

## Índex Annex

<b>ANNEX A: BIBLIOTECA USBTINLIB</b>	<b>2</b>
<b>ANNEX B: MÓDUL CONSTANTSUSB</b>	<b>12</b>
<b>ANNEX C: MÓDUL PERIODICTHREAD</b>	<b>14</b>
<b>ANNEX D: APLICACIÓ GUI CANUSBVIEWER</b>	<b>16</b>

## ANNEX A: Biblioteca usbTinLib

```

#!/usr/bin/python
#-*- coding: utf-8 -*-

import serial
import constantsUSB as cntUSB
import queue
import PeriodicThread as thread
import time

class USBTin(object):
    #Region USBTin
    def __init__(self, port, CANbaud, mode):
        """
        Initializing function for instance USBTin

        Parameters:
            - port (str) - COM Virtual port where the USBTin hardware is
            connected in the format "COMx". - Ex. "COM4"
            - CANbaud (str) - Baudrate for the CAN Channel. Defined by
            the keys in dictionaries BAUD or canBITRATE from constantsUSB. - Ex.
            "BAUD_125K" or "canBITRATE_125K"
            - mode (str) - Three possible modes to open the CAN Channel.
            Standard mode("Normal"), receiving but not writing ("Listen-Only"),
            receive own sent messages("Loopback")

        Returns:
            None
        """

        #Attributes SerialPort (baudrate - only supported)
        self.SerialPort=serial.Serial()
        self.SerialPort.port=port
        self.SerialPort.baudrate=115200
        self.SerialPort.timeout=0.01
        self.SerialPort.writeTimeout=0.01

        #CAN Attributes
        self.baudrate=CANbaud
        self.mode=mode
        self.CANOC="Closed"

        #Response and Messages Queues
        self.MSGqueue=queue.Queue()
        self.RPSqueue=queue.Queue()

        #FSM attributes
        self.inFrame=b' '
        self.state=cntUSB.STATE_INI

        #Threads
        self.FSMThread=thread.PeriodicThread(self.processBytes, 0.0001)

        self.openSerial()
    
```



```

def openSerial(self):
    """
        Performs a series of actions to establish a communication between
        the user (PC) and USBtin:

        1) Open the serial port.
        2) Initialize Periodic Thread for Finite State Machine function

        Parameters:
            None

        Returns:
            None
            (Print to the console if port is opened or it have raised an
            exception)
    """
    try:
        self.SerialPort.open()
        self.SerialPort.write(bytes('\rC\r', 'ascii')) #Recommended
        time.sleep(0.05) #Recommended
        self.SerialPort.write(bytes('C\r', 'ascii')) #Recommended
        self.SerialPort.read(100); #Recommended
        self.FSMThread.start()
        time.sleep(0.1)
        print("Port is opened.")
        self.RPSqueue.queue.clear()
    except serial.SerialException:
        print("Device not found - The COM port cannot be opened.")

def closeSerial(self):
    """
        Performs a series of actions to end the communication between the
        user (PC) and USBtin:

        1) Stop Periodic Thread for Finite State Machine function
        2) Close the CAN channel (if it is opened)
        3) Close the serial port

        Parameters:
            None

        Returns:
            None
            (Print to the console when the port is closed)
    """
    self.FSMThread.cancel()
    self.FSMThread.join()
    if self.CANOC=="Open": self.canClose()
    self.SerialPort.close()
    print("Port is closed.")

def askHWversion(self):
    """
        Get hardware USBtin version.
    """

```



```

Parameters:
    None

Returns:
    String indicating hardware version
"""
if self.CANOC=="Closed":
    self.RPSqueue.queue.clear()

self.SerialPort.write(bytes(cntUSB.usb_VERSION_HW+cntUSB.CR,'ascii'))
    time.sleep(0.2)
    return self.ReadResponse()
else:
    return cntUSB.canERR_KO

def askFWversion(self):
"""
Get firmware USBtin version.

Parameters:
    None

Returns:
    String indicating hardware version
"""
if self.CANOC=="Closed":
    self.RPSqueue.queue.clear()

self.SerialPort.write(bytes(cntUSB.usb_VERSION_FW+cntUSB.CR,'ascii'))
    time.sleep(0.2)
    return self.ReadResponse()
else:
    return cntUSB.canERR_KO

def askUSBstatus(self):
"""
Read status/error flag of can controller

Parameters:
    None

Returns:
    Fxx[CR] with xx as hexadecimal byte with following error
flags:

        Bit 0 - not used\n
        Bit 1 - not used\n
        Bit 2 - Error warning (Bit EWARN of MCP2515)\n
        Bit 3 - Data overrun (Bit RX1OVR or RX0OVR of MCP2515)\n
        Bit 4 - not used\n
        Bit 5 - Error-Passive (Bit TXEP or RXEP of MCP2515)\n
        Bit 6 - not used\n
        Bit 7 - Bus error (Bit TXBO of MCP2515)\n
"""
if self.CANOC=="Closed":
    self.RPSqueue.queue.clear()

self.SerialPort.write(bytes(cntUSB.usb_READSTATUS+cntUSB.CR,'ascii'))

```

```

        time.sleep(0.1)
        return self.ReadResponse()
    else:
        return cntUSB.canERR_KO

def SerialSetTimestamping(self, x):
    """
    Set time stamping on/off

    Parameters:
    - x (bin):
        0: off
        1: on

    Returns:
    None
        (Print to the console if the Time stamping is now on/off/or
        there is an input error)

    """
    self.SerialPort.write(bytes(cntUSB.usb_TIMESTAMP+str(x)+cntUSB.CR, 'ascii'))
    if x==0:
        print("Timestamping Off")
        time.sleep(0.05)
        print(self.ReadResponse())
    elif x==1:
        print("Timestamping On")
        time.sleep(0.05)
        print(self.ReadResponse())
    else:
        print("Timestamping input error!")
        time.sleep(0.05)
        print(self.ReadResponse())

#Region CAN

def canOpenChannel(self):
    """
    Opens the CAN channel in the mode specified on instance
    attributes.
    By default, the function is called inside openSerial function.

    Parameters:
    None

    Returns:
    None
        (Print to the console which CAN channel mode have been
        opened)
    """
    try:
        self.canSetBitrate(self.baudrate)
    except ValueError:

```



```

        return cntUSB.canERR_PARAM

    if self.mode == "Normal":

self.SerialPort.write(bytes(cntUSB.can_OPEN_NORMALMODE+cntUSB.CR,'ascii'))
)
    print("CAN Channel opened in normal mode")
    self.CANOC="Open"
elif self.mode == "Listen-only":

self.SerialPort.write(bytes(cntUSB.can_OPEN_LISTENONLY+cntUSB.CR,'ascii'))
)
    print("CAN Channel opened in listen-only mode")
    self.CANOC="Open"
elif self.mode == "Loopback":

self.SerialPort.write(bytes(cntUSB.can_OPEN_LOOPBACK+cntUSB.CR,'ascii'))
    print("CAN Channel opened in loopback mode")
    self.CANOC="Open"
else:
    print("CAN channel cannot be opened. Only -Normal-, -Loopback- or -Listen-only- are taken as inputs")

    time.sleep(0.1)
    return self.ReadResponse()

def canClose(self):
"""
Closes the CAN channel.
By default, the function is called inside closeSerial function.

Parameters:
    None

Returns:
    None
    (Print to the console when the CAN channel is closed)

"""
self.SerialPort.write(bytes(cntUSB.can_CLOSE+cntUSB.CR,'ascii'))
print("CAN Channel closed")
time.sleep(0.1)
self.CANOC="Closed"
return self.ReadResponse()

def canSetBitrate(self, baudrate):
"""
Sets baudrate for the CAN channel. It must be set before CAN
channel is opened.
By default, canSetBitrate is called inside openSerial function
(with self.baudrate as parameter).

Parameters:
    - baudrate: Defined by the keys in dictionaries BAUD or
canBITRATE

Returns:

```



```

    None

    Raises:
        ValueError - if the baudrate value is not standarized nor
        contained in baudrate dictionaries from constantsUSB
    """
    if baudrate in cntUSB.BAUD:
        self.SerialPort.write(bytes(cntUSB.can_SETBITRATE+str(cntUSB.BAUD[baudrate])+cntUSB.CR,'ascii'))
        self.baudrate=baudrate
        time.sleep(0.1)
        if self.ReadResponse()==0:
            print("Baudrate is set successfully")
    elif baudrate in cntUSB.canBITRATE:
        self.SerialPort.write(bytes(cntUSB.can_SETBITRATE+str(cntUSB.canBITRATE[baudrate])+cntUSB.CR,'ascii'))
        self.baudrate=baudrate
        time.sleep(0.1)
        if self.ReadResponse()==0:
            print("Baudrate is set successfully")
        else:
            print('Wrong Bitrate Value. Please set a proper value calling
            USBTin.canSetBitrate() method or change attribute value USBTin.baudrate')
            raise ValueError

    def canSetFilter(self, filter):
        self.SerialPort.write(bytes(cntUSB.can_FILTER_CODE+filter+cntUSB.CR,'ascii'))
        time.sleep(0.1)
        print(self.ReadResponse())

    def canSetMask(self, mask):
        self.SerialPort.write(bytes(cntUSB.can_FILTER_MASK+mask+cntUSB.CR,'ascii'))
        time.sleep(0.1)
        print(self.ReadResponse())

    def canWrite(self,msgId, dlc, messageList, *mask):
        """
        Writes a message through the serial port to the CAN channel

        Parameters:
            - msgId (str) - Message identifier
                - String in hexadecimal format between 000 to 7FF
                (Integer between 0 and 2047) for standard messages
                - String in hexadecimal format between 00000000 to
                1FFFFFFF (Integer between 0 and 536870911) for extended messages.

                Zeros are autocompleted in case it is not given the
                complete expression.
            - dlc (int) - Data Length Code. Indicates the number of data
                bytes in the message.
                - Integer from 1 to 8.
            - message (list of str - hex format)
        """

```



```

        - List of strings of data bytes in hexadecimal
representation. List length must be dlc.
        - mask (int) - Multiple parameters can be defined
          - 0x0001: Remote Request Message
          - 0x0002: Standard ID Message
          - 0x0004: Extended ID Message

    Returns:
    None
    (Print to the console that the message have been send
successfully)
"""

if self.CANOC=="Open":
    if cntUSB.canMSG_RTR in mask:
        if cntUSB.canMSG_STD in mask:
            msgId=msgId.zfill(3)
            frame_str="r"+msgId.upper()
            if messageList!=[]: return 'Remote Request Message
cannot have Data Bytes'
            elif cntUSB.canMSG_EXT in mask:
                msgId=msgId.zfill(8)
                frame_str="R"+msgId.upper()
                if messageList!=[]: return 'Remote Request Message
cannot have Data Bytes'
                elif cntUSB.canMSG_EXT in mask and cntUSB.canMSG_STD in
mask:
                    return "STD and EXT ID messages cannot be defined at
the same time. Please choose only one"
                else:
                    return "STD or EXT ID message is not defined. Please
define it in mask"
            else:
                if cntUSB.canMSG_STD in mask:
                    msgId=msgId.zfill(3)
                    frame_str="t"+msgId.upper()
                elif cntUSB.canMSG_EXT in mask:
                    msgId=msgId.zfill(8)
                    frame_str="T"+msgId.upper()
                elif cntUSB.canMSG_EXT in mask and cntUSB.canMSG_STD in
mask:
                    return "STD and EXT ID messages cannot be defined at
the same time. Please choose only one"
                else:
                    return "STD or EXT ID message is not defined. Please
define it in mask"
                frame_str=frame_str+str(dlc)
                for data_byte in messageList:
                    frame_str=frame_str+data_byte.zfill(2)
                self.SerialPort.write(bytes(frame_str+cntUSB.CR,'ascii'))
                time.sleep(0.1)
                return self.ReadResponse()
            else: print('CAN Channel is Closed. Message cannot be sent.')
        def processBytes(self):
"""

    If USBTin buffer is not empty, Finite State Machine function is
executed.

```

```

    processBytes is the run function in the periodic thread
"FSMThread"

    Parameters:
        None

    Returns:
        None
    """
    while self.SerialPort.inWaiting() != 0:
        self.FSM(self.SerialPort.read())

    def FSM(self, data):
        """
            Finite State Machine is in charge of classifying data received
            from USBTin.
            If state == STATE_INI, a new frame is started.
            If state == STATE_MSG or STATE_RPS, the data will be added to the
            previous started frame.
            When the carriage return is read as data, the frame will be added
            to the corresponding queue (RPSqueue or MSGqueue, depending on state)

            Parameters:
                - data (byte): Byte of data read from USBTin buffer

        Parameters:
            None
        """
        if self.state == cntUSB.STATE_INI:
            data_str=data.decode('ascii')
            if data_str==cntUSB.CHR_cT or data_str==cntUSB.CHR_t or
data_str==cntUSB.CHR_cR or data_str==cntUSB.CHR_r:
                self.inFrame=data
                self.state=cntUSB.STATE_MSG
            elif data == bytes(cntUSB.usb_ERR,'ascii'):
                self.RPSqueue.put(data)
                print("CAN Error")
                self.ReadResponse()
            elif data == bytes(cntUSB.CR,'ascii'):
                self.RPSqueue.put(data)
            else:
                self.inFrame=data
                self.state=cntUSB.STATE_RPS
        elif self.state == cntUSB.STATE_MSG:
            if data.decode('ascii') == cntUSB.CR:
                self.MSGqueue.put(self.inFrame)
                self.state = cntUSB.STATE_INI
                self.inFrame=b''
            else:
                self.inFrame=self.inFrame+data #inFrame es bytearray

        elif self.state == cntUSB.STATE_RPS:
            if data.decode('ascii')==cntUSB.CR:
                self.RPSqueue.put(self.inFrame)
                self.state=cntUSB.STATE_INI
                self.inFrame=b''
            else:
                self.inFrame=self.inFrame+data

```

```

def ReadResponse(self):
    """
        Reads and process elements (responses) from RPSqueue (instance
    attribute)

    Parameters:
        None

    Returns:
        None
        (Print to the console different strings depending on the
    response to previous messages)
    """
    if self.RPSqueue.empty():
        return cntUSB.canERR_NORPS
    else:
        evalRPS=self.RPSqueue.get()
        if evalRPS.decode('ascii') == 'z' or evalRPS.decode('ascii')
== 'Z':
            print("Message received")
            return cntUSB.canERR_OK
        elif evalRPS.decode('ascii') == cntUSB.CR:
            return cntUSB.canERR_OK
        elif bytes(cntUSB.usb_READSTATUS, 'ascii') in evalRPS:
            byte_stat=evalRPS.decode()[1:3]
            bin_stat=str(bin(int(byte_stat,16)))[2:].zfill(8)
            return bin_stat
        elif bytes(cntUSB.usb_VERSION_HW, 'ascii') in evalRPS:
            return evalRPS.decode('ascii')
        elif bytes(cntUSB.usb_VERSION_FW, 'ascii') in evalRPS:
            return evalRPS.decode('ascii')
        elif evalRPS == cntUSB.usb_ERR:
            return cntUSB.canERR_KO
        else:
            return evalRPS.decode('ascii')

def canReadMessage(self):
    """
        Reads and process elements (messages) from MSGqueue (instance
    attribute)

    Parameters:
        None

    Returns:
        - List of strings with message in the following format:
    [msgId mode, msgId, dlc, msgData]
    """
    frameList=[]
    if self.MSGqueue.empty():
        frameList.append(cntUSB.canERR_NOMSG)
        return frameList
    else:
        evalMSG=self.MSGqueue.get().decode('ascii')
        if cntUSB.CHR_cT in evalMSG:

```

```
frameList.append("Extended")
frameList.append(evalMSG[1:9])
frameList.append(evalMSG[9])
frameList.append(evalMSG[10:])
return frameList
elif cntUSB.CHR_t in evalMSG:
    frameList.append("Standard")
    frameList.append(evalMSG[1:4])
    frameList.append(evalMSG[4])
    frameList.append(evalMSG[5:])
    return frameList
elif cntUSB.CHR_cR in evalMSG:
    frameList.append("Extended RTR")
    frameList.append(evalMSG[1:9])
    frameList.append(evalMSG[9])
    frameList.append(evalMSG[10:])
    return frameList
elif cntUSB.CHR_r in evalMSG:
    frameList.append("Standard RTR")
    frameList.append(evalMSG[1:4])
    frameList.append(evalMSG[4])
    frameList.append(evalMSG[5:])
    return frameList
else:
    frameList.append(cntUSB.canERR_PARAM)
    return frameList
```

## ANNEX B: Mòdul constantsUSB

```

"""USBtin Constants"""
#USBtin Responses
CR='\x0D'           # Carriage Return (\r)
LF='\x0A'           # Line Feed (\n)
usb_ERR='\x07'

#USBtin Command Strings

usb_READSTATUS='F'
usb_TIMESTAMPING='Z'
usb_VERSION_FW='v'
usb_VERSION_HW='V'
usb_MSG RECEIVED='z'
can_CLOSE='C'
can_OPEN_LOOPBACK='l'
can_OPEN_LISTENONLY='L'
can_OPEN_NORMALMODE='O'
can_FILTER_MASK='m'
can_FILTER_CODE='M'
can_SETBITRATE='S'

can_NORMALMODE=1
can_LOOPBACK=2
can_LISTENONLY=3

#FSM State Constants
STATE_INI=0
STATE_MSG=1
STATE_RPS=2

#Message Sending Command Characters
CHR_r='r'           # RTR Standard ID
CHR_cR='R'          # RTR Extended ID
CHR_t='t'           # Standard ID
CHR_cT='T'          # Extended ID

#CAN Error Constants
canERR_OK=0
canERR_KO=-1
canERR_NOMSG=-2
canERR_NORPS=-2
canERR_PARAM=-1

#canWrite Message Masks
canMSG_RTR         =0x0001      # Message is a remote request
canMSG_STD          =0x0002      # Message has a standard ID
canMSG_EXT          =0x0004      # Message has a extended ID

#Bitrate Dictionaries
BAUD={"BAUD_10K":0, "BAUD_20K":1, "BAUD_50K":2, "BAUD_100K":3,
"BAUD_125K":4, "BAUD_250K":5, "BAUD_500K":6, "BAUD_800K":7, "BAUD_1M":8}

```



```
canBITRATE={"canBITRATE_10K":0,"canBITRATE_20K":1,"canBITRATE_50K":2,"can  
BITRATE_100K":3,"canBITRATE_125K":4,  
"canBITRATE_250K":5,"canBITRATE_500K":6,"canBITRATE_800K":7,"canBITRATE_1  
M":8}
```

## ANNEX C: Módul PeriodicThread

```

import logging
import threading

class PeriodicThread(object):
    """
    Python periodic Thread using Timer with instant cancellation
    """

    def __init__(self, callback=None, period=1, name=None, *args,
                 **kwargs):
        self.name = name
        self.args = args
        self.kwargs = kwargs
        self.callback = callback
        self.period = period
        self.stop = False
        self.current_timer = None
        self.schedule_lock = threading.Lock()

    def start(self):
        """
        Mimics Thread standard start method
        """
        self.schedule_timer()

    def run(self):
        """
        By default run callback. Override it if you want to use
        inheritance
        """
        if self.callback is not None:
            self.callback()

    def __run(self):
        """
        Run desired callback and then reschedule Timer (if thread is not
        stopped)
        """
        try:
            self.run()
        except (Exception, e):
            logging.exception("Exception in running periodic thread")
        finally:
            with self.schedule_lock:
                if not self.stop:
                    self.schedule_timer()

    def schedule_timer(self):
        """
        Schedules next Timer run
        """
        self.current_timer = threading.Timer(self.period, self.__run,
                                             *self.args, **self.kwargs)

```



```
if self.name:
    self.current_timer.name = self.name
self.current_timer.start()

def cancel(self):
    """
    Mimics Timer standard cancel method
    """
    with self.schedule_lock:
        self.stop = True
        if self.current_timer is not None:
            self.current_timer.cancel()

def join(self):
    """
    Mimics Thread standard join method
    """
    self.current_timer.join()

if __name__ == "__main__":
    import time
    def fun_out():
        print("Hello World")

    mth = PeriodicThread(fun_out, 1)
    mth.start()
    time.sleep(5)
    mth.cancel()
    mth.join()
```

## ANNEX D: Aplicació GUI CANUSBviewer

```

#!/usr/bin/python
#-*- coding: utf-8 -*-

import tkinter as tk
from tkinter import ttk
import usbTinLib as usb
import sys
import glob
import serial
import PeriodicThread as perThr
import constantsUSB as cntUSB
import time

#Function to enumerate USBTin ports
def serial_ports():
    """ Lists serial port names

        :raises EnvironmentError:
            On unsupported or unknown platforms
        :returns:
            A list of the serial ports available on the system
    """
    if sys.platform.startswith('win'):
        ports = ['COM%s' % (i + 1) for i in range(256)]
    elif sys.platform.startswith('linux') or \
        sys.platform.startswith('cygwin'):
        # this excludes your current terminal "/dev/tty"
        ports = glob.glob('/dev/tty[A-Za-z]*')
        for i in range(256):
            ports.append('/dev/ACM%s' %(i+1))
            ports.append('/dev/ttyACM%s' %(i+1))
    elif sys.platform.startswith('darwin'):
        ports = glob.glob('/dev/tty.*')
        ports.append('/dev/ACM%s' %(i+1))
        ports.append('/dev/ttyACM%s' %(i+1))
    else:
        raise EnvironmentError('Unsupported platform')

    result = []
    for port in ports:
        try:
            s = serial.Serial(port)
            s.close()
            result.append(port)
        except (OSError, serial.SerialException):
            pass
    result2=[]
    for port in result:
        if isUSBTin(port): result2.append(port)
    return result2

#Function tells if a port has a USBTin connected or not

```



```

def isUSBTin(port):
    serial_port=serial.Serial(port=port, baudrate=115200, timeout=0.1,
    writeTimeout=0.1)
    serial_port.write(bytes('\rC\r','ascii'))
    serial_port.write(bytes('C\r','ascii'))
    serial_port.read(100)

    serial_port.write(bytes(cntUSB.usb_VERSION_HW+cntUSB.CR+cntUSB.LF,'ascii')
    )
    HW=serial_port.read(10).decode('ascii')
    if HW=='': return False

    serial_port.write(bytes(cntUSB.usb_VERSION_FW+cntUSB.CR+cntUSB.LF,'ascii'
    ))
    FW=serial_port.read(10).decode('ascii')
    if HW[0]=='V' and FW[0]=='v':
        serial_port.close()
        return True
    else:
        serial_port.close()
        return False

#Frame related with USBTin instance. Creates and packs the Frame.
class USBTinFrame():
    def __init__(self, frame, COMPort, Baudrate, Mode, mainClass):

        self.MainFrame=ttk.LabelFrame(frame, text='USBTin in Serial Port
'+COMPort)
        self.RightFrame=ttk.Frame(self.MainFrame)
        self.ButtonFrame=ttk.Frame(self.RightFrame)
        self.mainClass=mainClass

        self.USBLib=usb.USBTin(COMPort, Baudrate, Mode)

        self.baudrate=Baudrate
        self.COMPort=COMPort
        self.mainClass.opened_instance[self.COMPort]=self
        print(self.mainClass.opened_instance)

        if Mode=='Normal': self.CANMode='Standard Mode'
        elif Mode=='Listen-Only': self.CANMode='Listen-Only Mode'
        elif Mode=='Loopback': self.CANMode='Loopback Mode'
        self.CANOC='Close'
        self.TimeStamping='Off'

        self.MessageMonitor=self.ReadMessageFrame(self.MainFrame)

    self.CANMenu=self.MenuCANMessage(self.MainFrame,self.MessageMonitor,
    self.USBLib)

        self.closeButton=ttk.Button(self.ButtonFrame, text="Close
"+self.COMPort+' Port Serial',
        command=self.closeInstance).pack(side=tk.BOTTOM,anchor=tk.S)
        self.closeCAN=ttk.Button(self.ButtonFrame, text="Open/Close CAN
Channel", command=self.CANCcloseOpen).pack(side=tk.BOTTOM, anchor=tk.S)

```



```

        self.ClearButton=ttk.Button(self.ButtonFrame, text="Clear
Console",
command=self.MessageMonitor.clearscreen).pack(side=tk.BOTTOM, anchor=tk.S)
        self.EditButton=ttk.Button(self.ButtonFrame, text="Edit CAN
Parameters", command=self.EditButton).pack(side=tk.TOP, anchor=tk.N)

        self.CANMenu._pack(side=tk.LEFT, anchor=tk.N)
        self.MessageMonitor._pack(side=tk.LEFT, anchor=tk.N)

        self.StatFrame=self.Stats(self.RightFrame, self)
        self.StatFrame._pack(side=tk.TOP, anchor=tk.N)
        self.ButtonFrame.pack(side=tk.BOTTOM, anchor=tk.S, pady=(120,0))

        self.RightFrame.pack(side=tk.LEFT, anchor=tk.N)

        self.ReadThread=perThr.PeriodicThread(callback=self.ReadMessage,
period=0.02)
        self.ReadThread.start()

        self.MainFrame.pack(side=tk.TOP, pady=(10,0))

    def CANcloseOpen(self):
        if self.CANOC=='Open':
            self.USBLib.canClose()
            self.CANOC='Close'
        else:
            self.USBLib.canOpenChannel()
            self.CANOC='Open'

        self.StatFrame.update()

    def ReadMessage(self):
        answer=self.USBLib.canReadMessage()
        if answer != [-2]:
            mode=answer[0]
            id=answer[1]
            dlc=answer[2]
            data=answer[3]
            if data!="":
                data_bytes=[]
                dlc_eval=int(dlc)
                ind=0
                while ind<dlc_eval*2:
                    data_bytes.append(data[ind:ind+2])
                    ind+=2
            if mode=='Standard' or mode=='Extended':
                line='MSG-R\t\t'+mode+'\t\t'+hex(id)+'\t\t'+str(dlc)
                for ibyte in data_bytes:
                    line=line+'\t\t'+hex(ubyte)
            else:
                line='MSG-R\t\t'+mode+'\t\t'+hex(id)+'\t\t'+str(dlc)+'\t'
            self.MessageMonitor.writenewline(line+'\n')

    def closeInstance(self):
        try:
            self.ReadThread.cancel()
            self.USBLib.closeSerial()

```



```

        self.mainClass.opened_ports.remove(self.COMPort)
    del self.mainClass.opened_instance[self.COMPort]
    self.MainFrame.destroy()
except AttributeError:
    self.ReadThread.cancel()
    self.MainFrame.destroy()

if self.mainClass.opened_ports==[]:

self.mainClass.welcome=WelcomeMessage(self.mainClass.MainFrame)

def EditButton(self):
    self.EditParameters(self.MainFrame, self.USBLib, self)

class EditParameters():
    def __init__(self, root, usbInst, USBTinframe):
        self.usb=usbInst
        self.tin=USBTinframe

        self.EditWin=tk.Toplevel(master=root)
        self.EditWin.title('Edit CAN Parameters')
        self.frameEdit=ttk.Frame(self.EditWin, width=400)

        self.frameBaudrate=ttk.Frame(self.frameEdit)
        baudLab=ttk.Label(self.frameBaudrate,
text="Baudrate").pack(side=tk.LEFT)
        self.BAUDVar=tk.StringVar()

BAUDBox=ttk.Combobox(self.frameBaudrate, textvariable=self.BAUDVar, values=[
"BAUD_10K", "BAUD_20K", "BAUD_50K", "BAUD_100K", "BAUD_125K",
"BAUD_250K", "BAUD_500K", "BAUD_800K", "BAUD_1M"])
        BAUDBox.pack(side=tk.LEFT)
        BAUDBox.set("BAUD_125K")

        self.frameMode=ttk.Frame(self.frameEdit)
        modeLab=ttk.Label(self.frameMode, text='CAN
Mode').pack(side=tk.LEFT)
        self.CANModeVar=tk.StringVar()

CANModeBox=ttk.Combobox(self.frameMode, textvariable=self.CANModeVar, values=[
"Standard Mode", "Listen-Only Mode", "Loopback Mode"])
        CANModeBox.pack(side=tk.LEFT)
        CANModeBox.set("Standard Mode")

        self.frameTime=ttk.Frame(self.frameEdit)

timeLab=ttk.Label(self.frameTime, text='Timestamping').pack(side=tk.LEFT)
        self.varTime=tk.BooleanVar()
        self.timeoff = ttk.Radiobutton(self.frameTime, text="Off",
variable=self.varTime, value=False).pack(anchor=tk.W, side=tk.LEFT)
        self.timeon = ttk.Radiobutton(self.frameTime, text="On",
variable=self.varTime, value=True).pack(anchor=tk.W, side=tk.LEFT)

```



```

        self.varTime.set(False)

        self.frameBaudrate.pack(side=tk.TOP, anchor=tk.W)
        self.frameMode.pack(side=tk.TOP, anchor=tk.W)
        self.frameTime.pack(side=tk.TOP, anchor=tk.W)

        self.OKButton=ttk.Button(self.frameEdit, text='Accept',
command=self.Edit).pack(side=tk.BOTTOM, anchor=tk.S)

        self.frameEdit.pack()

        self.EditWin.lift()

        self.EditWin.mainloop()

    def Edit(self):
        self.usb.canClose()
        self.usb.canSetBitrate(self.BAUDVar.get())
        self.usb.SerialSetTimestamping(self.varTime.get())
        self.mode=self.CANModeVar.get()
        if self.mode == "Standard Mode": self.usb.mode="Normal"
        elif self.mode == "Listen-Only Mode": self.usb.mode="Listen-
only"
        elif self.mode == "Loopback Mode": self.usb.mode="Loopback"
        self.usb.canOpenChannel()
        self.EditWin.destroy()

        self.tin.baudrate=self.BAUDVar.get()
        self.tin.CANMode=self.mode
        self.tin.TimeStamping=self.varTime.get()

        self.tin.StatFrame.update()

    class Stats():
        def __init__(self, frame, USBTinframe):
            self.StatsFrame=ttk.Frame(frame)
            self.tin=USBTinframe

            self.statTitle=ttk.Label(self.StatsFrame, text="USBTin
Parameters", relief=tk.SOLID).pack(side=tk.TOP, pady=(2,2))
            self.BaudVar=tk.StringVar()
            self.COMVar=tk.StringVar()
            self.CANmode=tk.StringVar()
            self.TimeStamp=tk.StringVar()
            self.CANOC=tk.StringVar()

            self.BaudFrame=ttk.Frame(self.StatsFrame)
            self.COMFrame=ttk.Frame(self.StatsFrame)
            self.CANFrame=ttk.Frame(self.StatsFrame)
            self.TimeFrame=ttk.Frame(self.StatsFrame)
            self.CANOCFrame=ttk.Frame(self.StatsFrame)

            self.COMLabel=ttk.Label(self.COMFrame, text='Serial Port:
').pack(side=tk.LEFT, anchor=tk.W)
            self.BaudLabel=ttk.Label(self.BaudFrame, text='CAN Baudrate:
').pack(side=tk.LEFT, anchor=tk.W)

```

```

        self.CANLabel=ttk.Label(self.CANFrame, text='CAN Mode: ')
        .pack(side=tk.LEFT, anchor=tk.W)
        self.TimeLabel=ttk.Label(self.TimeFrame, text='Timestamping: ')
        .pack(side=tk.LEFT, anchor=tk.W)
        self.CANOCLabel=ttk.Label(self.CANOCFrame, text='CAN Channel: ')
        .pack(side=tk.LEFT, anchor=tk.W)

        self.COM_data=ttk.Label(self.COMFrame,
textvariable=self.COMVar).pack(side=tk.LEFT, anchor=tk.W)
        self.Baud_data=ttk.Label(self.BaudFrame,
textvariable=self.BaudVar).pack(side=tk.LEFT, anchor=tk.W)
        self.CAN_data=ttk.Label(self.CANFrame,
textvariable=self.CANmode).pack(side=tk.LEFT, anchor=tk.W)
        self.Time_data=ttk.Label(self.TimeFrame,
textvariable=self.TimeStamp).pack(side=tk.LEFT, anchor=tk.W)
        self.CANOC_data=ttk.Label(self.CANOCFrame,
textvariable=self.CANOC).pack(side=tk.LEFT, anchor=tk.W)

        self.BaudVar.set(self.tin.baudrate)
        self.COMVar.set(self.tin.COMPort)
        self.CANmode.set(self.tin.CANMode)
        self.TimeStamp.set(self.tin.Timestamping)
        self.CANOC.set(self.tin.CANOC)

        self.COMFrame.pack(side=tk.TOP, anchor=tk.W, pady=(10,0))
        self.BaudFrame.pack(side=tk.TOP, anchor=tk.W)
        self.CANFrame.pack(side=tk.TOP, anchor=tk.W)
        self.CANOCFrame.pack(side=tk.TOP, anchor=tk.W)
        self.TimeFrame.pack(side=tk.TOP, anchor=tk.W)

    def update(self):
        self.BaudVar.set(self.tin.baudrate)
        self.CANmode.set(self.tin.CANMode)
        if self.tin.Timestamping == 0 or
self.tin.Timestamping=='Off':
            self.TimeStamp.set('Off')
        else:
            self.TimeStamp('On')
        self.CANOC.set(self.tin.CANOC)

    def __pack__(self, side, anchor):
        self.StatsFrame.pack(side=side, anchor=anchor)

class ReadMessageFrame():
    """Class to show lines of text using tk"""
    def __init__(self, frame, font=(0,
8,'normal'), width=120, maxlines=100):
        self.frame1 = ttk.Frame(frame)
        self.first_line = ttk.Label(self.frame1, text='Output/Input
Console', justify=tk.LEFT, anchor=tk.W, relief=tk.SOLID)
        self.title=ttk.Label(self.frame1,
text='Direction\tMode\t\tID\t\tDLC\tD0\tD1\tD2\tD3\tD4\tD5\tD6\tD7')
        self.first_line.pack(side=tk.TOP, anchor=tk.W)
        self.title.pack(side=tk.TOP, anchor=tk.W)
        self.subtitle=ttk.Label(self.frame1,
text='Direction\tMode\t\tID\t\tDLC\tD0\tD1\tD2\tD3\tD4\tD5\tD6\tD7')

```

```

        self.subtitle.pack(side=tk.BOTTOM, anchor=tk.W)
        self.Text = tk.Text(self.frame1, font=font, width=width)
        self.Text.pack(side=tk.LEFT, anchor=tk.W, fill=tk.Y,
expand="yes")
        self.s_start = ttk.Scrollbar(self.frame1)#Scrollbar
        self.s_start.pack(side=tk.RIGHT, fill=tk.Y)
        self.s_start.config(command=self.Text.yview)
        self.Text.config(yscrollcommand=self.s_start.set)
        self.maxlines = maxlines
        self.lines = 0

    def writenewline(self,line):
        self.Text.config(state=tk.NORMAL)
        self.Text.insert(tk.END, line)
        self.Text.yview_pickplace('end')
        self.Text.config(state=tk.DISABLED)
        self.lines = self.lines + 1
        if self.lines >= self.maxlines:
            self.Text.config(state=tk.NORMAL)
            self.Text.delete(1.0, 2.0)
            self.Text.config(state=tk.DISABLED)
            self.lines = self.lines - 1

    def clearscreen(self):
        self.Text.config(state='normal')
        self.Text.delete(1.0, tk.END)
        self.Text.config(state='disabled')
        self.lines = 0

    def __pack__(self, side, anchor):
        self.frame1.pack(side = side,expand=1, pady=1,
padx=1,fill='x', anchor=anchor)

class MenuCANMessage():
    """Class to show a multi entry menu using tk"""
    def __init__(self,frame, monitor, usplib):

        #Message sender
        self.monitor=monitor
        self.usplib=usplib
        self.masterframe=frame

        default='0x0'

        self.frameMain=ttk.Frame(self.masterframe)

        self.frame1 = ttk.Frame(self.frameMain, width=80)

        self.labelframe1=ttk.Label(self.frame1, text="Message
sender", relief=tk.SOLID).pack(side=tk.TOP, anchor=tk.N)

        self.iframe2 = ttk.Frame(self.frame1)
        ttk.Label(self.iframe2, text='Id').pack(side=tk.LEFT, padx=1)
        self.iden = tk.StringVar()

```

```

        ttk.Entry(self.iframe2, textvariable=self.iden,
width=10).pack(side=tk.LEFT, padx=1)
        self.iden.set(default)

        ttk.Label(self.iframe2, text='DLC').pack(side= tk.LEFT,
padx=1)
        self.DLCvar=tk.StringVar()

self.DLCBox=ttk.Combobox(self.iframe2, textvariable=self.DLCvar, values=[ '0
','1','2','3','4','5','6','7','8'], width=2)
        self.DLCBox.pack(side=tk.LEFT)
        self.DLCBox.set('1')
        self.DLCBox.bind("<<ComboboxSelected>>", lambda e:
self.updateDisactiveEntry())

        self.iframe2.pack(side=tk.TOP, expand=1, pady=(5,1), padx=1,
fill='x')

        self.iframe3 = ttk.Frame(self.frame1)

        ttk.Label(self.iframe3, text='D0').pack(side=tk.LEFT, padx=1)
        self.D0 = tk.StringVar()
        self.D0Entry=ttk.Entry(self.iframe3, textvariable=self.D0,
width=4)
        self.D0Entry.pack(side=tk.LEFT, padx=1)
        self.D0.set(default)

        self.D1 = tk.StringVar()
        self.D1Entry=ttk.Entry(self.iframe3, textvariable=self.D1,
width=4)
        self.D1Entry.pack(side=tk.RIGHT, padx=1)
        self.D1.set(default)
        ttk.Label(self.iframe3, text='D1').pack(side=tk.RIGHT,
padx=1)
        self.iframe3.pack(side = tk.TOP, expand=1, pady=1,
padx=1,fill='x')

        self.iframe4 = ttk.Frame(self.frame1)

        ttk.Label(self.iframe4, text='D2').pack(side=tk.LEFT, padx=1)
        self.D2 = tk.StringVar()
        self.D2Entry=ttk.Entry(self.iframe4, textvariable=self.D2,
width=4)
        self.D2Entry.pack(side=tk.LEFT, padx=1)
        self.D2.set(default)

        self.D3 = tk.StringVar()
        self.D3Entry=ttk.Entry(self.iframe4, textvariable=self.D3,
width=4)
        self.D3Entry.pack(side=tk.RIGHT, padx=1)
        self.D3.set(default)
        ttk.Label(self.iframe4, text='D3').pack(side=tk.RIGHT,
padx=1)
        self.iframe4.pack(side = tk.TOP, expand=1, pady=1,
padx=1,fill='x')

        self.iframe5 = ttk.Frame(self.frame1)

```

```

        ttk.Label(self.iframe5, text='D4').pack(side=tk.LEFT, padx=1)
        self.D4 = tk.StringVar()
        self.D4Entry=ttk.Entry(self.iframe5, textvariable=self.D4,
width=4)
            self.D4Entry.pack(side=tk.LEFT, padx=1)
            self.D4.set(default)

            self.D5 = tk.StringVar()
            self.D5Entry=ttk.Entry(self.iframe5, textvariable=self.D5,
width=4)
            self.D5Entry.pack(side=tk.RIGHT, padx=1)
            self.D5.set(default)
            ttk.Label(self.iframe5, text='D5').pack(side=tk.RIGHT,
padx=1)
            self.iframe5.pack(side = tk.TOP, expand=1, pady=1,
padx=1,fill='x')

            self.iframe6 = ttk.Frame(self.frame1)

            ttk.Label(self.iframe6, text='D6').pack(side=tk.LEFT, padx=1)
            self.D6 = tk.StringVar()
            self.D6Entry=ttk.Entry(self.iframe6, textvariable=self.D6,
width=4)
            self.D6Entry.pack(side=tk.LEFT, padx=1)
            self.D6.set(default)

            self.D7 = tk.StringVar()
            self.D7Entry=ttk.Entry(self.iframe6, textvariable=self.D7,
width=4)
            self.D7Entry.pack(side=tk.RIGHT,padx=1)
            self.D7.set(default)
            ttk.Label(self.iframe6, text='D7').pack(side=tk.RIGHT,
padx=1)
            self.iframe6.pack(side = tk.TOP, expand=1, pady=1,
padx=1,fill='x')

            self.MessageMode1=ttk.Frame(self.frame1)
            self.MessageMode2=ttk.Frame(self.frame1)
            self.varMode1=tk.BooleanVar()
            self.radio1 = ttk.Radiobutton(self.MessageMode1,
text="Normal", variable=self.varMode1, value=True,
command=self.updateDisactiveEntry).pack(anchor=tk.W, side=tk.LEFT)
            self.radio2 = ttk.Radiobutton(self.MessageMode1, text="RTR",
variable=self.varMode1, value=False,
command=self.updateDisactiveEntry).pack(anchor=tk.W, side=tk.LEFT)
            self.varMode1.set(True)

            self.varMode2=tk.BooleanVar()
            self.radio3 = ttk.Radiobutton(self.MessageMode2,
text="Standard", variable=self.varMode2, value=True).pack(anchor=tk.W,
side=tk.LEFT)
            self.radio4 = ttk.Radiobutton(self.MessageMode2,
text="Extended", variable=self.varMode2, value=False).pack(anchor=tk.W,
side=tk.LEFT)
            self.varMode2.set(True)

            self.MessageMode1.pack(anchor=tk.W, side=tk.TOP)
            self.MessageMode2.pack(anchor=tk.W, side=tk.TOP)

```

```

        self.iframe7 = ttk.Frame(self.frame1)
        self.button1 = ttk.Button(self.iframe7, text = "Send
Message", command=self.sendMessage)
        self.button1.pack()
        self.iframe7.pack(side = tk.TOP, expand=1, pady=1,
padx=1, fill='x')

self.listEntries=[self.D0Entry,self.D1Entry,self.D2Entry,self.D3Entry,sel
f.D4Entry,self.D5Entry,self.D6Entry,self.D7Entry]

self.listVarEntries=[self.D0,self.D1,self.D2,self.D3,self.D4,self.D5,self
.D6,self.D7]

        self.updateDisactiveEntry()

        self.frame1.pack(side=tk.TOP, anchor=tk.N)

#AskStatesButtons

        self.frame2=ttk.Frame(self.frameMain, width=80)
        self.labelframe2=ttk.Label(self.frame2, text='USBTin
Commands', relief=tk.SOLID).pack(side=tk.TOP, anchor=tk.N, pady=(45,0))
        self.askHWButton=ttk.Button(self.frame2, text='HW Version',
command=self.askHW).pack(side=tk.TOP, anchor=tk.N, pady=(5,0))
        self.askFWButton=ttk.Button(self.frame2, text="FW Version",
command=self.askFW).pack(side=tk.TOP, anchor=tk.N)
        self.askUSBButton=ttk.Button(self.frame2, text="USB Status",
command=self.askUSB).pack(side=tk.TOP, anchor=tk.N)
        self.frame2.pack(side=tk.TOP, anchor=tk.N)

def __pack(self, side, anchor):
    self.frameMain.pack(side = side, anchor=anchor, expand=1,
pady=1, padx=1, fill='x')

def askHW (self):
    if self.usplib.CANOC=="Closed":
        self.monitor.writenewline('Hardware Version:
'+self.usplib.askHWversion()+'\n'+ '\n')
    else:
        line="CAN Channel is opened.\n Please close it to\n
proceed with this action."
        TopLevelMsg(self.frameMain,line, False)
def askFW (self):
    if self.usplib.CANOC=="Closed":
        self.monitor.writenewline('Firmware Version:
'+self.usplib.askFWversion()+'\n'+ '\n')
    else:
        line="CAN Channel is opened.\n Please close it to\n
proceed with this action."
        TopLevelMsg(self.frameMain,line, False)
def askUSB (self):
    def eval01(num_01):
        if num_01=='0':
            return 'OK! '+ '\n'
        else:

```

```

        return 'Error Warning! +' + '\n'

    if self.usplib.CANOC=="Closed":
        USBstat=self.usplib.askUSBstatus()
        line0=4*' \t'+'USB
Status +' + '\n' + 4*' \t' + '----- +' + '\n'
        line1='Status Flag Byte: 0b' +USBstat+'\n'
        line2='Bit 2 (Error Warning) -----'
'+eval01(USBstat[2])
        line3='Bit 3 (Data Overrun) -----'
'+eval01(USBstat[3])
        line4='Bit 5 (Error-Passive) -----'
'+eval01(USBstat[5])
        line5='Bit 7 (Bus Error) -----'
'+eval01(USBstat[7])
        line=line0+line1+line2+line3+line4+line5+'\n'*3
        self.monitor.writenewline(line)
    else:
        line="CAN Channel is opened.\n Please close it to\n proceed with this action."
        TopLevelMsg(self.frameMain,line,False)

    def updateDisactiveEntry(self):
        if self.varModel.get()==False:
            for EntryDis in self.listEntries:
                EntryDis.delete(0,tk.END)
                EntryDis.config(state=tk.DISABLED)
        else:
            eval=int(self.DLCBox.get())

            for i1 in range(0,eval):
                self.listEntries[i1].config(state=tk.NORMAL)
                self.listVarEntries[i1].set('0x0')

            for i2 in range(eval,8):
                self.listEntries[i2].delete(0,tk.END)
                self.listEntries[i2].config(state=tk.DISABLED)

    def sendMessage(self):
        if self.usplib.CANOC=="Open":
            id=self.iden.get()[2:]
            dlc=self.DLCvar.get()

            data=[]
            B0=self.D0.get()[2:]
            data.append(B0)
            B1=self.D1.get()[2:]
            data.append(B1)
            B2=self.D2.get()[2:]
            data.append(B2)
            B3=self.D3.get()[2:]
            data.append(B3)
            B4=self.D4.get()[2:]
            data.append(B4)
            B5=self.D5.get()[2:]
            data.append(B5)
            B6=self.D6.get()[2:]
            data.append(B6)

```



```

        B7=self.D7.get()[2:]
        data.append(B7)

        if self.varMode1.get()==True: data_list=data[:int(dlc)]
        else: data_list=[]

        normal=self.varMode1.get()
        standard=self.varMode2.get()

        if normal:
            if standard:
                self.usblib.canWrite(id, int(dlc), data_list,
0x0002)
                line='MSG-
T\t\t'+'Standard'+'\t\t'+0x+id.zfill(3)+'\t\t'+dlc
                for ibyte in data_list:
                    line=line+'\t0x'+ibyte.zfill(2)
                self.monitor.writenewline(line+'\n')

            else:
                self.usblib.canWrite(id, int(dlc), data_list,
0x0004)
                line='MSG-
T\t\t'+'Extended'+'\t\t'+0x+id.zfill(8)+'\t\t'+dlc
                for ibyte in data_list:
                    line=line+'\t0x'+ibyte.zfill(2)
                self.monitor.writenewline(line+'\n')

            else:
                if standard:
                    self.usblib.canWrite(id, int(dlc), data_list,
0x0002, 0x0001)
                    line='MSG-T\t\t'+'Standard
RTR'+'\t\t'+0x+id.zfill(3)+'\t\t'+dlc+'\t'
                    self.monitor.writenewline(line+'\n')
                else:
                    self.usblib.canWrite(id, int(dlc), data_list,
0x0004, 0x0001)
                    line='MSG-T\t\t'+'Extended
RTR'+'\t\t'+0x+id.zfill(8)+'\t\t'+dlc+'\t'
                    self.monitor.writenewline(line+'\n')

            else:
                TopLevelMsg(self.masterframe, "CAN Channel is
closed.\nPlease open CAN Channel in transmitter node\n and also in
receiver node.", False)

```

  

```

class TopLevelMsg():
    def __init__(self, master, message, rem_master):
        self.master=master
        self.rem_master=rem_master

        self.MsgWin=tk.Toplevel(master=master)
        self.frame=ttk.Frame(self.MsgWin, width=384, height=162)

```



```

        ttk.Label(self.frame, text=message,
justify=tk.CENTER).pack(side=tk.TOP)

        ttk.Button(self.frame, text='OK',
command=self.closeTop).pack(side=tk.TOP)

        self.frame.pack()

        self.MsgWin.lift()

    def closeTop(self):
        self.MsgWin.destroy()
        if self.rem_master: self.master.destroy()

class NewCOMTop():
    def __init__(self, root, framepack, mainClass):
        self.NewWin=tk.Toplevel(master=root)
        self.frameWin=ttk.Frame(self.NewWin)

        self.mainClass=mainClass

        self.framepack=framepack
        self.root=root

        self.COMVar=tk.StringVar()
        ports=serial_ports()

COMBox=ttk.Combobox(self.frameWin, textvariable=self.COMVar, values=ports)
COMBox.pack(side=tk.LEFT)

    if ports!=[]:
        COMBox.set(ports[0])
    else:
        COMBox.set('No ports available')

    self.BAUDVar=tk.StringVar()

BAUDBox=ttk.Combobox(self.frameWin, textvariable=self.BAUDVar, values=["BAU
D_10K", "BAUD_20K", "BAUD_50K", "BAUD_100K", "BAUD_125K", "BAUD_250K",
"BAUD_500K", "BAUD_800K", "BAUD_1M"])
BAUDBox.pack(side=tk.LEFT)
BAUDBox.set("BAUD_125K")

    self.CANModeVar=tk.StringVar()

CANModeBox=ttk.Combobox(self.frameWin, textvariable=self.CANModeVar,
values=["Standard Mode", "Listen-Only Mode", "Loopback Mode"])
CANModeBox.pack(side=tk.LEFT)
CANModeBox.set("Standard Mode")

    self.OKButton=ttk.Button(self.frameWin, text='Accept',
command=self.openInstance).pack(side=tk.BOTTOM)

        self.frameWin.pack()

```

```

        self.NewWin.lift()

    def openInstance(self):
        port=self.COMVar.get()
        baud=self.BAUDVar.get()
        mode=self.CANModeVar.get()

        if mode == "Standard Mode": mode="Normal"
        elif mode == "Listen-Only Mode": mode="Listen-Only"
        elif mode == "Loopback Mode": mode="Loopback"

        if port=='No ports available':
            TopLevelMsg(self.NewWin,'No ports available to
open.'+'\n'+ 'Please connect a USBTin to a serial port.', True)

    else:
        self.mainClass.welcome.destroy()
        COMInstance=USBTinFrame(self.framepack, port, baud, mode,
self.mainClass)
        COMInstance.MainFrame.pack(side=tk.TOP, anchor=tk.NW)
        self.mainClass.opened_ports.append(port)
        print(self.mainClass.opened_ports)
        self.NewWin.destroy()

    class WelcomeMessage():
        def __init__(self, parentFrame):
            self.frame=parentFrame
            self.textFrame=ttk.Frame(self.frame, width=768, height=576)
            self.labelXTitle=ttk.Label(self.textFrame, text="CANUSBviewer -
USBTin GUI", foreground='blue', font=("Arial",20),
relief=tk.SUNKEN).pack(side=tk.TOP,anchor=tk.W)
            photo=tk.PhotoImage(file='USBTin_Image.gif')
            self.Labelimg=ttk.Label(self.textFrame, image=photo)
            self.Labelimg.photo=photo
            self.Labelimg.pack(side=tk.TOP)
            self.LabelWelc=ttk.Label(self.textFrame, text="Please open Serial
Port.\nClick 'New' Button on Menu Bar.", foreground="black",
font=("Helvetica",12)).pack(side=tk.TOP, anchor=tk.W)
            self.textFrame.pack()

        def destroy(self):
            self.textFrame.destroy()

    class VerticalScrolledFrame(ttk.Frame):
        """A pure Tkinter scrollable frame that actually works!
        * Use the 'interior' attribute to place widgets inside the scrollable
frame
        * Construct and pack/place/grid normally
        * This frame only allows vertical scrolling
"""

        def __init__(self, parent, *args, **kw):
            ttk.Frame.__init__(self, parent, *args, **kw)

            # create a canvas object and a vertical scrollbar for scrolling
it

```

```

vscrollbar = ttk.Scrollbar(self, orient=tk.VERTICAL)
vscrollbar.pack(fill=tk.Y, side=tk.RIGHT, expand=tk.FALSE)
canvas = tk.Canvas(self, bd=0,
highlightthickness=0, yscrollcommand=vscrollbar.set)
canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=tk.TRUE)
vscrollbar.config(command=canvas.yview)

# reset the view
canvas.xview_moveto(0)
canvas.yview_moveto(0)

# create a frame inside the canvas which will be scrolled with it
self.interior = interior = ttk.Frame(canvas)
interior_id = canvas.create_window(0, 0, window=interior,
anchor=tk.NW)

# track changes to the canvas and frame width and sync them,
# also updating the scrollbar
def _configure_interior(event):
    # update the scrollbars to match the size of the inner frame
    size = (interior.winfo_reqwidth(),
interior.winfo_reqheight())
    canvas.config(scrollregion="0 0 %s %s" % size)
    if interior.winfo_reqwidth() != canvas.winfo_width():
        # update the canvas's width to fit the inner frame
        canvas.config(width=interior.winfo_reqwidth())
    if interior.winfo_reqheight() != canvas.winfo_height():
        canvas.config(height=interior.winfo_reqheight())
interior.bind('<Configure>', _configure_interior)

def _configure_canvas(event):
    if interior.winfo_reqwidth() != canvas.winfo_width():
        # update the inner frame's width to fill the canvas
        canvas.itemconfigure(interior_id,
width=canvas.winfo_width())
    canvas.bind('<Configure>', _configure_canvas)

return

class App():
    def __init__(self):

        self.opened_ports=[]
        self.opened_instance={}

        self.root=tk.Tk()
        self.root.title('CANViewer')

        self.Canvas=VerticalScrolledFrame(parent=self.root)
        self.MainFrame=self.Canvas.interior

        self.welcome=WelcomeMessage(self.MainFrame)

        #MenuBar
        self.menubar = tk.Menu(self.root)
        self.menubar.add_command(label="New", command=self.popTopLevel)
        self.root.config(menu=self.menubar)

```



```
self.Canvas.pack(side=tk.TOP)

self.root.protocol("WM_DELETE_WINDOW", self.closeWin)
self.root.mainloop()

def popTopLevel(self):
    NewCOMTop(self.root, self.MainFrame, self)

def closeWin(self):
    while self.opened_instance != {}:
        key_list = list(self.opened_instance.keys())
        self.opened_instance[key_list[0]].closeInstance()
    self.root.destroy()

if __name__ == "__main__":
    App()
```