First In First Out
GPS	Global Position System
M2M	Machine to machine
MQTT	Message Queuing Telemetry Transport
. NMEA	National Marine Electronics Association
OBD	On-Board Diagnostics
OSI	Open System Interconnection
Rx	Receive data
TCP/IP	Transport Control Protocol / Internet Protocol
Tx	Transmit data
UART	Universal asynchronous receiver and transmitter
USB	Universal Serial Bus

Chapter 1

Data communication in C#

1.1 Introduction

The most used method for external communication with a computer is by serial communication. Serial communication means that there is a communication media where the data is transformed in serial form, each bit in every byte is transformed one by one on a single communication medium. The medium can be an electrical wire, in the air (wireless) or an optical cable (using light), and a protocol is needed to describe how the information is transformed and coded on the medium. A protocol contains a set of rules describing how to convert the information in the data between any applications and the communication media. USB, TCP/IP, MQTT, RS-485, CAN and RS-232C are a few of the communication protocols used where a protocol description is documenting the structure of the data and the usage of the communication media. This section is an introduction for using C# for accessing serial communication functions on a computer, for external communication with other communication devices, based on some sort of protocols.

1.2 Serial port data communication

1.2.1 Introduction

Data communication is important for communication between computers, for stand-alone devices like a GPS device, on local area network (LAN) and on internet. Every time you want to send or receive an online message, send or receive a SMS (short message service) send or receive an email, access a web page on internet or download any information from internet, data communication is needed. If you connect a device to your computer, like a GPS device, an USB memory stick, a mobile phone or a camera, data communication is also needed. A computer with external communication capabilities is often referred to as a node and a set of nodes need to use the same structure of the data and the communication media to be able to communicate. However, this structure can be very complex, as shown in Figure 1.1.

This structure is defined by a protocol document describing the details for how to use the communication media and how to interpret the data information. The protocol is a set of rules trying to structure sending and receiving of information. This can be compared with two humans that are trying

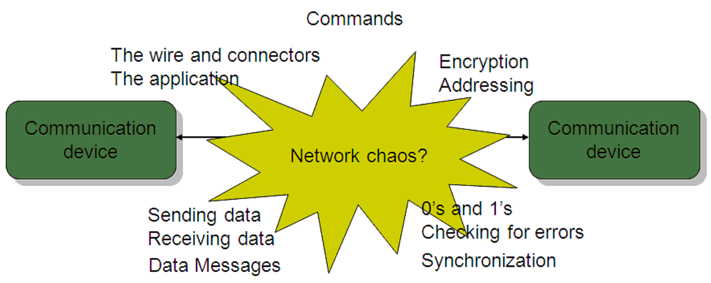


Figure 1.1: An overview of the challenges when two or more nodes (computers) need to communicate using a communication media.

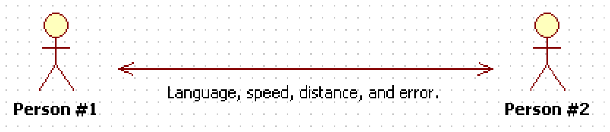


Figure 1.2: Compare a protocol with the situation where two people should be able to talk with and understand each other.

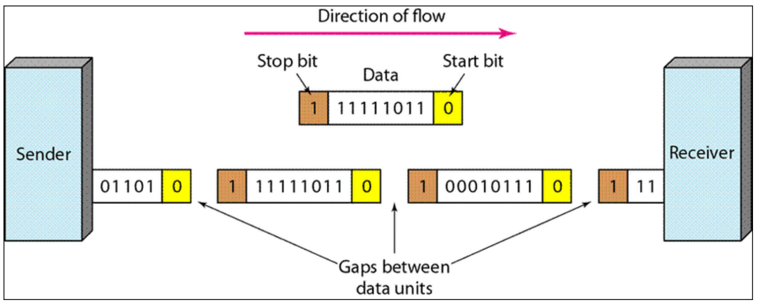


Figure 1.3: Asyncronous transmission using a start and stop bit for syncronization (copy from NetworkEncyclopedia.com / Aug-20).

to communicate, they also need to have a set of rules to understand each other like using a language that both understand, not talk at the same time, and not being too far away from each other, as indicated in Figure 1.2.

Since data communication contains many different states and rules, the Open System Interconnection reference model (OSI model) (Houldsworth 1990), (Levermore 2000), (Stevens 1990), (Stallings 2001) is often used to better structure the situation. The OSI model is a seven layer model defining different types of functionality that must be fulfilled by the protocol. The lowest layer in the OSI model is the physical layer describing the functionality for a physical connection between two or more nodes. This layer is describing the main functionality of the serial port communication. This connection may also be named M2M communication in some literature, short for machine to machine communication.

1.2.2 Physical layer

The physical layer may be wired, optical or wireless (Stallings 2001), and describes how to connect to the physical communication media, how to transfer the data on the communication media and how to handle the data flow on the communication media. The wired connection will be either parallel or serial communication. Parallel is using one wire for each bit, while serial is handling each bit in time slots on the same wire. Parallel is faster but serial is cheaper and can be used over longer distances. So serial communication is used most. Optical is using light (laser) as the communication carrier instead of electrical current as for a wired connection. Wireless can be protocols like Bluetooth and WiFi (Stallings 2001), using radio frequencies as the communication carrier. This chapter will focus further on wired connections only, for serial data communication.

1.2.3 Asynchronous serial communication

Serial data communication will be either synchronous or asynchronous communication where asynchronous is most used. In synchronous communication there will be a continuos stream of data bits and an external synchronization signal is needed to synchronize the sender and the receiver. This synchronization signal must use an extra wire. The asynchronous communication will divide the data stream into characters and add synchronization bits for each of the characters in the data stream. No external synchronization signal is needed however the transmission will take a little more time as the synchronization bits must be transmitted along with the data bits. Figure 1.3 shows asynchronous communication where each byte has a start bit and a stop bit for synchronization between the transmitter (sender) and the receiver. Each byte in this figure consists of eight bits.

The RS-232C, RS-422 and RS-485 protocols are some examples of asynchronous protocols used a lot on computer systems, and these protocols need a set of communication parameters to be configured

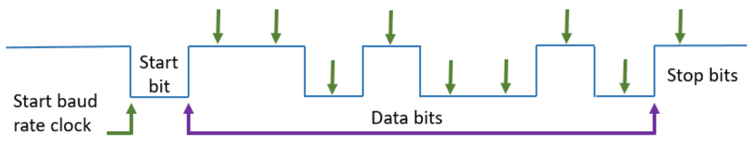


Figure 1.4: The transmiision of the bits in asyncronous serial communication. It will start with a start bit, then the data bits, and a stop bit at the end.

before starting transferring the data stream. RS-232C has normally only one transmitter node and one receiver node, RS-422 has normally one transmitter node and several receiver node, while RS-485 has normally several both transmitter and receiving nodes.

The number of data bits and stop bits can be configured while one start bit is fixed. The communication parameters that can be configured are:

- Baud rate; defining the transmission speed, the number of bits per. second. The baud rate is normally either 300, 600, 1200, 2400, 4800, 9600, 19200, 38400 or 115200.
- Data bits; defining the number of bits in each character to transmit. The number of data bits are either 5, 6, 7 or 8.
- Parity check; single bit summing up as either None, Even or Odd parity. Even or odd parity will add an extra bit to the data stream, before the stop bit.
- Stop bits; number of bits after the data bits and any parity bit. The number of stop bits are either 1, 1.5 or 2.

The start bit is used for synchronization as the receiver(s) do not know when any node is transmitting. The start bit is normally changing the electrical carrier voltage level from high to low indicating that new data will be transmitted. The start bit will also synchronize the baud rate system for the receiver as the baud rate system is used to define when a new bit is to be transmitted and when the receiver can read this next bit. Figure 1.4 shows the message package, the transmission of one byte, starting with a start bit and ending with the stop bit. The figure also shows when the receiver should check for a new bit, indicated by the vertical green arrows.

The parity bit will be added after the data bits, and before the stop bit(s) if even or odd parity is configured. The extra bit will be 0 or 1 to fulfill the parity configuration. If the data bits are 1101 1111, as shown as one byte in Figure 1.4 will the number of "1" be 7, an odd number of bits. If odd parity is selected will the extra parity bit be "0", or if even parity is selected will the extra parity bit be "1". Parity bit checking can be useful if communication in a noise environment, or over a long distance.

The conversion between the data byte and the data bits, and including communication parameters, is handled by a hardware chip named universal asynchronous receiver and transmitter (UART). The UART will control the baud rate, the number of data bits, any optional parity check, and include the start and stop bits. The frame is the data bits and is part of the total package as shown in figure 1.4. The UART will handle any framing errors meaning any errors of the number of data bits. The reasons can be configuration of wrong baudrate or wrong number of data bits. The UART also contains a first in first out buffer (FIFO) for the receiving part, for example 16 bytes, in addition to the parity and framing error checking. The FIFO buffer allows the serial port to receive more characters before any software driver needs to read these characters. The UART is shown in figure 1.5 where UART1 is part of the transmitter node transmitting a character from the TX (transmitter) part to the RX (receiver) part of the receiver node. So UART 2 is part of the receiver node.

The control of the UART is handled by the drivers of the operating system, and most programming languages have ready made libraries for communication with these drivers. This document will explore the serial port library in C#.

1.2.4 Communication protocols

Before looking into the serial port programming, two asynchronous serial protocols will be exploited, that will be used as examples in the programming section. These protocols are the National Marine Electronics Association (NMEA) 183 protocol that is often used for GPS (Global Position System), and the on-board diagnostics (OBD) used as a network inside most cars produced after 2000. The NMEA

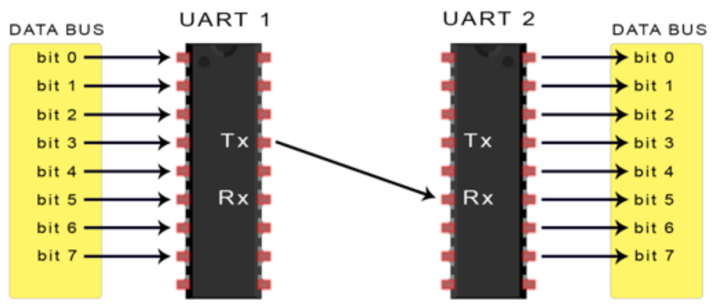


Figure 1.5: The UART hardware chips converting between data bytes and data bits for asyncronous communication between two nodes.

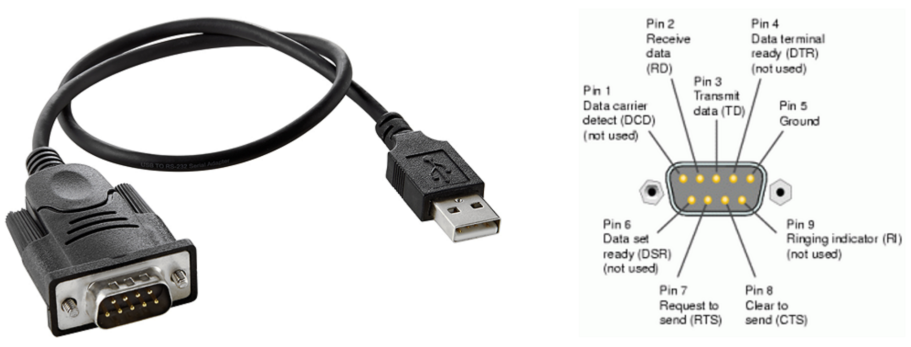


Figure 1.6: One version of an USB to serial port adapter, including the pins for a 9 pins serial port connector. The connector and pinning is defined in the RS-232C protocol.

183 is a one way protocol only so the transmitter is cyclic transmitting all necessary information and any receiver needs to listen to the information and store only the information needed. So only one node can be the transmitter, the rest of the nodes will be receivers. The OBD protocol is a two-way protocol where all the nodes can be either a transmitter or a receiver. Any node can send a request on the network and the node with the information will answer within a specific time frame. However, before digging into these two different protocols, a short introduction on connecting to the serial port is needed.

Serial port

Older computers had one or several RS−232C or RS−485 ports while modern computers only have USB ports. However, many versions of USB to serial port adapters exist, one of these is shown in Figure 1.6.

The three most important pins of the serial port connector is the receive data (RD) pin (rx: pin 2), transmit data (TD) pin (tx: pin 3) and ground (GND) pin (pin 5). Pin 2 is the Rx (receive) connection and pin 3 is the Tx (Transmit) connection as shown in Figure 1.5. This is shown for 9 pins connectors, but there exists 25 pins connectors as well. One way of testing such an adapter is by interconnecting pin 2 and pin 3 and make a test option in the software just to transmit a dummy text and check receiving the same text. This test option is often named loopback testing, and will test both the hardware and the software for that node.

NMEA 183 protocol

The NMEA 183 protocol is a cyclic protocol where the transmitting node is sending a set of messages in a cyclic manner, and is the protocol used for many GPS devices. One is these devices is the Pharos iGPS−180, shown in Figure 1.7. This is a standalone device, the power requirement is 5 V (normally from the USB port) and the protocol is based on RS-232C, the baud rate is 4800, 8 data bits, no parity and one stop bit.

The GPS device has an integrated antenna and is using the NMEA protocol for transmitting the GPS information. The NMEA protocol messages have the following format:

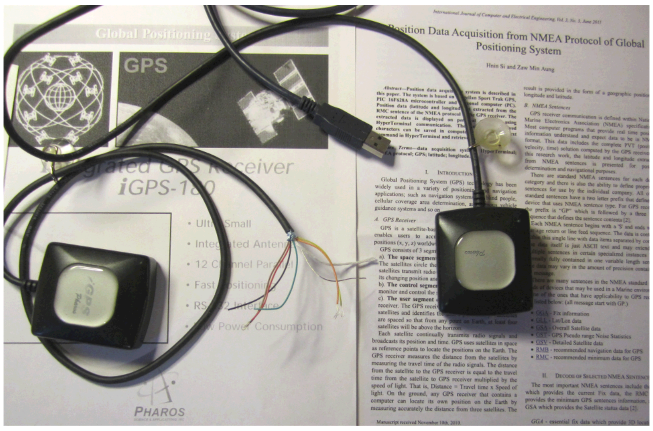
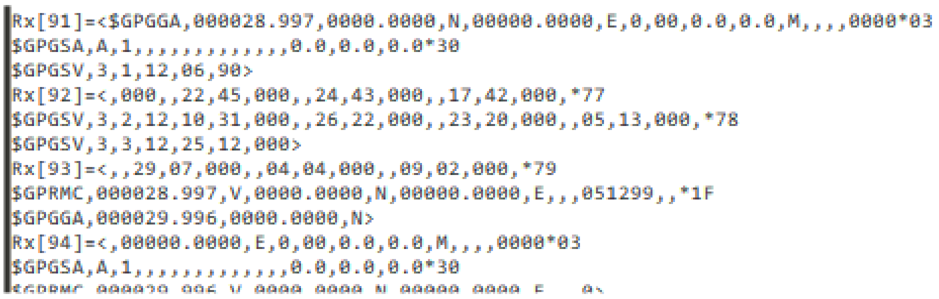


Figure 1.7: GPS devices are hardware devices available in many sizes and forms. The Pharos iGPS-180 is the device used for collecting the data in this document. The figure shows two such devices, one with an USB connection and one with just power and communication connection.



```
Rx[91]=<$GPGGA,000028.997,0000.0000,N,00000.0000,E,0,00,0.0,0.0,M,,,,0000*03
$GPGSA,A,1,,,,,,,,,,,,,0.0,0.0,0.0*30
$GPGSV,3,1,12,06,90>
Rx[92]=<,000,,22,45,000,,24,43,000,,17,42,000,*77
$GPGSV,3,2,12,10,31,000,,26,22,000,,23,20,000,,05,13,000,*78
$GPGSV,3,3,12,25,12,000>
Rx[93]=<,,29,07,000,,04,04,000,,09,02,000,*79
$GPRMC,000028.997,V,0000.0000,N,00000.0000,E,,,051299,,*1F
$GPGGA,000029.996,0000.0000,N>
Rx[94]=<,00000.0000,E,0,00,0.0,0.0,M,,,,0000*03
$GPGSA,A,1,,,,,,,,,,,,,0.0,0.0,0.0*30
$GPRMC,000029.996,V,0000.0000,N,00000.0000,E,,,0>
```

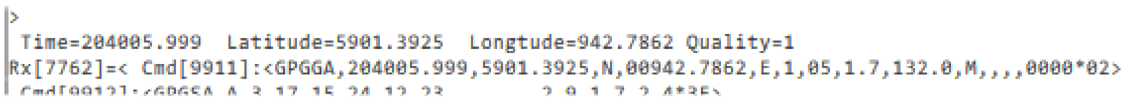
Figure 1.8: An example of some of the NMEA commands like GPGGA, GPGSV, GPRMC and GPGSA.

$ Command , Param#1 , Param#2 , .., Param#N * ChkSum

where all the fields are based on printable ASCII characters. The $ indicates the start of a new message, the Command indicates the type and number of parameters (1 to N), comma is the standard field separator, and * (asterisk) indicates the checksum of the message. The asterisk is immediately followed by a checksum represented as a two-digit hexadecimal number, indicating the end of one NMEA message. The checksum is the bitwise exclusive OR of the ASCII codes of all characters between the $ and *. The message ends with the <CR><LF> codes. An example of some of the NMEA messages is shown in Figure 1.8.

Figure 1.8 shows one of the challenges with asynchronous serial communication as one and one character is sent. The figure shows that the receive function has been used four times, Rx[91] to Rx[94]. The first read function, Rx[91], contains three NMEA messages. However, only the first part of the last NMEA message is read, the $GPGSV command. This is quite common when reading asynchronous serial communication protocols and the software must include logic to merge the last part of this read function and the first part of the next read function. As shown in the figure, the NMEA messages start with the $ sign and end with checksum field starting with the * sign where each field is separated by the , sign.

The most used commands for GPS information in the NMEA protocol are, where some of them are also shown in Figure 1.8:



```
>
 Time=204005.999  Latitude=5901.3925  Longtude=942.7862 Quality=1
Rx[7762]=< Cmd[9911]:<GPGGA,204005.999,5901.3925,N,00942.7862,E,1,05,1.7,132.0,M,,,,0000*02>
 Cmd[9912]:<GPGSA,A,3,17,15,24,12,23,       2,9,1,7,2,4*3E>
```

Figure 1.9: An example of the GPGGA command with valid information, at a location at Stathelle in Norway.

Command	Description	Fields
GPGGA	Global Positioning System Fix Data	10 (12)
GPGLL	Geographic Latitude and Longitude	4
GPGSA	Satellite status	6
GPGSV	Satellites in view	7
GPRMC	Recommended Minimum sentence C	8
GPVTG	Track made good and ground speed	4

The first two characters of the command field defines the sending device and the last characters are actually the NMEA command. The sending device can be either GP, GL or GN. The NMEA GGA command (GPGGA) can be used to receive the location of a moving device like a car. Figure 1.8 shows a GGA command, but before the GPS device has received data from enough satellites, the contents is therefor marked as invalid (field #6 is 0). Also note that one of the GGA messages are broken into two parts (in Rx[93] and Rx[94]), and must be merged into one message by logic in the software. The GPS device will use some time to get information from a set of satellites before valid information can be added to these messages. Figures 1.9 shows a message (Rx[7762]) with valid information for a GGA command (field #6 is 1).

The GGA command description in the NMEA protocol documentation has the following information for the different fields (in the message):

Field	Description	Example
1	UTC time from satellite	20:40:06 UTC
2	Latitude	5901.3925
3	Latitude direction	N
4	Longitude	942.7862
5	Longitude direction	E
6	GPS quality: 0=invalid, 1=GPS fix, 2=DGPS fix (0-8)	1
7	Number of satellites	05
8	Horizontal dilution of position	1.7
9	Altitude, Meters, above mean sea level	132.0 M
10	Height of geoid (mean sea level) above WGS84 ellipsoid	
11	Empty field, time of seconds since last DGPS update	
12	Empty field, DGPS station ID number	
	checksum after the * sign	02

The example column (to the right) contains the values from Figures 1.9. The first field, after the command, is an accurate UTC time showing that the information was received at 20:40:06 UTC time, being 22:40:06 local time (Norwegian summer time). The format of latitude and longitude coordinates in the NMEA format is (d)ddmm.mmmm where d=degrees and m=minutes. The degree part can be calculated as an integer by dividing with 100, and minutes part by dividing the rest with 60. The GPS latitude position based on the values from the example column will be:

$$\begin{aligned} 5901.3925/100 &= 59 \\ 1.3925/60 &= 0.023208 \end{aligned} \tag{1.1}$$

and the GPS longitude position will be:

$$\begin{aligned} 942.7862/100 &= 9 \\ 42.7862/60 &= 0.713103 \end{aligned} \tag{1.2}$$

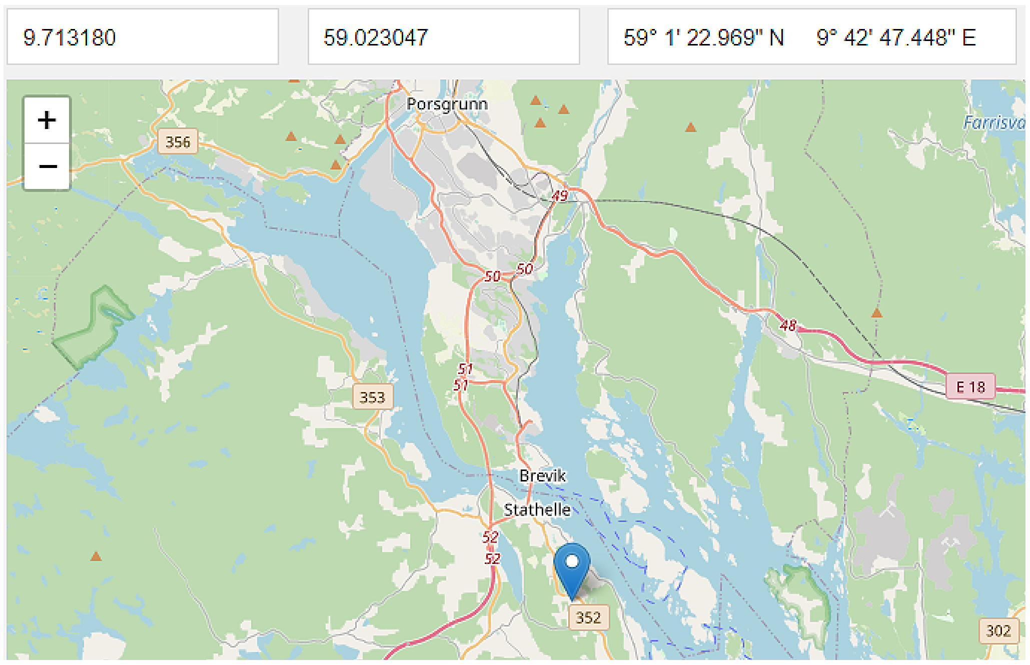


Figure 1.10: A map of the Grenland area in Norway, with the Porsgrunn, Brevik and Stathelle cities. The blue mark is the location of the GPS device at Stathelle, with the position on top of the map.

The direction factor for the latitude position is N=+1 and S=−1, and the direction factor for the longitude position E=+1 and W=−1. The address position for the GPS device is shown at the top of the map in Figure 1.10 with the longitude position = 9.713180 and the latitude position = 59.023047, very close to the position received from the GPS device. These positions are:

Type	Latitude	Longitude
Map	59.023047	9.713180
GPS	59.023208	9.713103

The deviation is due to that the position in Figure 1.10 is based on a street address while the GPS device has a specific position within that address area. Older GPS devices, like this device, have an accuracy of about 5 m while newer GPS devices (newer than 2018) may have an accuracy of about 30 cm. A GPS device can then be used to receive and store the position of a moving object, like a car.

GPS devices using the NMEA protocol has either a serial port or an USB port for connection to the computer, and the serial port (COM port) library can be used in C#. To get more information for a specific moving object, other protocols must be used. The OBD protocol is such a protocol for a car.

OBD-2 / EODB protocol

The On-Board Diagnostic (OBD-2 or OBD-II) standard is a requirement for all cars sold in the United States from 1996 and is a protocol for a network of sensors, actuators and controllers inside cars. European On-Board Diagnostics (EOBD) is the European equivalent of the US OBD II standard, which applies to petrol cars sold in Europe from 2001 (and diesel cars three years later). Cars are using the Controller Area Network (CAN bus) to let microcontrollers and devices to communicate without the need for a server, so the CAN bus is the local area network (LAN) for all the devices installed in a modern car. The OBD protocol is defining the connection port, the signals and the commands. A modern car includes an industrial control system with several distributed control systems like engine controller, braking system (ABS) controller, transmission controller, suspension controller and air conditioner controller to mention some. One possible option for an external communication with this network (LAN) is by using the OBD port.

The CAN protocol is a complex protocol that can be hard to implement in software. A better option is to use the ELM327 interface which is a protocol converter converting between the CAN commands and type of Attention (AT) commands. AT commands is normally used for communication with modems

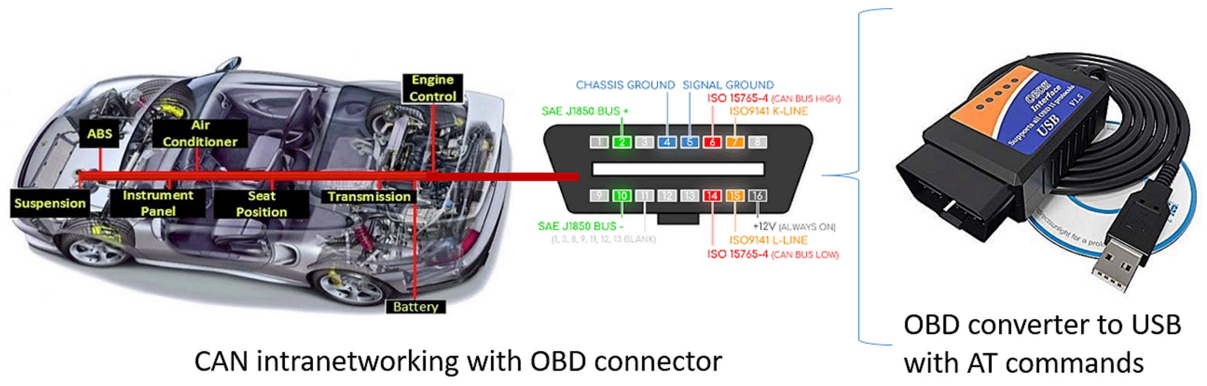


Figure 1.11: The OBD connector, and an example of the ELM327 interface.

where the commands normally start with AT. The ELM327 implementation has customized a set of these commands (Elm 2016). The OBD port and the ELM327 interface is shown in Figure 1.11.

The ELM327 interface can connect to a computer using either USB, Bluetooth or WiFi connection[1]. The easiest way is using an USB connection as this connection can be used as a serial port (COM port) in C#, just like the NMEA protocol. Unlike the NMEA protocol, the OBD protocol is a two-way protocol meaning that the software must first send a request for a specific information, wait for the system to respond, and then send a new request. In this case must the software also handles any timeout situation in case the system is not responding. The protocol baud rate is either 9600 or 38400 depending on the setting of the ELM327 device, 8 data bits, no parity and one stop bit.

One possible startup sequence of AT commands for communicating with the ELM327 interface is:

	Tx	**Rx**	**Comments**
1	ATZ	ATZ ELM327 v1.5	Device reset
2	ATE0	ATE0 OK	Message echo On=1/Off=0
3	ATH0	OK	Message header On=1/Off=0
4	ATL0	OK	Message Line feed On=1/Off=0
5	ATSP0	OK	Set protocol: 0=Auto (Different CAN protocols)
6	ATDP	AUTO	Display protocol type
7	ATI	ELM327 v1.5	Device identification
$8 - N$			Looping getting the values

Note that some of these commands may take some time before an answer is received. As shown in Figure 1.11 the car has several controllers monitoring and controlling a set of devices as sensors and actuators. The OBD protocol is handling a set of parameters and these parameters are linked to values handled by these controllers. Many of the these parameters are defined as standard parameters in OBD and are assigned a parameter id number (PID) that will be used by the protocol to get the parameter value.

The OBD protocol is defining different service types or modes and the most important service types are:

Type	**Description**
01	Show current data
03	Show stored diagnostic trouble codes
04	Clear diagnostic trouble codes and store values

The most used service type is 01, to get the current value for a specific parameter. The service types 03 and 04 are used to read any error codes (often combined with a warning on the dash board), and to clear these error codes. The OBD protocol defines at least 10 different service types, check the protocol documentation for more information.

[1]Many ELM327 devices contain old hardware and will not be supported by standard Windows 10 or 11 drivers. In these cases, add an older serial driver supporting either the CH340 (Arduino) or PL2303HXA hardware depending on your ELM327 device.

Using the service type 01, a set of parameters can be read from the engine controller. The OBD protocol is defining the parameter ids to read these parameters and a few of these parameters ids are:

PID Hex	PID Dec	Description	Min Value	Max Value	Unit	Rx bytes	Equation
04	4	Calculated engine load	0	100	%	1 (A)	$\frac{A*100}{255}$
05	5	Engine coolant temperature	-40	215	°C	1 (A)	$A-40$
$0C$	12	Engine speed	0	16383.75	rpm	2 (A+B)	$\frac{(256*A)+B}{4}$
$0D$	13	Vehicle speed	0	255	km/h	1 (A)	A
$0F$	15	Intake air temperature	-40	215	°C	1 (A)	$A-40$
11	17	Trottle position	0	100	%	1 (A)	$\frac{A*100}{255}$

Information about more PID can be found in the OBD protocol documentation, a lot is available on internet, like the OBD-II PIDs web site[2]. Many car manufacturers may also have defined their own PIDs in addition to the standard PIDs.

The serial port driver in C# can be used together with the ELM327 device to get information from the OBD port. The commands will be to set up and initialize the ELM327 device and start requesting information. The transmit commands will be:

$$ServiceType\ PIDCode\ (Hex)\ <CR><LF>$$

and the receive information will be:

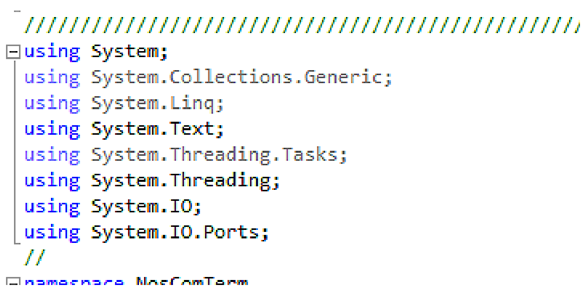
$$0x40 + ServiceType\ AnswerBytes\ (Hex)\ <CR><LF>$$

An example of the startup communication with the ELM327 device is shown in the following list, where Tx is the command sent from C# application and Rx is the data received by the C# application. The list of the first commands are:

```
16-Apr-2020;16:03:16;Tx;ATZ
16-Apr-2020;16:03:17;Rx;ELM327 v1.5
16-Apr-2020;16:03:17;Tx;ATE0
16-Apr-2020;16:03:18;Rx;ATE0 OK>
16-Apr-2020;16:03:18;Tx;ATH0
16-Apr-2020;16:03:18;Rx;OK>
16-Apr-2020;16:03:18;Tx;ATL0
16-Apr-2020;16:03:19;Rx;OK>
16-Apr-2020;16:03:19;Tx;ATSP0
16-Apr-2020;16:03:23;Rx;OK>
16-Apr-2020;16:03:23;Tx;ATDP
16-Apr-2020;16:03:24;Rx;AUTO>
16-Apr-2020;16:03:24;Tx;ATI
16-Apr-2020;16:03:25;Rx;ELM327 v1.5
16-Apr-2020;16:03:26;Tx;010D
16-Apr-2020;16:03:26;Rx;41 0D 08
16-Apr-2020;16:03:26;Tx;0104
16-Apr-2020;16:03:26;Rx;41 04 67
```

A timestamp is included and note that the respons of the ATSP0 command, set the specific protocol between the ELM327 device and the CAN bus, takes almost four seconds. However, the normal timing is faster, about 200 to 300 ms for each message sequence. The last four lines are OBD information from the vehicle where (notice that all values are hexadecimal values):

[2]OBD-II PIDs web site: https://en.wikipedia.org/wiki/OBD-II_PIDs (Oct-19).



```
////////////////////////////////////////////////////
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;
using System.IO;
using System.IO.Ports;
//
namespace NosComTerm
```

Figure 1.12: Include the serial port library with the “using System.IO.Ports;” statement.

Id		**Service**	**Command**	**Value**	**Description**
1	Tx	01	$0D$		Request speed
2	Rx	41	$0D$	08	Speed response
3	Tx	01	04		Request engine load
4	Rx	41	04	67	Engine load response

The first request is for the vehicle speed, the response starts with $0x41$ which is $0x40 + 01$ which is a valid service type 01 response. The command is $0x0D$ so the current speed is 8 km/h. The second request is the engine load, and the engine load will be (see the PID list):

$$\frac{0x67 * 100}{255}\% = \frac{103 * 100}{255}\% = 40,4\%$$

The application must, in a cyclic manner, request the wanted parameters from the car. Parameters that are changing often should be requested more frequently than parameters not changing that much. Parameters changing often can be the vehicle and engine speed while parameters not changing that often can be the engine coolant temperature and the intake air temperature. An evaluation of which parameters that are important for your application and how often each parameter should be recorded should be decided in the analysis section of the software development.

1.3 Serial port programming in C#

1.3.1 Introduction

Serial port programming in C# is quite straightforward by using the standard serial library. Include the serial port library in your code, as shown in Figure 1.12, and the serial port can be used like a text file meaning that the port can be open, read from, written to and be closed.

Note that open is exclusive so only one application can open a specific serial port at a given time and therefor it is important to check for a valid open response. For the same reason remember to close any serial port no longer in use in your application, including when changing from one serial port to another serial port.

In this chapter a console application will be developed to test the basic serial port functions with some snippets of source code of using the serial ports and decoding the protocols. As always when developing a software application, a use case diagram should be made and then choose the type of tier architecture. An use case diagram for the application is shown in Figure 1.13. The use case shows the user functions as 1) select a serial (COM) port, 2) set the parameters for this COM port, 3) select the protocol for decoding the information on this COM port and 4) handling the data communication on the COM port.

A three tier architecture is choosen, one tier for the user interface (user interaction), one tier for the business logic of the application meaning handling the logic of the serial protocols and the information in the application, and one tier for the data layer to handle the serial port. This tier should also handle any logging of data to a file, if needed. A tier can also be named a layer in the literature. The reason for a tier architecture is to get a better structure of the application, and to better separate the code for user interface, the business logic and the data handling.

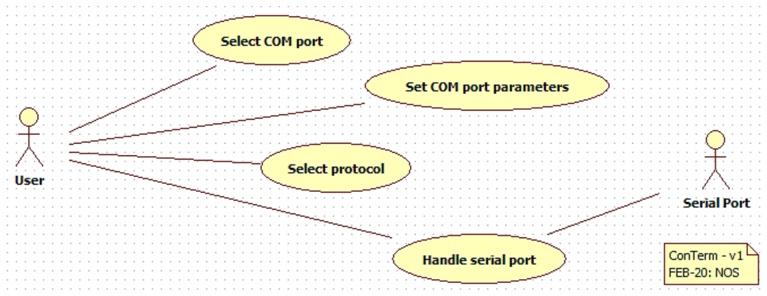


Figure 1.13: A use case diagram for the terminal application, with the user functions of selecting a COM port, set the COM port parameters, select the protocol for that COM port, and handling the data communication on that specific COM port.

1.3.2 Three tier architecture

The application was developed as a C# console application using the Visual Studio IDE. The IDE creates a console template with the Program class and the static Main() method. The class ProgramSetupConsoleTerm was created to make the three tier architecture as shown in the C# code snippet:

```
////////////////////////////////////////////////////////////////////////////////
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace NosComTerm
{
    /// <summary>
    class Program
    ///
    /// </summary>
    {
        static void Main(string[] args)
        {
            ProgramSetupConsoleTerm rProgramSetupConsoleTerm;

            rProgramSetupConsoleTerm = new ProgramSetupConsoleTerm();
        }
    }
    /// <summary>
    class ProgramSetupConsoleTerm
    ///
    /// </summary>
    {
        LayerData rLayerData;
        LayerBusiness rLayerBusiness;
        LayerUser rLayerUser;

        public ProgramSetupConsoleTerm()
        {
            /// include a try/catch block?
            rLayerData = new LayerData();
            rLayerBusiness = new LayerBusiness(rLayerData);
            rLayerUser = new LayerUser(rLayerBusiness);
            rLayerUser.MainLoop();
        }
```

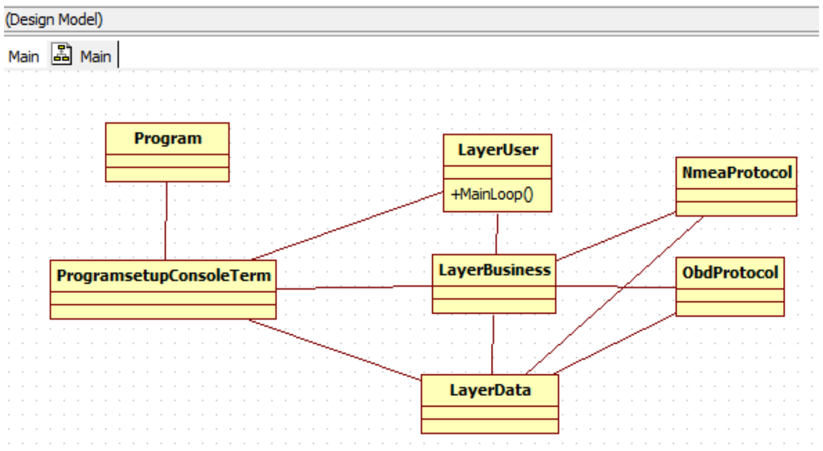


Figure 1.14: The main classes of the terminal application with a three tier architecture.

```
    }
}
///////////////////////////////////////// EOC /////////////////////////////////
```

The ProgramSetupConsoleTerm class contains only the constructor running when the object was created. The object creates the tier classes, starts with the LayerData tier, then the LayerBusiness tier including a reference to the LayerData object, then the LayerUser tier object including the reference to the LayerBusiness object, and finally starts the MainLoop() method in the LayerUser object. The programming code standard used for development defines that any objects should start with the 'r' character to indicate a reference. The MainLoop() method will display the menu and wait for any input selection by the user.

A try/catch block could have been included as the new statements are system operation and it is a good advice to have a general try/catch block on a "high level" to catch any errors not catched by other try/catch blocks.

The class diagram for the application is shown in Figure 1.14 where the main classes are included. Two classes to handle the NMEA and OBD protocol are included by the LayerBusiness class. Normally a separate class for handling the SerialPort should have been included by the LayerData class, but the SerialPort methods are included in the LayerData to simplifying the application.

The main menu has selections for handling either the NMEA or the OBD protocol that will call an appropriate method in the LayerBusiness class. The method in the LayerBusiness class will call either the method in the NmeaProtocol or in the ObdProtocol class depending on the selected protocol, which will use the serial port methods in the LayerData class.

1.3.3 Serial port library

The serial port library has a set of predefined methods that can be used for handling any serial port on the computer. According to Figure 1.13, the use case diagram, the first option should be to get a list of available serial port, select one of the available serial port(s), and the second option should be to open the serial port with the right communication parameters as baudrate, data bits, parity and stop bits before handling any protocol information.

Available serial ports

The serial port library has a method that will return a string array with the available serial ports in the system. The source code snippet for a method getting the available serial ports in your system:

```
  /// <summary>
  private string[] GetListOfAvailableComPorts()
  /// Purpose: fill a string array with available COM ports, null if no com ports.
  /// Version: 1.0: 26-JAN-20: NOS
```

```
/// </summary>
{
    string[] saComPortNames;

    try
    {
        saComPortNames = SerialPort.GetPortNames();
    }
    catch
    {
        saComPortNames = null;
    }
    return saComPortNames;
}
```

The method will return the string array filled with the available serial ports, and saComPort-Names.Length will define the number of serial ports in the system. Try/catch is used since this is a system command, and the string array will be null if an error.

Open and Close

Select an available serial port from the list, and open the serial port using the serial port name from the list, the baudrate and with 8 data bits, no parity and one stop bit as default. The serial port names in Windows are on the form *COMx* and in Linux on the form /dev/ttyx, where x is a number from 1 to N (Linux from 00 to N-1). The source code snippet for opening the serial port is:

```
/// <summary>
public bool Open(string sPortName, int iBaudrate, out string sOpenMsg)
/// Purpose: open a COM port with selectable baudrate, fixed data bits, parity and stop bits.
/// Version: 1.0: 26-JAN-20: NOS
/// </summary>
{
    if (bOpenPort == true)
    {
        rSerialPort.Close();
    }
    try
    {
        rSerialPort = new SerialPort(sPortName, iBaudrate, Parity.None, 8, StopBits.One);
        rSerialPort.ReadTimeout = 500;
        rSerialPort.WriteTimeout = 500;
        rSerialPort.Open();
        sOpenMsg = "Open <" + sPortName + "> serial port OK!";
        bOpenPort = true;
    }
    catch (Exception e)
    {
        sOpenMsg = "Error open <" + sPortName + "> serial port: " + e.Message;
        bOpenPort = false;
    }
    return bOpenPort;
}
```

The rSerialPort object reference is a global variable for the class as the object reference must be used for any read() and write() methods for serial port. The binary flag bOpenPort is used for defining if a serial port is open or not. This flag can also be used in the Read(), Write() and Close() section to verify that the serial port is already open. So the first operation when opening a serial port is to close any already opened serial port. Then make a new serialPort object with the correct settings of the baudrate, data bits, parity and stop bits. These parameter can also be changed after the serial port is opened. The parameters ReadTimout and WriteTimeout are timeout parameters for letting the serial port driver of the operating system to timeout if no characters are read or written within the timeout period. These timeout parameters are in ms, in this case the timeout parameters are 500 ms. The Read() operation will try to read a specific number of bytes and will not finish unless the number of characters are read or if the timeout parameter is set. The application will wait for ever ("hang") if no valid if the ReadTimout parameter value is not set (used). An ok open message will be returned to the calling method with the binary open flag set to true, if the open() method was ok. An error message with information will be returned to the calling method with the binary open flag set to false if the open() method fails.

The source code snippet for closing the serial port:

```
/// <summary>
public void Close()
/// Purpose: close an open serial port
/// Version: 1.0: 26-JAN-20: NOS
/// </summary>
{
    if (bOpenPort == true)
    {
        rSerialPort.Close();
        bOpenPort = false;
    }
}
```

The binary flag bOpenPort is used to indicate an already open serial port, see the code snippet for open the serial port.

Read and Write

The Read() method will read a number of bytes from the serial port, the parameters of the Read() method are:

public int Read(byte[] buffer, int offset, int count);

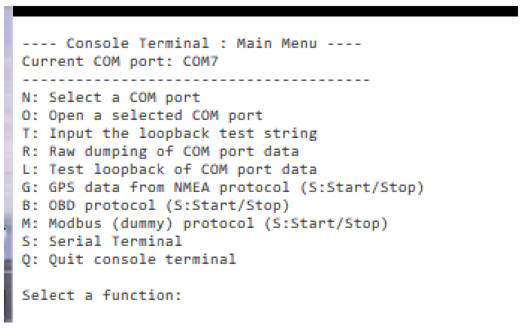
where buffer is a byte array, offset is starting point in the buffer array (normally 0), and count is the number of bytes that you want to read from the serial port. Offset is used if you have read only part of a message and want to read the rest of the message. The Read() method supports several type of parameters, check the C# serial port documentation for more information. The source code snippet for an example of reading from the serial port:

```
/// <summary>
public string Read(int iCntMax, bool bTimeoutMsg)
/// Purpose: read a serial port with a maximum number of bytes, with timeout support.
/// Version: 1.0: 26-JAN-20: NOS
/// </summary>
{
    int iLen, iMsgCnt, iOffset, iMaxLoop;
    string sMsgBuf;
    byte[] bMsgBuf;

    iLen = 0;
    sMsgBuf = "";
    try
    {
        bMsgBuf = new byte[iCntMax + 16];
        iOffset = 0;
```

```
            try
            {
                iMaxLoop = 0;
                while (iOffset < iCntMax && iMaxLoop < 64)
                {
                    iLen = rSerialPort.Read(bMsgBuf, iOffset, (bMsgBuf.Length - iOffset));
                    iOffset += iLen;
                    iMaxLoop++;
                }
                for (iMsgCnt = 0; iMsgCnt < iOffset; iMsgCnt++)
                {
                    sMsgBuf = sMsgBuf + Convert.ToChar(bMsgBuf[iMsgCnt]);
                }
            }
            catch (TimeoutException)
            {
                // Do nothing
                if (iOffset > 0)
                {
                    for (iMsgCnt = 0; iMsgCnt < iOffset; iMsgCnt++)
                    {
                        sMsgBuf = sMsgBuf + Convert.ToChar(bMsgBuf[iMsgCnt]);
                    }
                    if (bTimeoutMsg == true)
                    {
                        sMsgBuf = sMsgBuf + "<Timeout>";
                    }
                }
            }
            catch (Exception e)
            {
                // Any other exceptions
                sMsgBuf = sMsgBuf + "<Exception=" + e.Message + ">";
            }
        }
        catch (Exception e)
        {
            sMsgBuf = sMsgBuf + "<Serial port error=" + e.Message + ">";
        }
        return sMsgBuf;
    }
```

The source code snippet for an example of writing to the serial port:

```
    /// <summary>
    public bool Write(string sBuf)
    /// Purpose: write a string buffer to the serial port
    /// Version: 1.0: 26-JAN-20: NOS
    /// </summary>
    {
        bool bWriteOk;

        try
        {
            rSerialPort.Write(sBuf);
            bWriteOk = true;
        }
```



```
---- Console Terminal : Main Menu ----
Current COM port: COM7
--------------------------------------
N: Select a COM port
O: Open a selected COM port
T: Input the loopback test string
R: Raw dumping of COM port data
L: Test loopback of COM port data
G: GPS data from NMEA protocol (S:Start/Stop)
B: OBD protocol (S:Start/Stop)
M: Modbus (dummy) protocol (S:Start/Stop)
S: Serial Terminal
Q: Quit console terminal

Select a function:
```

Figure 1.15: The output from the MainMenu() method in the application, showing the options that can be selected.

```
        catch (Exception e)
        {
            sBuf = "Error write to the serial port: " + e.Message;
            bWriteOk = false;
        }
        return bWriteOk;
    }
```

1.3.4 Terminal console application

A console application was developed in C# for testing out the serial port, and NMEA and OBD protocols are two of the protocols that are included. A console application is useful for making a test version and the developed application is developed for testing different type of serial port operations. These operations can be to test the hardware of software of the computer by loopback testing, just dumping all the information transmitted and received regardless of any protocols, a general terminal application able to transmit commands, and test out different types of data communication protocols. The application was developed according to the class diagram shown in Figure 1.14.

Menu selections

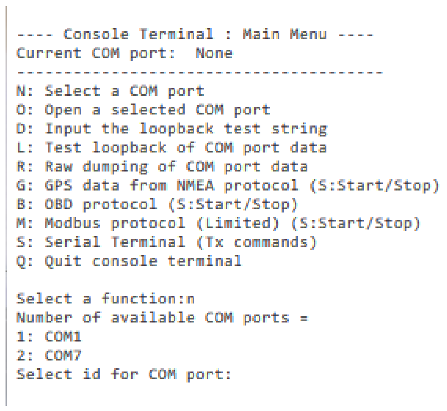
The output from the MainLoop() method is shown in Figure 1.15, showing the selectable options in the application.

The following options are available in this terminal console application, but not all of these options will be explained in this section:

Key	Description	Explained
N	Select a serial port, based on available ports in the system	X
O	Open the selected serial port, choosing the right baud rate	X
D	Input the test string used for the loopback testing of the port	
L	Loopback testing of the selected serial port	X
R	Dumping of raw data from the serial port	(X)
G	Display GPS data from the NMEA protocol (S: Start/Stop button)	(X)
B	Display OBD data from the OBD protocol (S:Start/Stop button)	
M	Display Modbus data from the Modbus protocol (S:Start/Stop button)	
S	Generic serial port terminal, transmit commands and receive data	
Q	Quit the terminal console application.	

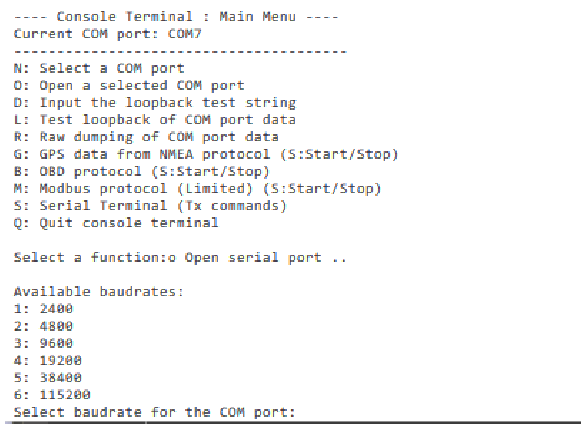
The first option should be to select a serial port. In this case the application is running on a Windows system so the serial port names will be COMx where x is a number from 1 to N, depending on the number of available serial ports. The user must use the ‘N’ button to make this selection, and the output from the application is shown in Figure 1.16.

On top of the screen is the current serial port selected, in this case None. At bottom is a list of avaialble serial ports, COM1 and COM7, and the ’2’ button is used to choose serial port COM7. The



```
---- Console Terminal : Main Menu ----
Current COM port:  None
---------------------------------------
N: Select a COM port
O: Open a selected COM port
D: Input the loopback test string
L: Test loopback of COM port data
R: Raw dumping of COM port data
G: GPS data from NMEA protocol (S:Start/Stop)
B: OBD protocol (S:Start/Stop)
M: Modbus protocol (Limited) (S:Start/Stop)
S: Serial Terminal (Tx commands)
Q: Quit console terminal

Select a function:n
Number of available COM ports =
1: COM1
2: COM7
Select id for COM port:
```

Figure 1.16: The 'N' option in the software for selecting a serial port. Available serial ports in this system is COM1 and COM7.



```
---- Console Terminal : Main Menu ----
Current COM port: COM7
---------------------------------------
N: Select a COM port
O: Open a selected COM port
D: Input the loopback test string
L: Test loopback of COM port data
R: Raw dumping of COM port data
G: GPS data from NMEA protocol (S:Start/Stop)
B: OBD protocol (S:Start/Stop)
M: Modbus protocol (Limited) (S:Start/Stop)
S: Serial Terminal (Tx commands)
Q: Quit console terminal

Select a function:o Open serial port ..

Available baudrates:
1: 2400
2: 4800
3: 9600
4: 19200
5: 38400
6: 115200
Select baudrate for the COM port:
```

Figure 1.17: The selection of a baud rate for open the COM7 serial port.

next operation is to open this serial port with the appropriate baud rate like the snippet code shown for the Open() method. It is of course possible to combine the selection of a serial port and open a serial port in the same menu selection if wanted. The menu option 'O' will give the output as shown in Figure 1.17 with the selected serial port on the top, and a list of available baud rates at the bottom.

Use the button for the number in front of the baud rate to select the valid baudrate, and the respons shown in Figure 1.18 will be the results. In this case the serial port COM7 is now opened ok and can be used by the application. An error will be displayed if the open() statement fails, as shown in the open() source code snippet.

The port is now open with the selected baud rate and the serial port can now be used for data communication.

Loopback testing

The first time using a serial port, a loopback testing can be useful to check that both the hardware and software is working. Loopback testing means that an interconnection between the transmitter pin and the received pin for the serial port must be inserted, and all the data transmitted for serial port should be received by the application. In this way both the transmitter and receiving section of the application, and the serial port will be testing. Figure 1.19 shows a serial port connector with a loopback test connector (to the left, Rx and Tx pins connected) and the NMEA protocol connector (in front, only Rx and Gnd pins connected).

The loopback connector is inserted and the 'D' option is selected to make the loopback test string to be "SerialPortTesting". Then the 'L' option is selected and the output from the application is shown in Figure 1.20. The application will add a counter id after each time reading the loopback test string, so the output should be "SerialPortTestingx" where x is a number from 1 and increasing. The read opeation is defined to read 16 characters from the serial port for each operation, the reason for the figure to display

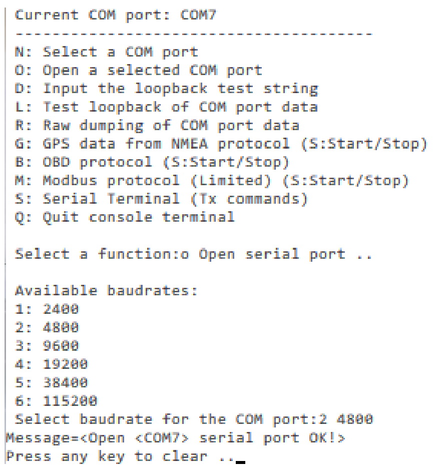

```
Current COM port: COM7
---------------------------------------
N: Select a COM port
O: Open a selected COM port
D: Input the loopback test string
L: Test loopback of COM port data
R: Raw dumping of COM port data
G: GPS data from NMEA protocol (S:Start/Stop)
B: OBD protocol (S:Start/Stop)
M: Modbus protocol (Limited) (S:Start/Stop)
S: Serial Terminal (Tx commands)
Q: Quit console terminal

Select a function:o Open serial port ..

Available baudrates:
1: 2400
2: 4800
3: 9600
4: 19200
5: 38400
6: 115200
Select baudrate for the COM port:2 4800
Message=<Open <COM7> serial port OK!>
Press any key to clear ..
```

Figure 1.18: The '2' button is used to select 4800 as the baud rate, and the serial port is now open OK. The COM7 serial port can now be used by the application.

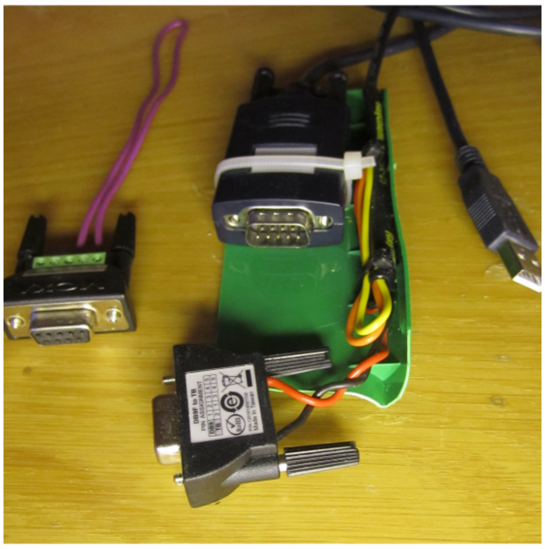
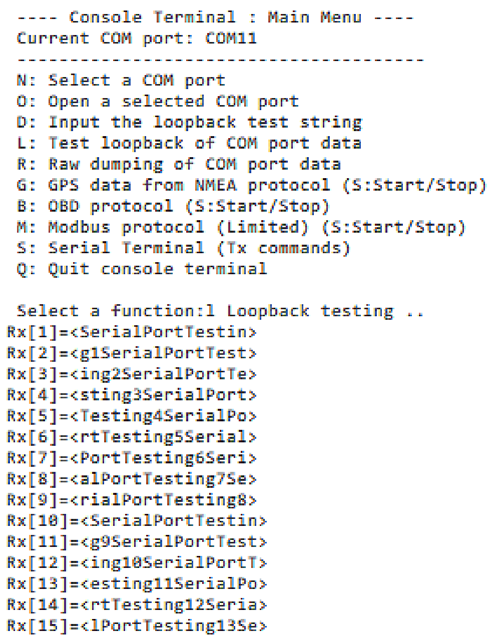


Figure 1.19: A serial port with a loopback serial port connector that can be inserted to test the serial port. The loopback connection is just interconnecting pin 2 and pin 3 of the serial port, the transmitter and receiver pins.



```
---- Console Terminal : Main Menu ----
Current COM port: COM11
--------------------------------------
N: Select a COM port
O: Open a selected COM port
D: Input the loopback test string
L: Test loopback of COM port data
R: Raw dumping of COM port data
G: GPS data from NMEA protocol (S:Start/Stop)
B: OBD protocol (S:Start/Stop)
M: Modbus protocol (Limited) (S:Start/Stop)
S: Serial Terminal (Tx commands)
Q: Quit console terminal

Select a function:l Loopback testing ..
Rx[1]=<SerialPortTestin>
Rx[2]=<g1SerialPortTest>
Rx[3]=<ing2SerialPortTe>
Rx[4]=<sting3SerialPort>
Rx[5]=<Testing4SerialPo>
Rx[6]=<rtTesting5Serial>
Rx[7]=<PortTesting6Seri>
Rx[8]=<alPortTesting7Se>
Rx[9]=<rialPortTesting8>
Rx[10]=<SerialPortTestin>
Rx[11]=<g9SerialPortTest>
Rx[12]=<ing10SerialPortT>
Rx[13]=<esting11SerialPo>
Rx[14]=<rtTesting12Seria>
Rx[15]=<lPortTesting13Se>
```

Figure 1.20: The console output from the loopback testing option.

only 16 characters on each line. However, all the characters are read so the serial port and the application is working. The screen update will stop when the loopback test connector is disconnected, or use the ‘Q’ button to stop this operation. This output also shows the challenges on receiving information on a serial port and merge the information into a valid structure. The receiving method may stop the reading at any time and position of the receiving information, depending on the impementation of reading the serial port. If reading too few characters, the application need logic to merge these characters into a valid message. If reading too many characters, the application will timeout and use more time in collecting the information. Also, reading too many characters may end up reading several messages at the same time, as shown in Figure 1.8.

Dumping of raw data

This can be a very useful option just to dump the all the caracters received from the serial port. Convert the byte[] buffer from the read() method to a string, and print the string on the console. This is shown in Figure 1.8 but remember that this requires an ASCII protocol with mainly printable characters. Any non-printable characters must be converted to printable characters like 0x08, 0x0a and 0x0d can be converted to <TAB>, <LF> and <CR>. For a binary protocol, like Modbus RTU, the information must be converted to printable characters before this option can be used.

Implement NMEA protocol

A console application can be a good tools to test out the first version of the protocol coding and decoding, to get some experience how to merge and extract the protocol information. Figure 1.8 shows the raw data for the NMEA protocol showing the structure of the message, and Figure 1.9 show the extraction of some of the information from the NMEA command GPGGA. The list in section 1.2.4 shows a sequence of OBD messages, with timestamps, both transmitted and received, for the OBD protocol. Both of these protocols are ASCII protocols meaning that the information consists of ASCII characters, where many of the ASCII characters are printable characters. The decoding of such protocols will be to use different string operations in C#. The general structure is 1) to find the start of the message, 2) find the end of the message to be sure that the whole message has been received, 3) any error checking if checksum is available, 4) subtract the different fields of the message depending on the field separator, and 5) decode the information wanted from these fields.

A source code snippet for one way of extracting information from a NMEA message. Just part of the method is included to get an idea of one possible way of extracting such information:

```
/// <summary>
int decodeNmeaMsg(string sNmeaMsg, out string sErrMsg)
/// Purpose: decode a NMEA message
/// Version: 1.0: 29-JAN-20: NOS
/// </summary>
{
    string[] sNmeaFields;
    string sNmeaCmd, sNmeaChksum;
    double dGpsTime, dLatitude, dLongitude;
    int iGpsQuality;

    /// Spilt the fields based on comma and star
    sNmeaFields = sNmeaMsg.Split(',','*');
    sErrMsg = "";
    /// The last field is the checksum, after the star
    sNmeaChksum = sNmeaFields[sNmeaFields.Length - 1];
    /// No test of the checksum - must be included in the final solution!
    /// Get the NMEA commands
    if (sNmeaFields[NMEA_CMD_OFFSET].Length > 2)
    {
        /// Remove the GP part of the message command
        sNmeaCmd = sNmeaFields[NMEA_CMD_OFFSET].Substring(2);
        if (string.Equals(sNmeaCmd, "GGA", StringComparison.OrdinalIgnoreCase) == true)
        {
            try
            {
                dGpsTime = Convert.ToDouble(sNmeaFields[NMEA_GGA_CMD_GPS_TIME_OFFSET]);
```

The message format of any protocol is important to understand when implementing such a protocol and a console application can be a very good way of testing such an implementation before integrating the protocol support into the final application.

1.4 Network communication programming in C#

1.4.1 Introduction

Data communication is important for communication between computers, both on local area networks (LAN)(intranet) and on Internet. The previous chapter looked into serial port communication, using serial ports for external communication. In this chapter will the focus be network data communication, meaning data communication over a network. Serial port data communication has a limited number of transmitters and receivers on the same communication media, and normally the communication media has also a limited length as well. The maximum length is also often depending on the baud rate for serial data communication. Network data communication is using a more complex structure with hardware devices like switches, routers, gateways and firewalls to link network system together, route the messages between these network systems, and using firewalls to protect for which messages that can be received by each sub network.

This means that a network protocol will be more complex than a serial port protocol, however the OSI model can also be used to structure the functionality of these protocols. The serial port protocols depends mainly on the lowest layer in the OSI model, while network protocols will depend on several layers in this protocol model. Still, network communication must still depend on some sort of serial data communication, like the RS-485 protocol used in many network protocols, as the physical layer. The physical layer can still use either wired, optical or wireless physical connection for data communication.

1.4.2 OSI model

The OSI model is an abstract model used to structure any communication protocol into the functionality of seven layers, as shown on the left side in Figure 1.21. The serial port communication described in the

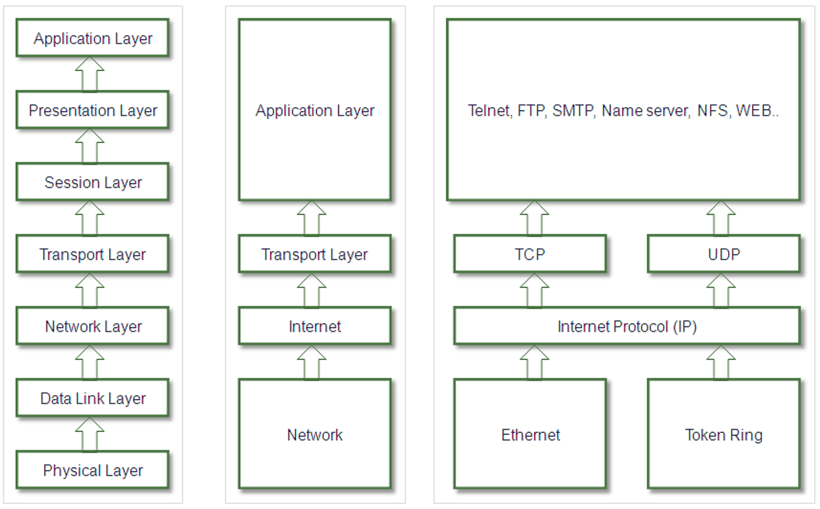


Figure 1.21: The OSI model, the general model to the left, the Internet structure in the middle, and the TCP/IP solution to the right.

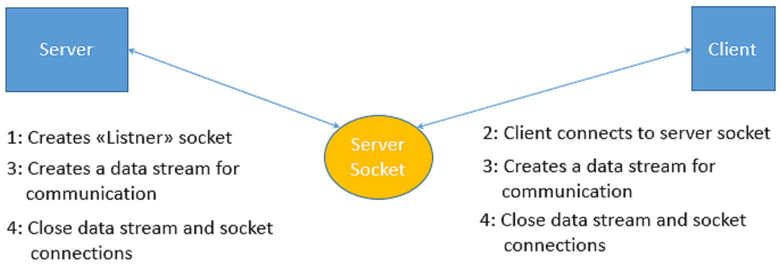

Server
Client
Server Socket
1: Creates «Listner» socket
3: Creates a data stream for communication
4: Close data stream and socket connections
2: Client connects to server socket
3: Creates a data stream for communication
4: Close data stream and socket connections

Figure 1.22: The socket connection steps for communication for server and client objects in C#.

previous chapter describes the functionality in the physical layer of the OSI model, while the network data communication is described by the functionality of the first four layers, physical, data link, network and transport layers.

This chapter will not focus on the OSI model(Stevens 1990)(Stallings 2001), but on the TCP/IP protocol (Comer 1991) (Stevens 1990) (Crowcroft & Phillips 2002) (Bentham 2002) used as one protocol for the transport layer and corresponding support from C# libraries. The structure of the TCP/IP protocol is indicated to the right in Figure 1.21. The figure shows that the TCP/IP protocol is using the functionality of the Internet Protocol (IP) for communicating with the nodes on the network, where the main functionality is logical addressing. A node in one local area network (LAN) connected to the Internet can communicate with another node connected to another LAN on the Internet using an IP protocol. The IP protocol will use either the Ethernet or Token Ring protocols for handling the functionality of the two lowest layers in the OSI model, as shown to the right in the figure. The Ethernet protocol is the most used protocol today for handling the Internet communication on these lowest layers.

1.4.3 TCP/IP protocol

The TCP/IP protocol is a network protocol and is based on the client/server concept. This concept define there must be an existing server with some sort of information that a client wants get access to. In data communication a socket is used to establish a connection between two or more nodes (computer with communication capabilities), and these sockets will be used to send data between the nodes. For the TCP/IP protocol will the server first create a listener socket, and any client will create a socket for connecting to the listener socket of a specific server. For the TCP/IP protocol will the IP addresses be used by the both the server and clients to create the socket for communicating, so the IP address of the server should be known. When the socket connection is established, the nodes can exchange information using the socket connection (read and write methods). Figure 1.22 shows the socket connection where the server first make a listener socket and waits for any client socket to connect to that socket (step 2).

When the connection is fulfilled will both systems make a data stream for exchanging information, by reading and writing. Note that this will be a multitasking system as these to applications will run independent of each other. Ending the connection by closing the data stream and socket connection at both ends.

1.4.4 Socket programming in C#

Socket programming (Stevens 1990) (Crowcroft & Phillips 2002) (Bentham 2002) (Mahmoud 2004) is based on a bidirectional communication link between two applications, a server application and a client application, see Figure 1.22. Socket programming is using the IP address so the TCP/IP protocol needs to be implemented in the system, but most nodes today have the TCP/IP protocol included. The C# socket library is included in the System.Net and System.Net.Sockets namespaces, so these two libraries must be included in your application, as shown in the following list::

```
using System;
using System.Text;
using System.Net.Sockets;
using System.Net;
```

The first step is to get the IP address of the, either the running node or the remote node. Use the following commands to get a list of the available IP addresses for the node:

```
ipHostInfo = Dns.GetHostEntry(Dns.GetHostName());
ipAddress = ipHostInfo.AddressList[0];
```

The first line gets the information from the Domain Name System, and the information is using the name of this node. The second line gets a list of available IP addresses for this node. The reason for several IP addresses is that the system has support for both IPv4 and IP v6 protocols with a different type of IP addresses. In this case is the first IP address used.

For the server will the next steps be:

```
tcpListener = new TcpListener(ipAddress, iSocketPortId);
tcpListener.Start(5);
tcpClient = tcpListener.AcceptTcpClient();
tcpNetworkStream = tcpClient.GetStream();
```

First make a listener socket that is used for a client to connect, and start this listener socket. In this case it allows for up to five clients to connect at the same time. The AcceptTcpClient() command will wait (block) until a client is connecting to this socket, with the correct IP address and socket port number. Accepting the client connection, a network data stream is created to allow for reading and writing data between the server and client.

The commands will close the connected after finishing the communication:

```
tcpNetworkStream.Close();
tcpClient.Close();
```

For the client will the next steps be:

```
tcpClient = new TcpClient(ipAddress.ToString(), iSocketPortId);
tcpNetworkStream = tcpClient.GetStream();
```

The TcpClient() command will connect to an existing socket connection, with the correct IP address and socket port number. Accepting the server connection, a network data stream is created to allow for reading and writing data between the server and client.

The network data stream can now be used for transmitting and receiving bytes using the Read() and Write() methods for the data stream references. The Read() method will read a number of bytes, and return the number of bytes read. The Read() method can also be extended with the buffer offset and maximum number of bytes to read. The Read() method reading a number of bytes and converting to a printable string:

```
byte[] baBuffer = new byte[64];
int iLen = tcpNetworkStream.Read(baBuffer);
String sBuffer = Encoding.ASCII.GetString(baBuffer);
sBuffer = sBuffer.TrimEnd('\0');
```

The Write() method will write a number of bytes, no return. The Write() method can also be extended with the buffer offset and maximum number of bytes to write. The Write() method making a string, converting the string into a byte array, and write the byte array:

```
string sBuffer = DateTime.Now.ToString() + ": Server command=<" + sBuffer + ">";
byte[] baBuffer = Encoding.ASCII.GetBytes(sBuffer);
tcpNetworkStream.Write(baBuffer);
```

The commands will close the connected after finishing the communication:

```
tcpNetworkStream.Close();
tcpClient.Close();
```

Note that these socket methods are system methods and try/catch blocks must be used to make the implementation more robust. The Read() and Write() methods handling only a byte array is the challenge for TCP/IP data communication as no standard protocol exists for handling the data structure in these byte arrays. If your application is receiving the byte stream $160A326790B386D2B4A8324447$ you will not have any information about this information unless informed by the sender application system. This is also one of the reason for many proprietary protocols for Internet of Thing (IoT) systems, using the TCP/IP protocol do NOT define any data structure of the byte buffers. The solution will often be to define a new protocol on top of the TCP/IP protocol for handling the application layer of the data communication, see Figure 1.21.

These socket methods will be used in a console test application to show the implementation of socket communication, and a simple protocol will be defined on top of the TCP/IP protocol to handle the messages between the socket server and client.

1.4.5 Console test application

A simple console application will be developed to test out the concept of client/server programming using sockets in C#. The application will also show the challenges with the TCP/IP protocol as this protocol is just defining a data buffer for exchanging information, no interpreting of the information in the data buffer. A very simple protocol has been define, two string commands, for communication between the server and the client. These commands are string based, “LIST” and “Quit”; “List” will display the IP address(es) of the server node and “Quit” will terminate the server/client connection. The TCP/IP protocol is also used for IoT and IIoT devices and these devices will have the same challenge with the TCP/IP protocol, and is the reason for many different IoT protocols, both proprietary and open protocols. These protocols will define the interpretation of the TCP/IP buffer, and support the three upper layers in the OSI model, normally only the “Application layer”. The “presentation layer” contains any security functionality and should be integrated into any of these protocols.

The first task when designing a new software application is to make a use case diagram, and this diagram is shown in Figure 1.23.

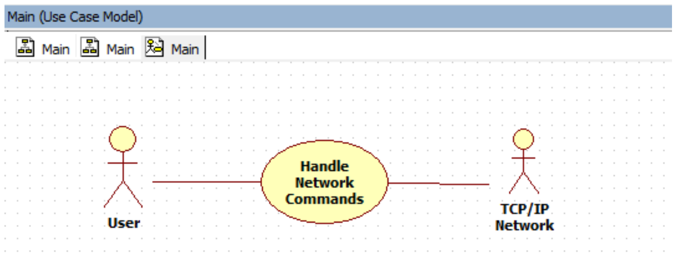


Figure 1.23: The use case diagram for the console socket test application, only one use case for testing commands between a local socket server and socket client.

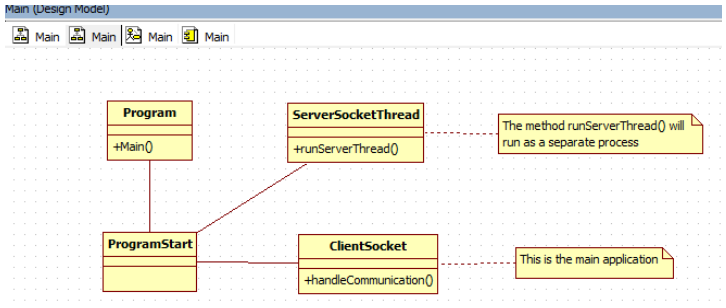


Figure 1.24: The one tier architecture shown in a class diagram, for the network test application.

Since this is a only an application for testing/showing the socket functions in C# for network communication, only an one tier application is developed. If the application is going to be extended with new functionality, a two or three tier application should have been developed. The console application consists of four classes, as shown in the class diagram in Figure 1.24.

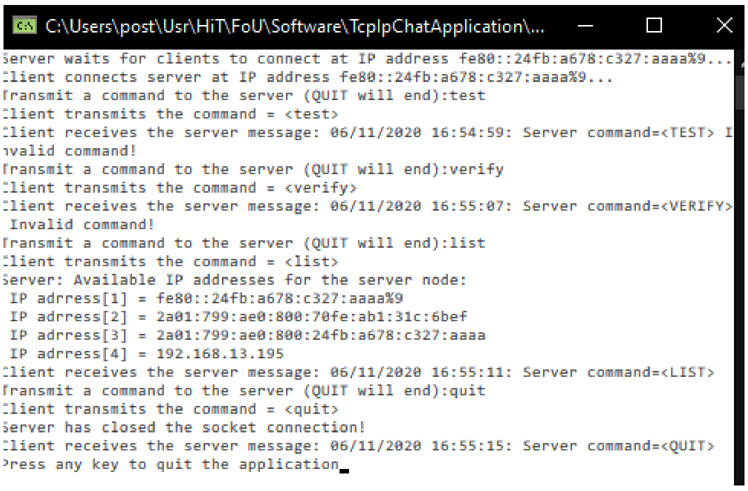
The main method of the server class (*RunServerThread()*) will run as a separate process, making this a multitasking system. The main method in the client class (*HandleCommunication()*) will contain the functionality of the client, with interface to the user selecting any commands. Program class is generated by the Visual Studio IDE, the ProgramStart class is added to make an object, and this object will first start the server thread as part of the server object, and then the main loop in the client object. The output from this multitasking system is shown in Figure 1.25.

The user first send the command “test” to the server, and the server respond with an invalid command message. Then the user tries the “verify” command with the same response from the socket server. The next command is “list”, listing the IP addresses of the server, and finally the “quit” command first terminating the socket connection, then the server process and then the client application. Note that for simplicity will the server write the list messages to the console, the best solution is however for the server to send this information as text messages to the client, and for the client to display this information on the console. Why is this a better solution?

The simple protocol implemented in the application is the two text commands “*LIST*” and “*QUIT*” checked for in the server thread. Also note that the information is sent as plain text on the socket stream meaning that the information can be seen by anybody with access to the communication media and the TCP/IP protocol. Part of a data security policy can be to crypt/decrypt the byte buffers when transmitting and receiving the data. Wireshark[3] is one of the free applications that can be used to visualize data from external TCP/IP messages. The source code for the network console test application:

```
 ///////////////////////////////////////////////////////////////////////////////////
///
/// Purpose: make a console application showing the usage of TCP/IP socket programming,
///          one thread running as a server, and the application running as a client
/// Version: 19-OCT-20: NOS
////////////////////////////////////////////////////////////////////////////////////
using System;
```

[3]Wireshark, a free and open-source network data packet analyzer, www.wireshark.org, Nov-19.



```
erver waits for clients to connect at IP address fe80::24fb:a678:c327:aaaa%9...
lient connects server at IP address fe80::24fb:a678:c327:aaaa%9...
ransmit a command to the server (QUIT will end):test
lient transmits the command = <test>
lient receives the server message: 06/11/2020 16:54:59: Server command=<TEST> I
valid command!
ransmit a command to the server (QUIT will end):verify
lient transmits the command = <verify>
lient receives the server message: 06/11/2020 16:55:07: Server command=<VERIFY>
 Invalid command!
ransmit a command to the server (QUIT will end):list
lient transmits the command = <list>
erver: Available IP addresses for the server node:
 IP adrress[1] = fe80::24fb:a678:c327:aaaa%9
 IP adrress[2] = 2a01:799:ae0:800:70fe:ab1:31c:6bef
 IP adrress[3] = 2a01:799:ae0:800:24fb:a678:c327:aaaa
 IP adrress[4] = 192.168.13.195
lient receives the server message: 06/11/2020 16:55:11: Server command=<LIST>
ransmit a command to the server (QUIT will end):quit
lient transmits the command = <quit>
erver has closed the socket connection!
lient receives the server message: 06/11/2020 16:55:15: Server command=<QUIT>
ress any key to quit the application_
```

Figure 1.25: The output for the network test application, the socket server running as a background prosess (thread) and the socket client running as the foreground application with user interaction.

```
using System.Text;
using System.Net.Sockets;
using System.Net;
using System.Threading;
///
namespace NosConsoleSocket
{
    ////////////////////////////////////////////////////////////////////////////////////////
    class Program
    {
        /// <summary>
        static void Main()
        /// Purpose: The main entry point for the application.
        /// Version: 1.0: 20-OCT-20: NOS
        /// </summary>
        {
            /// Convert from a static to a dynamic object
            new ProgramStart();
        }

    ////////////////////////////////////////////////////////////////////////////////////////
    class ProgramStart
    {
        const int iSocketPortNumber = 9999;
        ServerSocketThread rServerSocketThread;
        ClientSocket rClientSocket;
        /// <summary>
        public ProgramStart()
        ///
        /// Purpose: the startup object for handling the server thread and client
        /// Version: 21-OCT-20: NOS
        /// </summary>
        {
            try
            {
                /// start the socket server first as a separate thread
                rServerSocketThread = new ServerSocketThread(iSocketPortNumber);
                /// Wait to let the server create the connection point for clients
                Thread.Sleep(500);
                /// start the socket client to communicate with the server
                rClientSocket = new ClientSocket(iSocketPortNumber);
                rClientSocket.handleCommunication();
            }
            catch (exception ex)
            {
                Console.WriteLine(" Application error: " + ex.Message());
            }
        }
    }
    ////////////////////////////////////////////////////////////////////////////////////
    class ServerSocketThread
    {
        int iSocketPortId;
        bool bServerThreadFlag;
        Thread thServerSocket;
        /// <summary>
        public ServerSocketThread(int iSPN)
        ///
        /// Purpose: constructor making the server thread using sockets
        /// Version: 22-OCT-20: NOS
        /// </summary>
        {
            iSocketPortId = iSPN;
            bServerThreadFlag = true;
            thServerSocket = new Thread(new ThreadStart(this.runServerThread));
            thServerSocket.Start();
        }
        /// <summary>
        private void runServerThread()
        ///
        /// Purpose: the server thread running as a separate process, handling the server
        ///          functions.
        /// Version: 22-OCT-20: NOS
        /// </summary>
        {
            IPHostEntry ipHostInfo;
            IPAddress ipAddress;
            TcpListener tcpListener;
            TcpClient tcpClient;
            NetworkStream tcpNetworkStream;
            byte[] baBuffer;
            string sBuffer;
            int iCnt;
            bool bCmdValid;

            try
            {
                /// get the IP address information this node
                ipHostInfo = Dns.GetHostEntry(Dns.GetHostName());
                ipAddress = ipHostInfo.AddressList[0];
                /// make a TCP connection point based on the IP address and the port number
                tcpListener = new TcpListener(ipAddress, iSocketPortId);
                /// Start the listner connected, allow up to five clients at the same time
                tcpListener.Start(5);
                /// wait for a client to connect to this conenction point (port ID)
                Console.WriteLine("Server waits for clients to connect at IP address " +
                            ipAddress.ToString() + "...");
                tcpClient = tcpListener.AcceptTcpClient();
                /// a client has connected, make a data stream for communicating with the client
                tcpNetworkStream = tcpClient.GetStream();
                /// loop for communicating with the client, let the client close the connection.
                while (bServerThreadFlag == true)
                {
                    /// wait for receiving a message from the client
                    baBuffer = new byte[64];
                    tcpNetworkStream.Read(baBuffer);
                    sBuffer = Encoding.ASCII.GetString(baBuffer);
                    sBuffer = sBuffer.TrimEnd('\0');
                    /// check for a valid message from the client
                    /// Note that the TCP/IP protocol just define a byte data buffer, no data
                    /// structure. You must define your own protocol based on this buffer. The same
                    /// for IoT communication, often defining a propritary protocol on top of TCP/IP
                    /// Define a simple protocol, based on string commands; LIST and QUIT.
                    /// LIST shows the IP addresses for the server and QUIT terminates the server
                    /// connection.
                    sBuffer = sBuffer.ToUpper();
                    if (string.Equals(sBuffer, "QUIT") == true)
                    {
                        bServerThreadFlag = false;
                        bCmdValid = true;
                    }
                    else if (string.Equals(sBuffer, "LIST") == true)
                    {
                        Console.WriteLine("Server: Available IP addresses for the server node:");
                        for (iCnt = 0; iCnt < ipHostInfo.AddressList.Length; iCnt++)
                        {
                            Console.WriteLine(" IP adrress[" + (iCnt + 1).ToString() + "] = " +
                                ipHostInfo.AddressList[iCnt].ToString());
                        }
                        bCmdValid = true;
                    }
                    else
                    {
                        bCmdValid = false;
                    }
                    /// Send an answer back to the client
                    sBuffer = DateTime.Now.ToString() + ": Server command=<" + sBuffer + ">";
                    if (bCmdValid == false)
                    {
                        sBuffer = sBuffer + " Invalid command!";
                    }
                    baBuffer = Encoding.ASCII.GetBytes(sBuffer);
                    tcpNetworkStream.Write(baBuffer, 0, baBuffer.Length);
                }
                /// The connection is closed, close the data stream and socket.
                tcpNetworkStream.Close();
                tcpClient.Close();
                Console.WriteLine("Server has closed the socket connection!");
            }
            catch (Exception ex)
            {
                /// Display any errors for the server socket thread
                Console.WriteLine("Server error: ", ex.Message);
            }
            }
        }
        ////////////////////////////////////////////////////////////////////////////////
        class ClientSocket
        {
            const int MAX_RX_LEN = 128;
            int iSocketPortId;
            IPHostEntry ipHostInfo;
            IPAddress ipAddress;
            TcpClient tcpClient;
            NetworkStream tcpNetworkStream;
            string sTxBuf, sRxBuf;
            byte[] baBuf;
            /// <summary>
            public ClientSocket(int iSPN)
            ///
            /// Purpose: constructor for the client class
            /// Version 23-OCT-20: NOS
            /// </summary>
            {
                iSocketPortId = iSPN;
            }
            /// <summary>
            public void handleCommunication()
            ///
            /// Purpose: the client application, communicating with the server process
            /// Version: 23-OCT-20: NOS
            /// </summary>
            {
                try
                {
                    /// Get the IP address of this node
                    ipHostInfo = Dns.GetHostEntry(Dns.GetHostName());
                    ipAddress = ipHostInfo.AddressList[0];
                    /// connect to server running on localhost at the specific port number id
                    tcpClient = new TcpClient(ipAddress.ToString(), iSocketPortId);
                    Console.WriteLine("Client connects server at IP address " + ipAddress.ToString() +
                                      "...");
                    /// make the data stream for reading and writing with the server
                    tcpNetworkStream = tcpClient.GetStream();
                    Console.Write("Transmit a command to the server (QUIT will end):");
                    sTxBuf = "";
                    /// handle the simple string based protocol for the server communication
                    while (string.Compare(sTxBuf, "QUIT", true) != 0)
                    {
                        /// get the next command from the user
                        sTxBuf = Console.ReadLine();
                        Console.WriteLine("Client transmits the command = <" + sTxBuf + ">");
                        baBuf = Encoding.ASCII.GetBytes(sTxBuf);
                        /// transmit the command to the server, and wait for any answer
                        tcpNetworkStream.Write(baBuf, 0, baBuf.Length);
                        /// release cpu time for other processes while waiting for server response
                        Thread.Sleep(100);
                        /// Read any answer from the server
                        baBuf = new byte[MAX_RX_LEN];
                        tcpNetworkStream.Read(baBuf, 0, baBuf.Length - 1);
                        sRxBuf = Encoding.ASCII.GetString(baBuf);   // convert byte array to string
                        Console.WriteLine("Client receives the server message: " + sRxBuf);
```

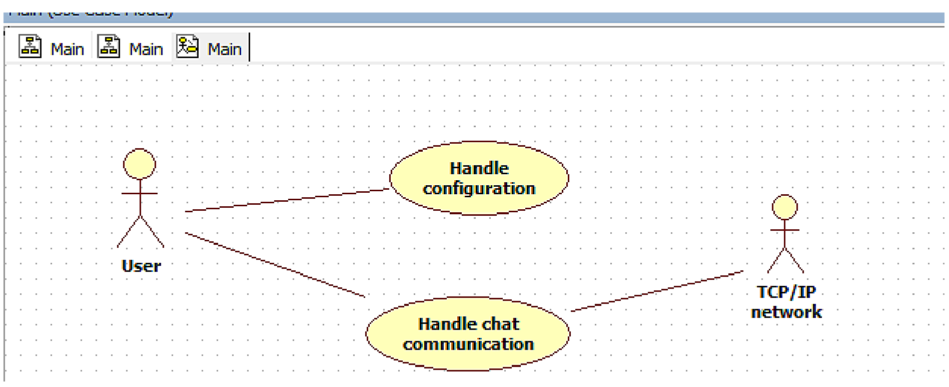


Figure 1.26: The use case diagram for the main functionality of the chat application, with two use cases. One use case for handling configuration and one use case for handling the chat communication.

```
                    if (string.Compare(sTxBuf, "QUIT", true) != 0)
                    {
                        Console.Write("Transmit a command to the server (QUIT will end):");
                    }
                }
                /// Close the socket communication with the server
                tcpNetworkStream.Close();
                tcpClient.Close();
                Console.Write("Press any key to quit the application");
                Console.ReadKey();
            }
            catch (Exception ex)
            {
                /// Handle any client erros
                Console.WriteLine("Client error: ", ex.Message);
            }
        }
    }
}
///
/////////////////////////////////////////// EOC ///////////////////////////////////////////
```

1.4.6 GUI Chat application

The console test application is running both the server and the client on the same node which does not take any advantages of network communication. A Graphical User Interface (GUI) application that can be used as a chat application between two nodes is developed to show full utilization of network communication. With this application two user should be able to communicate (chat) running the same application on two different nodes in a local area network (LAN). It should also be possible to use the application over Internet but requires that the selected port number is available for communication through any firewalls. Note that the usage of this application should trigger your “antivirus/internet security” protection as this application is trying to make a TCP/IP connection on a non-standard port number.

The use case diagram for application is shown in Figure 1.26, showing the main user functions for the application.

The first use case will be handle the configuration of the application (server/client mode, IP address and socket port number) and the second use case for the chat function where the users can chat with each other by writing messages. The connection will be part of the second use case as this will involve both users, where the remote user is shown as the TCP/IP network actor. The structure of the application will be a three tier architecture, one tier for the user interface, one tier for the business logic for handling the communication modes and exchange of messages, and the tier for the data layer handling the socket

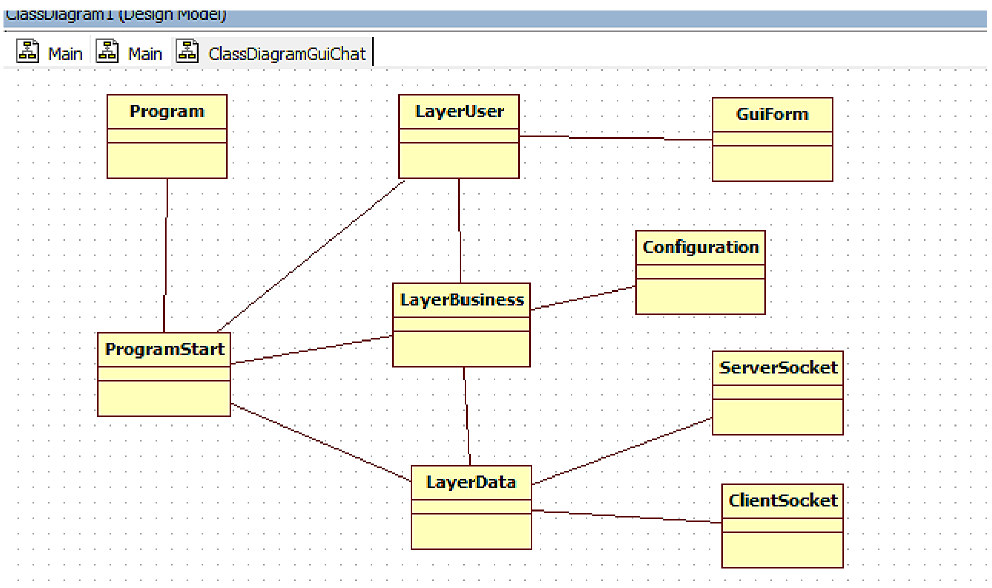


Figure 1.27: The three tier architecture class diagram of the GUI chat application, with the program startup classes, the tier classes, socket mode classes and a class for configuration parameters.

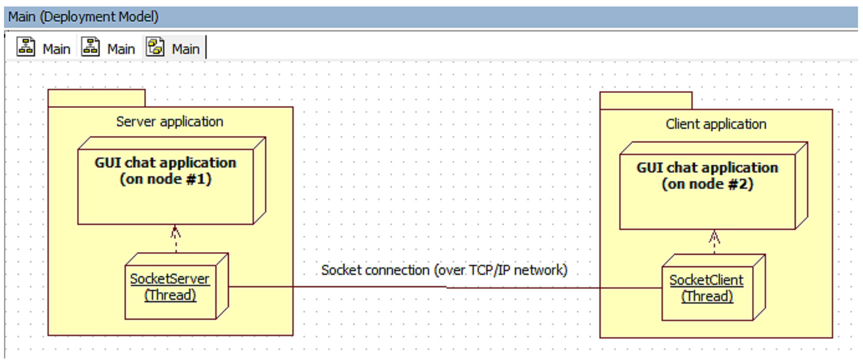


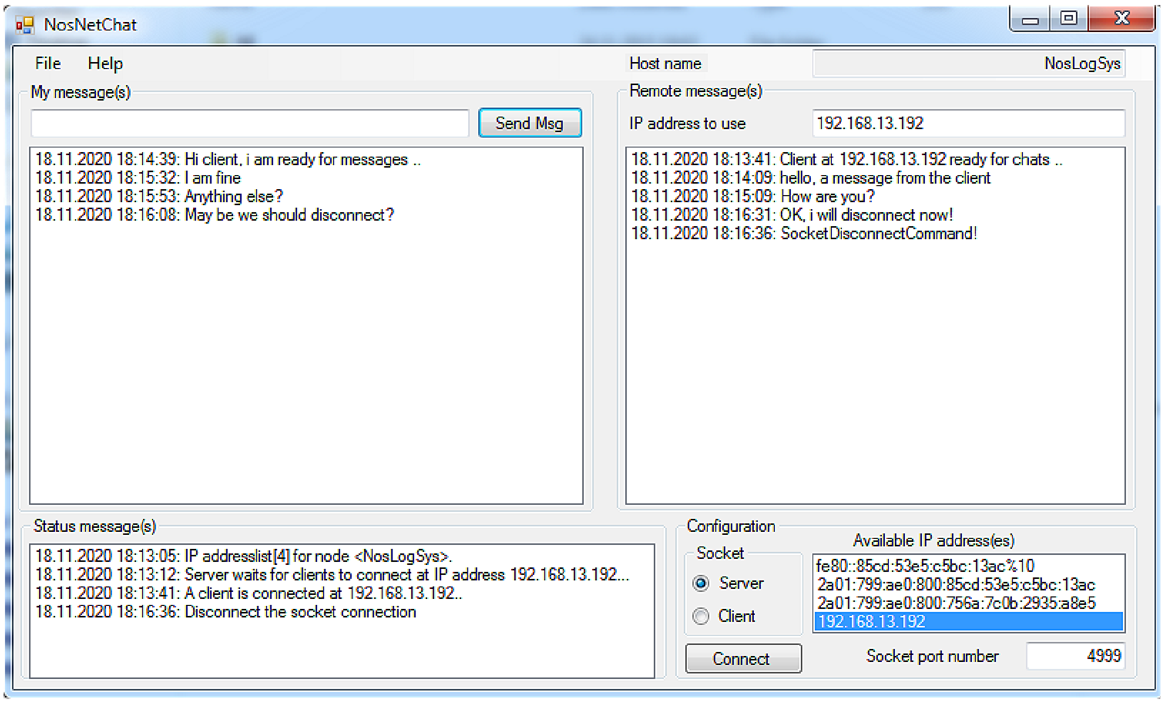
Figure 1.28: The deployment diagram for the application, running two applications. One application as the server, one application as the client, and using a socket connection as the TCP/IP communication.

communication. The data layer can also be extended with a log feature saving the messages on a text file. The class diagram for this structure is shown in Figure 1.27.

The most important classes is shown in this class diagram, two classes (Program and ProgramStart) for starting the application, three classes for handling the tier functionality of the application (LayerUser, LayerBusiness and LayerData), one class for the configuration parameters, and two classes handling the threads for server and client socket communication. Only one of these threads will be running in a specific application, depending on the configuration. A thread is very useful to let the socket communication be a background process independent of any user interaction. Figure 1.28 shows the deployment diagram, showing the two instances of the application running in a network. One application configured as the socket server, with the ServerSocket thread object activated, and one application configured as the socket client, running the ClientSocket thread object.

The C# library and methods are the same as used for the console application, however this application is extended with a configuration section and a three tier architecture. The user interface for the application is shown in Figures 1.30 and 1.29, where Figure 1.30 shows the configuration and some messages for the client application and Figure 1.29 shows the configuration and some messages for the server application.

The user interface consists of five sections, one section with the menu bar and the host name on the top, two message sections with the transmitted messages to the left and the received messages to the right. The section in the lower left corner is status information from the application, in this case the connection and disconnection to any remote nodes. The last section, in the lower right corner, is the



Host name: NosLogSys

My message(s)

```
18.11.2020 18:14:39: Hi client, i am ready for messages ..
18.11.2020 18:15:32: I am fine
18.11.2020 18:15:53: Anything else?
18.11.2020 18:16:08: May be we should disconnect?
```

Remote message(s)

IP address to use: 192.168.13.192

```
18.11.2020 18:13:41: Client at 192.168.13.192 ready for chats ..
18.11.2020 18:14:09: hello, a message from the client
18.11.2020 18:15:09: How are you?
18.11.2020 18:16:31: OK, i will disconnect now!
18.11.2020 18:16:36: SocketDisconnectCommand!
```

Status message(s)

```
18.11.2020 18:13:05: IP addresslist[4] for node <NosLogSys>.
18.11.2020 18:13:12: Server waits for clients to connect at IP address 192.168.13.192...
18.11.2020 18:13:41: A client is connected at 192.168.13.192..
18.11.2020 18:16:36: Disconnect the socket connection
```

Configuration

Socket: [✓ Server] Client

Available IP address(es)

```
fe80::85cd:53e5:c5bc:13ac%10
2a01:799:ae0:800:85cd:53e5:c5bc:13ac
2a01:799:ae0:800:756a:7c0b:2935:a8e5
192.168.13.192
```

Socket port number: 4999

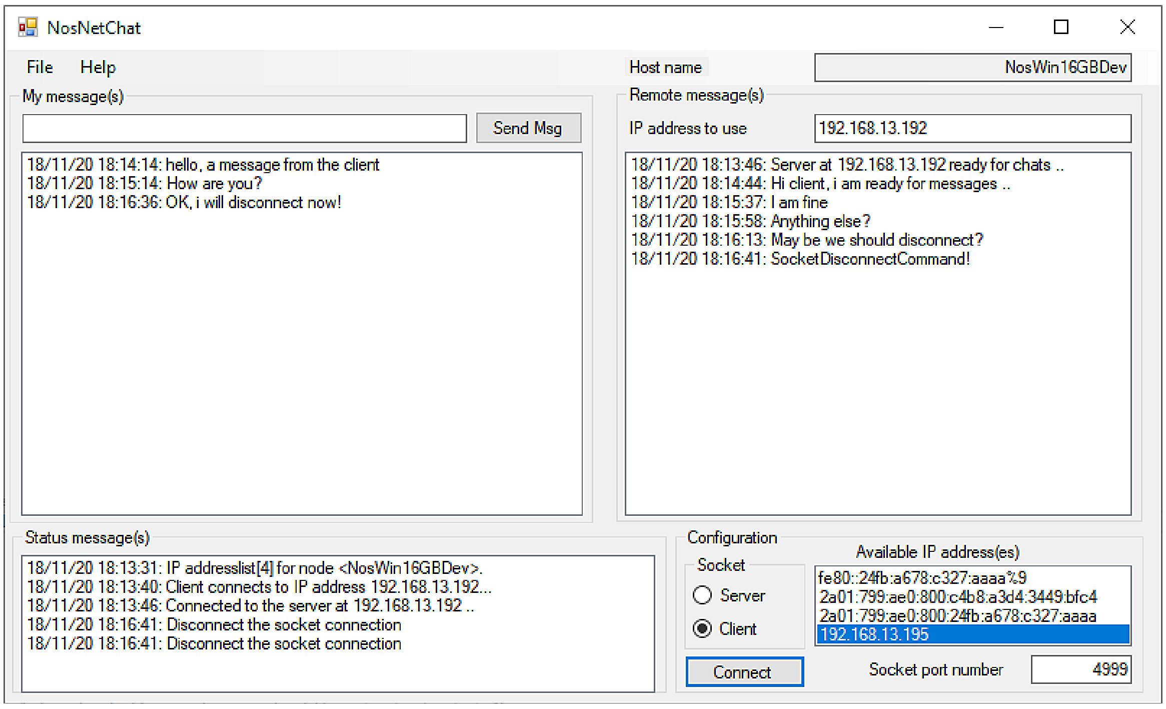
Figure 1.29: The configuration and messages for the server application.

configuration section, where the user selects the socket (server or client), the port number and the IP to use.

The setup of the system should start with the server application, and the configuration section. Select an unused port number, a good choice is to select a port number above 1024. The valid port numbers are 0 to 65535. Note that using an unused port number above 1024 should result in a warning from the Antivirus/Internet protection of your system as these ports will also be used for a cyber security attach of computers. Select the socket type, server or client, and an IP address. A socket connection consists of a socket number and an IP address. In the upper right corner of the application in Figure 1.29 the name of node is shown, the server node name for the socket the connection. The IP addresses are shown in the list in the lower right corner, shown both IPv4 and IPv6 addresses for this node. A good choice can be to use the IPv4 addresses as this is more easy for the user of the client application to input the IP address. Click on the address that you want to use, and the IP address will be shown in the right upper corner. Use the *Connect* button to start the server socket thread. Messages in the status window, lower left corner, shows that IP addresses are listed, and that the server has started with a specific IP address. The Connect button will change the text to *Disconnect* when connected.

The setup of the client application must use the same setting as the server application, as shown in Figure 1.30. The node name in the upper right corner is the name of the client node, not the same as the server node. The next field in the upper right corner must be the IP address of the server, so the same IP address as in Figure 1.29. Use the same IP address and socket number as the server, and use the *Connect* button to connect to the server socket. Information in the status windows shows the status of this connection. When connected, input a message in the input field in the upper left corner and use the *Send Msg* button to the send the message to the remote node. The upper left list window shows the messages sent from the node, and the upper right list window shows the messages received from the remote node.

The source code for this application will not be included, but is using the same methods as shown for the console application. Some comments with code snippets will however be included for specific part of this application, as a multitasking (multithreading) GUI application will include some challenges with sending events between the threads. The software architecture is a three tier architecture for this application, as shown in Figure 1.27 and the normal message flow is to use methods to get information from a lower tier. The challenge is to get information from a lower tier up to a higher tier when receiving external information and in this application is the publish/subscriber concept (best practice - design pattern) used. The publisher/subscriber concept consist of one publisher, active on a lower tier, and activated when receiving external information. Any method in a higher tier can subscribe on these


NosNetChat

File Help

Host name NosWin16GBDev

My message(s)

Send Msg

```
18/11/20 18:14:14: hello, a message from the client
18/11/20 18:15:14: How are you?
18/11/20 18:16:36: OK, i will disconnect now!
```

Remote message(s)

IP address to use 192.168.13.192

```
18/11/20 18:13:46: Server at 192.168.13.192 ready for chats ..
18/11/20 18:14:44: Hi client, i am ready for messages ..
18/11/20 18:15:37: I am fine
18/11/20 18:15:58: Anything else?
18/11/20 18:16:13: May be we should disconnect?
18/11/20 18:16:41: SocketDisconnectCommand!
```

Status message(s)

```
18/11/20 18:13:31: IP addresslist[4] for node <NosWin16GBDev>.
18/11/20 18:13:40: Client connects to IP address 192.168.13.192...
18/11/20 18:13:46: Connected to the server at 192.168.13.192 ..
18/11/20 18:16:41: Disconnect the socket connection
18/11/20 18:16:41: Disconnect the socket connection
```

Configuration

Socket: Server [✓ Client]

Available IP address(es)

```
fe80::24fb:a678:c327:aaaa%9
2a01:799:ae0:800:c4b8:a3d4:3449:bfc4
2a01:799:ae0:800:24fb:a678:c327:aaaa
192.168.13.195
```

Connect

Socket port number 4999

Figure 1.30: The configuration and messages for the client application.

publisher, and be informed when new external information is received.

The data flow in this application is, with reference to Figure 1.27, is when sending a message is to make the message in the user tier, send the message buffer down to business tier, further down to the data tier, and the active thread for sending to the remote node. The challenge is receiving a message from the remote node. One solution can be to let the user tier poll the socket thread all the time but this will waste cpu power and memory, here the publisher/subscribe concept should be used. The threads will be the publishers, and the user tier will be the subscriber, receiving an event whenever new information is received from the remote node.

The source code snippets for the publisher/subscriber logic:

1 : define the callback methods as delegate methods in the header of the LayerUser class:

```
/// <summary>
/// Purpose: event handler with the UiEventArgs object as information
/// </summary>
public delegate void HandlerUiMsg(object rSender, UiEventArgs eUi);
delegate void AddReceivingMsgCallback(string txt);
delegate void AddStatusMsgCallback(string txt);
```

2 : define the event handler in both the LayerUser and LayerData classes:

```
    HandlerUiMsg eListenUiMsg = new HandlerUiMsg(OnHandlerUiMsg);
    rDataLayer.ePublishUiMsg += eListenUiMsg;
```

3 : define the event handlers. Note that the events has to be sent between different threads so these handlers must be thread safe:

```
    /// <summary>
    public void OnHandlerUiMsg(object rsender, UiEventArgs eArgs)
    ///
    /// Purpose: handling UI event commands
```

```
/// Version: 14-NOV-20: NOS
/// </summary>
{
    switch (eArgs.iCmd)
    {
        case (int)Constants.UiEventCmd.MsgRxList:
            addReceivingMsgThread(eArgs.sMsg);
            break;
        case (int)Constants.UiEventCmd.MsgStatusList:
            AddStatusMsgThread(eArgs.sMsg);
            break;
        case (int)Constants.UiEventCmd.MsgSocketButton:
            rGuiForm.buttonSocketConnect.Text = eArgs.sMsg;
            break;
        default:
            addStatusMsg("Error: Rx invalid UI event cmd = " + eArgs.iCmd.ToString());
            break;
    }
}

/// <summary>
public void AddStatusMsg(string sStatusMsg)
///
/// Purpose: add a messge to the status window
/// Version: 13-NOV-20: NOS
/// </summary>
{
    string sMsg;

    sMsg = DateTime.Now.ToString() + ": " + sStatusMsg;
    rGuiForm.listBoxStatusMessages.Items.Add(sMsg);
}
/// <summary>
public void AddStatusMsgThread(string sRxMsg)
///
/// Purpose: update the status window in the GUI, threadsafe
/// Version: 1.0: 11-OCT-20: NOS
/// </summary>
{
    // InvokeRequired compares the thread IDs
    if (rGuiForm.listBoxStatusMessages.InvokeRequired == true)
    {
        try
        {
            AddStatusMsgCallback rCallback = new AddStatusMsgCallback(addStatusMsgThread);
            rGuiForm.Invoke(rCallback, new Object[] { sRxMsg });
        }
        catch
        {
            // No support, normally at startup or close time
        }
    }
    else
    {
        AddStatusMsg(sRxMsg);
    }
}
/// <summary>
```

```
    private void AddReceiveMsg(string sRxMsg)
    ///
    /// Purpose: add a messge to the receive message window
    /// Version: 13-NOV-20: NOS
    /// </summary>
    {
        string sMsg;

        sMsg = DateTime.Now.ToString() + ": " + sRxMsg;
        rGuiForm.listBoxRxMessages.Items.Add(sMsg);
    }
    /// <summary>
    public void AddReceivingMsgThread(string sRxMsg)
    ///
    /// Purpose: update the receive window in the GUI, threadsafe
    /// Version: 1.0: 11-OCT-20: NOS
    /// </summary>
    {
        // InvokeRequired compares the thread IDs
        if (rGuiForm.listBoxRxMessages.InvokeRequired == true)
        {
            try
            {
                AddReceivingMsgCallback rCallback = new
                                        AddReceivingMsgCallback(addReceivingMsgThread);
                rGuiForm.Invoke(rCallback, new Object[] { sRxMsg });
            }
            catch
            {
                // No support, normally at startup or close time
            }
        }
        else
        {
            AddReceiveMsg(sRxMsg);
        }
    }
```

4 : define the event class for the event:

```
/// <summary>
public class UiEventArgs : EventArgs
///
/// Purpose: a public class for event messages for UI publisher/subscribe
/// Version: 14-NOV-20: NOS
/// </summary>
{
    public int iCmd;
    public int iCode;
    public string sMsg;
}
```

5 : define the event handler in the LayerData class:

```
    /// <summary>
    public void publishUiEvent(UiEventArgs eArgs)
```

```
///
/// Purpose: publisher event methods for sending events to the UI layer
/// Version: 13-NOV-20: NOS
/// </summary>
{
    if (ePublishUiMsg != null)
    {
        ePublishUiMsg(this, eArgs);
    }
}
```

The souce code snippets for making the either the server socket thread or the client socket thread:

```
/// <summary>
private bool makeSocketConnection(int iSocketMode, int iSockId, string sIpAddress)
///
/// Purpose: make a socket connection, ether server or client
/// Version: 12-NOV-20: NOS
/// </summary>
{
    bool bValid;

    ipAddress = IPAddress.Parse(sIpAddress);
    iSocketPortId = iSockId;
    switch (iSocketMode)
    {
        case (int)Constants.Socket.Server:
            publishEventToUi((int)Constants.UiEventCmd.MsgStatusList,
                "Server waits for clients to connect at IP address " + sIpAddress +
                "...", iSocketMode);
            iActiveSocketConnection = (int)Constants.Socket.Server;
            /// Start the server thread
            thSocketThread = new Thread(new ThreadStart(this.ServerSocketThread));
            bValid = true;
            break;
        case (int)Constants.Socket.Client:
            publishEventToUi((int)Constants.UiEventCmd.MsgStatusList,
                "Client connects to IP address " + sIpAddress + "...", iSocketMode);
            iActiveSocketConnection = (int)Constants.Socket.Client;
            /// Start the client thread
            thSocketThread = new Thread(new ThreadStart(this.ClientSocketThread));
            bValid = true;
            break;
        default:
            publishEventToUi((int)Constants.UiEventCmd.MsgStatusList,
                "Socket error: Invalid socket mode = " + iSocketMode.ToString() + "!",
                iSocketMode);
            iActiveSocketConnection = (int)Constants.Socket.None;
            bValid = false;
            break;
    }
    thSocketThread.Start();
    publishEventToUi((int)Constants.UiEventCmd.MsgSocketButton, "Disconnect",
                iActiveSocketConnection);
    return bValid;
}
```

The souce code snippets for the server socket thread, including the receive method:

```
/// <summary>
public void ServerSocketThread()
///
/// Purpose: thread for handling the server thread
/// Version: 16-NOV-20: NOS
/// </summary>
{
    byte[] baBuffer;
    /// make a TCP connection point based on the IP address and the port number
    tcpListener = new TcpListener(ipAddress, iSocketPortId);
    /// Start the listner connection to let clients to connect
    tcpListener.Start();
    /// wait for a client to connect to this conenction point (port ID)
    tcpClient = tcpListener.AcceptTcpClient();
    /// a client has connected, make a data stream for communicating with the client
    tcpNetworkStream = tcpClient.GetStream();
    /// send a start message
    publishEventToUi((int)Constants.UiEventCmd.MsgStatusList,
        "A client is connected at " + ipAddress.ToString() + "..", 0);
    sendSocketMessage("Server at " + ipAddress.ToString() + " ready for chats ..");
    /// Connected, wait for messages
    while (bRunFlag == true)
    {
        /// wait for receiving a message from the client
        baBuffer = new byte[64];
        if (tcpNetworkStream.Read(baBuffer, 0, 64) > 0)
        {
            receiveSocketMsg(baBuffer);
        }
        Thread.Sleep(100);
    }
    /// close the data stream channel
    if (tcpNetworkStream != null)
    {
        tcpNetworkStream.Close();
    }
    /// close the socket connection
    if (tcpClient != null)
    {
        tcpClient.Close();
    }
}

/// <summary>
private void receiveSocketMsg(byte[] baBuffer)
///
/// Purpose: receiving a data buffer from the socket connection
/// Version: 13-NOV-20: NOS
/// </summary>
{
    string sBuffer;

    sBuffer = Encoding.ASCII.GetString(baBuffer);
    sBuffer = sBuffer.TrimEnd('\0');
    publishEventToUi((int)Constants.UiEventCmd.MsgRxList, sBuffer, 0);
    if (sBuffer.Equals(SOCKET_DISCONNECT_CMD) == true)
```

```
        {
            closeSocketConnections();
        }
    }
```

The souce code snippets for the writing method, using the either the server or client socket:

```
    /// <summary>
    public bool sendSocketMessage(string sSendMsg)
    ///
    /// Purpose: the message to send on an active socket connection
    /// Version: 13-NOV-20: NOS
    /// </summary>
    {
        bool bValid;
        byte[] baBuffer;

        switch (iActiveSocketConnection)
        {
            case (int)Constants.Socket.Server:
            case (int)Constants.Socket.Client:
                if (tcpNetworkStream != null)
                {
                    baBuffer = Encoding.ASCII.GetBytes(sSendMsg);
                    tcpNetworkStream.Write(baBuffer, 0, baBuffer.Length);
                    bValid = true;
                }
                else
                {
                    bValid = false;
                }
                break;
            case (int)Constants.Socket.None:
            default:
                bValid = false;
                break;
        }
        return bValid;
    }
```

1.4.7 Security

Both the reading and writing functions are just sending plain strings (text buffers) that can easy can be read by other applications, and these applications can also easy change the contents of the strings. Adding any type of security requires another protocol on top of TCP/IP as TCP/IP is only handling data buffers and the receivers must have some sort of protocol for decrypting the information. Applications like Wireshark can very easly get all the data buffers for all TCP/IP connections on your computer.

Looking at Figure 1.21 the Presentation layer is normally containing any data security function. One way to extend this application can be by crypting the strings before sending them and decrypt them when receiving them. One easy way of doing this can be to add a number to each character in the string, for example 2, meaning that the character A will be C, B will D and so one. The sender and receiver must both agree on the coding so the receiver will, in this case, subtract 2 from each of the characters. The string "Hello world" will then be transmitted as "Jgnnq yqtnf" as long as characters outside the range A to Z and a to z will not be crypted/decrypted.

Bibliography

Bentham, J. (2002), *TCP/IP Lean, Web servers for embedded systems*, second edn, CMPBooks.

Comer, D. E. (1991), *Internetworking with TCP/IP; Volume I; Principles, Protocols and Architecture*, second edn, Prentice-Hall International Editions.

Crowcroft, J. & Phillips, I. (2002), *TCP/IP and Linux Implementation; Systems code for Linux Internet*, Wiley.

Elm (2016), *ELM327 - OBD to RS232 Interpreter.* Accessed FEB-2019.

Houldsworth, J. (1990), *OSI Handbook*, International Computers Ltd. (ICL), UK.

Levermore, G. J. (2000), *Building Energy Management Systems Applications to Low-Energy HVAC and Natural Ventilation Control*, second edn, E and FN Spon.

Mahmoud, Q. H. (2004), *Middleware for Communications*, Wiley, West Sussex PO19 8SQ, England.

Stallings, W. (2001), *Wireless communication and networks*, Prentice-Hall International Editions.

Stevens, W. R. (1990), *Unix networking programming*, Prentice-Hall International Editions.

Index