Master Thesis
Master's degree in Industrial Engineering

Development of a CAN-FD Sniffer using Python

ANNEX

Author: Joaquín Cortés Fuentes
Director: Juan Manuel Moreno Eguilaz
Call: June 2019

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

DEVELOPMENT OF A CAN-FD SNIFFER USING PYTHON - ANNEX

JOAQUÍN CORTÉS FUENTES

Master's degree in Industrial Engineering
Electronic Engineering
Universitat Politècnica de Catalunya
Escola Tècnica Superior d'Enginyeria Industrial de Barcelona
(ETSEIB)

June 2019

Contents

A Code

A.1 canfdlib

```
# Author: Joaquin Cortes
#
# Permission is hereby granted, free of charge, to any person
    obtaining a copy
# of this software and associated documentation files (the "Software
    "), to deal
# in the Software without restriction, including without limitation
    the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/
    or sell
# copies of the Software, and to permit persons to whom the Software
     is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be
    included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
    EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
    MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT
    SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR
    OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
    ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
    DEALINGS IN
# THE SOFTWARE.

import ft # modified Adafruit SPI
from random import randint
from constants import *
from classes import *
import binascii

def set_bit(v, index, x):
    """Set the index:th bit of v to 1 if x is truthy, else to 0, and
     return the new value."""
    mask = 1 << index  # Compute mask, an integer with just bit '
    index' set.
    v &= ~mask  # Clear the bit indicated by the mask (if x is False
    )
    if x:
        v |= mask  # If x was True, set the bit indicated by the
    mask.
    return v  # Return the result, we're done.
```

```
##########################
#### Class definition ####
##########################

class CANFD_SPI(ft.SPI):

    def __init__(self, ft232h, cs, max_speed_hz, mode, bitorder,
    SPI_DEFAULT_BUFFER_LENGTH, SPI_MAX_BUFFER_LENGTH,
                 SPI_BAUDRATE, verbose=False):
        super(CANFD_SPI, self).__init__(ft232h, cs, max_speed_hz,
    mode, bitorder)

        self.SPI_DEFAULT_BUFFER_LENGTH = SPI_DEFAULT_BUFFER_LENGTH
        self.SPI_MAX_BUFFER_LENGTH = SPI_MAX_BUFFER_LENGTH
        self.SPI_BAUDRATE = SPI_BAUDRATE

        self.can_config = REG_CAN_CONFIG()
        self.opMode = NORMAL_MODE
        self.state = "idle"  # APP_STATE_INIT
        self.clk = CAN_SYSCLK_40M
        self.txFromFlash = True
        self.switchChanged = True
        self.ramInitialized = False
        self.selectedBitTime = CAN_500K_4M #CAN_500K_4M  #
    CAN_500K_2M

        self.rxFlags = CAN_RX_FIFO_NO_EVENT
        self.txFlags = CAN_RX_FIFO_NO_EVENT
        self.errorFlags = CAN_ERROR_FREE_STATE

        self.txConfig = CAN_TX_FIFO_CONFIG()
        self.rxConfig = CAN_RX_FIFO_CONFIG()

        self.txObj = CAN_TX_MSGOBJ()
        self.rxObj = CAN_RX_MSGOBJ()

        self.txCounter = 0

        self.transmitBuffer = []
        self.receiveBuffer = []

        self.txchannel = CAN_FIFO_CH2
        self.rxchannel = CAN_FIFO_CH1

        self.txdlc = CAN_DLC_64

        self.verbose=verbose

    def initialize(self):
        # Initialize
        self.reset()
        if self.verbose:
            print("Resetting...")
        self.eccEnable()
        if not self.ramInitialized:
            self.ramInit(0xFF)
            self.ramInitialized = True

        # configure device
        self.configureObjectReset()
        self.can_config.IsoCrcEnable = 1
```

```
        self.can_config.StoreInTEF = 0  # 0?
        self.can_config.TXQEnable = 0  # should be 0?
        self.configure()

        # setup TX FIFO
        self.transmitChannelConfigureObjectReset()
        self.txConfig.FifoSize = 0
        self.txConfig.PayLoadSize = CAN_PLSIZE_64
        self.txConfig.TxPriority = 0
        self.transmitChannelConfigure()
        self.txCounter = 0

        # setup RX FIFO

        self.receiveChannelConfigureObjectReset()
        self.rxConfig.FifoSize = 15
        self.rxConfig.PayLoadSize = CAN_PLSIZE_64
        self.rxConfig.RxTimeStampEnable = 1
        self.receiveChannelConfigure()

        # write 0 to filters (FIFOLINK)
        self.writeByte(0x1d0, 0x81)

        # Setup bit time
        self.bitTimeConfigure(self.clk, self.selectedBitTime)
        self.operationModeSelect(NORMAL_MODE)
        #self.state = APP_STATE_TEST_RAM_ACCESS

    # SPI Access Function
    def reset(self):
        spiTransmitBuffer = []
        spiTransmitBuffer.append(cINSTRUCTION_RESET << 4)
        spiTransmitBuffer.append(0)
        self._assert_cs()
        self.write(spiTransmitBuffer)
        self._deassert_cs()

    def readByte(self, address):
        spiTransmitBuffer = []
        spiTransmitBuffer.append((cINSTRUCTION_READ << 4) + ((
address >> 8) & 0xF))
        spiTransmitBuffer.append(address & 0xFF)
        self._assert_cs()
        self.write(spiTransmitBuffer)
        result = self.read(1)
        self._deassert_cs()
        return int(binascii.hexlify(result), 16)

    def readWord(self, address):
        spiTransmitBuffer = []
        spiTransmitBuffer.append((cINSTRUCTION_READ << 4) + ((
address >> 8) & 0xF))
        spiTransmitBuffer.append(address & 0xFF)
        self._assert_cs()
        self.write(spiTransmitBuffer)
        rx = self.read(4)
        self._deassert_cs()
        word = 0
        return int(binascii.hexlify(rx), 16)

    def readByteArray(self, address, length):
```

```
        spiTransmitBuffer = []
        spiTransmitBuffer.append((cINSTRUCTION_READ << 4) + ((
    address >> 8) & 0xF))
        spiTransmitBuffer.append(address & 0xFF)
        self._assert_cs()
        self.write(spiTransmitBuffer)
        response = self.read(length)
        self._deassert_cs()
        result = [byte for byte in response]
        return result

    def readWordArray(self, address, length):
        spiTransmitBuffer = []
        spiTransmitBuffer.append((cINSTRUCTION_READ << 4) + ((
    address >> 8) & 0xF))
        spiTransmitBuffer.append(address & 0xFF)
        self._assert_cs()
        self.write(spiTransmitBuffer)
        rx = binascii.hexlify(self.read(4 * length))
        self._deassert_cs()
        w = [int(rx[i:i + 8], 16) for i in xrange(0, len(rx), 8)]
        return w

    def writeByte(self, address, data):
        spiTransmitBuffer = []
        spiTransmitBuffer.append((cINSTRUCTION_WRITE << 4) + ((
    address >> 8) & 0xF))
        spiTransmitBuffer.append(address & 0xFF)
        spiTransmitBuffer.append(data)
        self._assert_cs()
        self.write(spiTransmitBuffer)
        self._deassert_cs()

    def writeWord(self, address, data):
        spiTransmitBuffer = []
        spiTransmitBuffer.append((cINSTRUCTION_WRITE << 4) + ((
    address >> 8) & 0xF))
        spiTransmitBuffer.append(address & 0xFF)

        # divide data in byte chunks
        data = [int(hex(int(data) >> i & 0xff).replace('L', ''), 16)
     for i in (24, 16, 8, 0)]
        spiTransmitBuffer = spiTransmitBuffer + data
        self._assert_cs()
        self.write(spiTransmitBuffer)
        self._deassert_cs()

    def writeByteArray(self, address, data):
        spiTransmitBuffer = []
        spiTransmitBuffer.append((cINSTRUCTION_WRITE << 4) + ((
    address >> 8) & 0xF))
        spiTransmitBuffer.append(address & 0xFF)
        spiTransmitBuffer = spiTransmitBuffer + data
        self._assert_cs()
        self.write(spiTransmitBuffer)
        self._deassert_cs()

    def writeByteArrayCRC(self, address, data, fromram=False):
        spiTransmitBuffer = []
        spiTransmitBuffer.append((cINSTRUCTION_WRITE_CRC << 4) + ((
    address >> 8) & 0xF))
```

```
        spiTransmitBuffer.append(address & 0xFF)
        if fromram:
            spiTransmitBuffer.append(len(data) >> 2)
        else:
            spiTransmitBuffer.append(len(data))
        spiTransmitBuffer = spiTransmitBuffer + data
        crcResult = self.calculateCRC16(spiTransmitBuffer)
        spiTransmitBuffer.append((crcResult >> 8) & 0xFF)
        spiTransmitBuffer.append(crcResult & 0xFF)
        self._assert_cs()
        self.write(spiTransmitBuffer)
        self._deassert_cs()

    def writeWordArray(self, address, data):
        spiTransmitBuffer = []
        spiTransmitBuffer.append((cINSTRUCTION_WRITE << 4) + ((
address >> 8) & 0xF))
        spiTransmitBuffer.append(address & 0xFF)
        for word in data:
            # divide data in byte chunks
            data = [int(hex(word >> i & 0xff).replace('L', ''), 16)
for i in (24, 16, 8, 0)]
            spiTransmitBuffer = spiTransmitBuffer + data
        self._assert_cs()
        self.write(spiTransmitBuffer)
        self._deassert_cs()

    # Configuration
    def configure(self):
        ciCon_word = 0x60079804
        if self.verbose:
            print("CiCON Reset values: {}".format(hex(ciCon_word)))
        for i in range(24, 29):
            ciCon_word = set_bit(ciCon_word, i, (self.can_config.
DNetFilterCount >> i & 1))
        ciCon_word = set_bit(ciCon_word, 29, self.can_config.
IsoCrcEnable)
        ciCon_word = set_bit(ciCon_word, 30, self.can_config.
ProtocolExceptionEventDisable)
        ciCon_word = set_bit(ciCon_word, 16, self.can_config.
WakeUpFilterEnable)
        for j, i in enumerate(range(17, 19)):
            ciCon_word = set_bit(ciCon_word, i, (self.can_config.
WakeUpFilterTime >> j) & 1)
        ciCon_word = set_bit(ciCon_word, 21, self.can_config.
BitRateSwitchDisable)
        ciCon_word = set_bit(ciCon_word, 8, self.can_config.
RestrictReTxAttempts)
        ciCon_word = set_bit(ciCon_word, 9, self.can_config.
EsiInGatewayMode)
        ciCon_word = set_bit(ciCon_word, 10, self.can_config.
SystemErrorToListenOnly)
        ciCon_word = set_bit(ciCon_word, 11, self.can_config.
StoreInTEF)
        ciCon_word = set_bit(ciCon_word, 12, self.can_config.
TXQEnable)
        for j, i in enumerate(range(4, 8)):
            ciCon_word = set_bit(ciCon_word, i, (self.can_config.
TxBandWidthSharing >> j) & 1)

        if self.verbose:
```

```
            print("(CONFIG) CiCON word to write at {address}: {word}
".format(address=hex(cREGADDR_CiCON), word=(hex(ciCon_word))))
        self.writeWord(cREGADDR_CiCON, ciCon_word)
        if self.verbose:
            print("(CONFIG) CiCON word read after config: {}".format
(hex(self.readWord(cREGADDR_CiCON))))

    def configureObjectReset(self):
        self.can_config.DNetFilterCount = 0
        self.can_config.IsoCrcEnable = 1
        self.can_config.ProtocolExpectionEventDisable = 1
        self.can_config.WakeUpFilterEnable = 1
        self.can_config.WakeUpFilterTime = 0b11
        self.can_config.BitRateSwitchDisable = 0
        self.can_config.RestrictReTxAttempts = 0
        self.can_config.EsiInGatewayMode = 0
        self.can_config.SystemErrorToListenOnly = 0
        self.can_config.StoreInTEF = 1
        self.can_config.TXQEnable = 1
        self.can_config.TxBandWidthSharing = 0

    # Operating mode
    def operationModeSelect(self, opMode):
        for mode in (CONFIGURATION_MODE, opMode):
            byte = self.readByte(cREGADDR_CiCON + 3)
            byte = (0xF8 & byte) + mode
            if self.verbose:
                print("byte to write at {address} to select opMode:
{byte}".format(address=hex(cREGADDR_CiCON+3), byte=hex(byte)))
            self.writeByte(cREGADDR_CiCON + 3, byte)
            self.opMode = mode

    def operationModeGet(self):
        d = self.readByte(cREGADDR_CiCON + 2)
        if self.verbose:
            print("byte read at {address} to get opMode: {byte}".
format(address=hex(cREGADDR_CiCON + 2), byte=hex(d)))
        d = (d >> 5) & 0x7
        if self.verbose:
            print("opMode: {}".format(hex(d)))
        if d == NORMAL_MODE:
            return NORMAL_MODE
        elif d == SLEEP_MODE:
            return SLEEP_MODE
        elif d == INTERNAL_LOOPBACK_MODE:
            return INTERNAL_LOOPBACK_MODE
        elif d == EXTERNAL_LOOPBACK_MODE:
            return EXTERNAL_LOOPBACK_MODE
        elif d == LISTEN_ONLY_MODE:
            return LISTEN_ONLY_MODE
        elif d == CONFIGURATION_MODE:
            return CONFIGURATION_MODE
        elif d == CLASSIC_MODE:
            return CLASSIC_MODE
        elif d == RESTRICTED_MODE:
            return RESTRICTED_MODE
        else:
            return INVALID_MODE

    # CAN transmit
    def transmitChannelConfigure(self):
```

```
        cififocon_word = 0x00046000
        #address = cREGADDR_CiFIFOCON + (self.txchannel *
CiFIFO_OFFSET)
        if self.verbose:
            print("CiFIFOCON reset value: {}".format(hex(
cififocon_word)))
        ## DEBUG: modify cififocon to enable flags (???)
        #cififocon_word = set_bit(cififocon_word, 24, 1)
        #cififocon_word = set_bit(cififocon_word, 25, 1)
        #cififocon_word = set_bit(cififocon_word, 26, 1)

        cififocon_word = set_bit(cififocon_word, 30, self.txConfig.
RTREnable)
        cififocon_word = set_bit(cififocon_word, 31, 1) # txEnable
        for j, i in enumerate(range(8, 13)):
            cififocon_word = set_bit(cififocon_word, i, (self.
txConfig.TxPriority >> j) & 1)
        for j, i in enumerate(range(13, 15)):
            cififocon_word = set_bit(cififocon_word, i, (self.
txConfig.TxAttempts >> j) & 1)
        for j, i in enumerate(range(0, 5)):
            cififocon_word = set_bit(cififocon_word, i, (self.
txConfig.FifoSize >> j) & 1)
        for j, i in enumerate(range(5, 8)):
            cififocon_word = set_bit(cififocon_word, i, (self.
txConfig.PayLoadSize >> j) & 1)
        address = cREGADDR_CiFIFOCON + (self.txchannel *
CiFIFO_OFFSET)
        if self.verbose:
            print("(CONFIG) TX CiFIFOCON to write at {address}: {
word}".format(address=hex(address), word=hex(cififocon_word)))
        self.writeWord(address, cififocon_word)
        if self.verbose:
            print("(CONFIG) TX CiFIFOCON word read after config: {}"
.format(hex(self.readWord(address))))

    def transmitChannelConfigureObjectReset(self):
        self.txConfig.RTREnable = 0
        self.txConfig.TxPriority = 0
        self.txConfig.TxAttempts = 0b11
        self.txConfig.FifoSize = 0
        self.txConfig.PayLoadSize = 0

    def receiveChannelConfigure(self):
        if self.rxchannel == CAN_FIFO_CH0:
            return -100
        cififocon_word = 0x00046000
        if self.verbose:
            print("RX CiFIFOCON reset value: {}".format(hex(
cififocon_word)))
        #address = cREGADDR_CiFIFOCON + (self.rxchannel *
CiFIFO_OFFSET)
        cififocon_word = set_bit(cififocon_word, 29, self.rxConfig.
RxTimeStampEnable)
        cififocon_word = set_bit(cififocon_word, 31, 0)  # txEnable
        for j, i in enumerate(range(0, 5)):
            cififocon_word = set_bit(cififocon_word, i, (self.
rxConfig.FifoSize >> j) & 1)
        for j, i in enumerate(range(5, 8)):
            cififocon_word = set_bit(cififocon_word, i, (self.
rxConfig.PayLoadSize >> j) & 1)
```

```
        address = cREGADDR_CiFIFOCON + (self.rxchannel *
CiFIFO_OFFSET)
        if self.verbose:
            print("(CONFIG) RX CiFIFOCON to write at {address}: {
word}".format(address=hex(address), word=hex(cififocon_word)))
        self.writeWord(address, cififocon_word)
        if self.verbose:
            print("(CONFIG) RX CiFIFOCON word read after config: {}"
.format(hex(self.readWord(address))))

    def receiveChannelConfigureObjectReset(self):
        self.rxConfig.FifoSize = 0
        self.rxConfig.PayLoadSize = 0
        self.rxConfig.RxTimeStampEnable = 0

    def bitTimeConfigure(self, clk, selectedBitTime):
        if clk == CAN_SYSCLK_40M:
            self.bitTimeConfigureNominal40MHz(selectedBitTime)
            self.bitTimeConfigureData40MHz(selectedBitTime)

        else:
            print('CLK should be 40 MHz MHz')

        """
        elif clk == CAN_SYSCLK_20M:
            self.bitTimeConfigureNominal20MHz(selectedBitTime)
            self.bitTimeConfigureData20MHz(selectedBitTime)
        """

    # bitTime is 500K_2M

    def bitTimeConfigureNominal40MHz(self, selectedBitTime):
        ciNbtcfg_word = 0x0F0F3E00
        if self.verbose:
            print("CiNBTCFG reset value: {}".format(hex(
ciNbtcfg_word)))
        if selectedBitTime in all500k:
            for j, i in enumerate(range(24, 31)): # SJW
                ciNbtcfg_word = set_bit(ciNbtcfg_word, i, (15 >> j)
& 1)
            for j, i in enumerate(range(16, 22)): # TSEG2
                ciNbtcfg_word = set_bit(ciNbtcfg_word, i, (15 >> j)
& 1)
            for j, i in enumerate(range(8, 16)): #TSEG1
                ciNbtcfg_word = set_bit(ciNbtcfg_word, i, (62 >> j)
& 1)
            for j, i in enumerate(range(0, 8)): #BRP
                ciNbtcfg_word = set_bit(ciNbtcfg_word, i, 0)
        elif selectedBitTime in all250k:
            for j, i in enumerate(range(24, 31)): # SJW
                ciNbtcfg_word = set_bit(ciNbtcfg_word, i, (31 >> j)
& 1)
            for j, i in enumerate(range(16, 22)): # TSEG2
                ciNbtcfg_word = set_bit(ciNbtcfg_word, i, (31 >> j)
& 1)
            for j, i in enumerate(range(8, 16)): #TSEG1
                ciNbtcfg_word = set_bit(ciNbtcfg_word, i, (126 >> j)
 & 1)
            for j, i in enumerate(range(0, 8)): #BRP
                ciNbtcfg_word = set_bit(ciNbtcfg_word, i, 0)
        elif selectedBitTime in all1000k:
```

```
            for j, i in enumerate(range(24, 31)): # SJW
                ciNbtcfg_word = set_bit(ciNbtcfg_word, i, (7 >> j) &
 1)
            for j, i in enumerate(range(16, 22)): # TSEG2
                ciNbtcfg_word = set_bit(ciNbtcfg_word, i, (7 >> j) &
 1)
            for j, i in enumerate(range(8, 16)): #TSEG1
                ciNbtcfg_word = set_bit(ciNbtcfg_word, i, (30 >> j)
& 1)
            for j, i in enumerate(range(0, 8)): #BRP
                ciNbtcfg_word = set_bit(ciNbtcfg_word, i, 0)
        elif selectedBitTime == CAN_125K_500K:
            for j, i in enumerate(range(24, 31)): # SJW
                ciNbtcfg_word = set_bit(ciNbtcfg_word, i, (63 >> j)
& 1)
            for j, i in enumerate(range(16, 22)): # TSEG2
                ciNbtcfg_word = set_bit(ciNbtcfg_word, i, (63 >> j)
& 1)
            for j, i in enumerate(range(8, 16)): #TSEG1
                ciNbtcfg_word = set_bit(ciNbtcfg_word, i, (254 >> j)
 & 1)
            for j, i in enumerate(range(0, 8)): #BRP
                ciNbtcfg_word = set_bit(ciNbtcfg_word, i, 0)
        else:
            return -1
        if self.verbose:
            print("(CONFIG) CiNBTCFG to write at {address}: {word}".
format(address=hex(cREGADDR_CiNBTCFG), word=hex(ciNbtcfg_word)))
        self.writeWord(cREGADDR_CiNBTCFG, ciNbtcfg_word)
        if self.verbose:
            print("(CONFIG) CiNBTCFG read after config: {}".format(
hex(self.readWord(cREGADDR_CiNBTCFG))))

    def bitTimeConfigureData40MHz(self, selectedBitTime):
        ciDbtcfg_word = 0x03030E00
        ciTdc_word = 0x00100200
        if self.verbose:
            print("CiDBTCFG reset value: {}".format(hex(
ciDbtcfg_word)))
            print("CiTDC reset value: {}".format(hex(ciTdc_word)))
        ciTdc_word = set_bit(ciTdc_word, 15, 0)
        ciTdc_word = set_bit(ciTdc_word, 16, 1)
        tdcValue = 0
        if selectedBitTime == CAN_500K_1M:
            # Data BR
            for j, i in enumerate(range(24, 28)):  # SJW
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (7 >> j) &
 1)
            for j, i in enumerate(range(16, 20)):  # TSEG2
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (7 >> j) &
 1)
            for j, i in enumerate(range(8, 13)):  # TSEG1
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (30 >> j)
& 1)
            for j, i in enumerate(range(0, 8)): #BRP
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, 0)
            # SSP
            for j, i in enumerate(range(24, 30)):  # TDCV
                ciTdc_word = set_bit(ciTdc_word, i, (tdcValue >> j)
& 1)
            for j, i in enumerate(range(16, 9)):  # TDCO
```

```
            ciTdc_word = set_bit(ciTdc_word, i, (31 >> j) & 1)
    elif selectedBitTime == CAN_500K_2M:
        # Data BR
        for j, i in enumerate(range(24, 28)):  # SJW
            ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (3 >> j) &
 1)
        for j, i in enumerate(range(16, 20)):  # TSEG2
            ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (3 >> j) &
 1)
        for j, i in enumerate(range(8, 13)):  # TSEG1
            ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (14 >> j)
& 1)
        for j, i in enumerate(range(0, 8)): #BRP
            ciDbtcfg_word = set_bit(ciDbtcfg_word, i, 0)
        # SSP
        for j, i in enumerate(range(24, 30)):  # TDCV
            ciTdc_word = set_bit(ciTdc_word, i, (tdcValue >> j)
& 1)
        for j, i in enumerate(range(16, 9)):  # TDCO
            ciTdc_word = set_bit(ciTdc_word, i, (15 >> j) & 1)
    elif selectedBitTime == CAN_500K_3M:
        # Data BR
        for j, i in enumerate(range(24, 28)):  # SJW
            ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (2 >> j) &
 1)
        for j, i in enumerate(range(16, 20)):  # TSEG2
            ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (2 >> j) &
 1)
        for j, i in enumerate(range(8, 13)):  # TSEG1
            ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (8 >> j) &
 1)
        for j, i in enumerate(range(0, 8)): #BRP
            ciDbtcfg_word = set_bit(ciDbtcfg_word, i, 0)
        # SSP
        for j, i in enumerate(range(24, 30)):  # TDCV
            ciTdc_word = set_bit(ciTdc_word, i, (tdcValue >> j)
& 1)
        for j, i in enumerate(range(16, 9)):  # TDCO
            ciTdc_word = set_bit(ciTdc_word, i, (9 >> j) & 1)
    elif selectedBitTime in (CAN_500K_4M, CAN_1000K_4M):
        # Data BR
        for j, i in enumerate(range(24, 28)):  # SJW
            ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (1 >> j) &
 1)
        for j, i in enumerate(range(16, 20)):  # TSEG2
            ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (1 >> j) &
 1)
        for j, i in enumerate(range(8, 13)):  # TSEG1
            ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (6 >> j) &
 1)
        for j, i in enumerate(range(0, 8)): #BRP
            ciDbtcfg_word = set_bit(ciDbtcfg_word, i, 0)
        # SSP
        for j, i in enumerate(range(24, 30)):  # TDCV
            ciTdc_word = set_bit(ciTdc_word, i, (tdcValue >> j)
& 1)
        for j, i in enumerate(range(16, 9)):  # TDCO
            ciTdc_word = set_bit(ciTdc_word, i, (7 >> j) & 1)
    elif selectedBitTime == CAN_500K_5M:
        # Data BR
        for j, i in enumerate(range(24, 28)):  # SJW
```

```
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (1 >> j) &
 1)
            for j, i in enumerate(range(16, 20)):  # TSEG2
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (1 >> j) &
 1)
            for j, i in enumerate(range(8, 13)):  # TSEG1
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (4 >> j) &
 1)
            for j, i in enumerate(range(0, 8)): #BRP
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, 0)
            # SSP
            for j, i in enumerate(range(24, 30)):  # TDCV
                ciTdc_word = set_bit(ciTdc_word, i, (tdcValue >> j)
& 1)
            for j, i in enumerate(range(16, 9)):  # TDCO
                ciTdc_word = set_bit(ciTdc_word, i, (5 >> j) & 1)
        elif selectedBitTime == CAN_500K_6M7:
            # Data BR
            for j, i in enumerate(range(24, 28)):  # SJW
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (0 >> j) &
 1)
            for j, i in enumerate(range(16, 20)):  # TSEG2
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (0 >> j) &
 1)
            for j, i in enumerate(range(8, 13)):  # TSEG1
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (3 >> j) &
 1)
            for j, i in enumerate(range(0, 8)): #BRP
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, 0)
            # SSP
            for j, i in enumerate(range(24, 30)):  # TDCV
                ciTdc_word = set_bit(ciTdc_word, i, (tdcValue >> j)
& 1)
            for j, i in enumerate(range(16, 9)):  # TDCO
                ciTdc_word = set_bit(ciTdc_word, i, (4 >> j) & 1)
        elif selectedBitTime in (CAN_500K_8M, CAN_1000K_8M):
            # Data BR
            for j, i in enumerate(range(24, 28)):  # SJW
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (0 >> j) &
 1)
            for j, i in enumerate(range(16, 20)):  # TSEG2
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (0 >> j) &
 1)
            for j, i in enumerate(range(8, 13)):  # TSEG1
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (2 >> j) &
 1)
            for j, i in enumerate(range(0, 8)): #BRP
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, 0)
            # SSP
            for j, i in enumerate(range(24, 30)):  # TDCV
                ciTdc_word = set_bit(ciTdc_word, i, (1 >> j) & 1)
            for j, i in enumerate(range(16, 9)):  # TDCO
                ciTdc_word = set_bit(ciTdc_word, i, (3 >> j) & 1)
        elif selectedBitTime == CAN_500K_10M:
            # Data BR
            for j, i in enumerate(range(24, 28)):  # SJW
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (0 >> j) &
 1)
            for j, i in enumerate(range(16, 20)):  # TSEG2
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (0 >> j) &
 1)
```

```
            for j, i in enumerate(range(8, 13)):  # TSEG1
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (1 >> j) &
 1)
            for j, i in enumerate(range(0, 8)): #BRP
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, 0)
            # SSP
            for j, i in enumerate(range(24, 30)):  # TDCV
                ciTdc_word = set_bit(ciTdc_word, i, (0 >> j) & 1)
            for j, i in enumerate(range(16, 9)):  # TDCO
                ciTdc_word = set_bit(ciTdc_word, i, (2 >> j) & 1)
        elif selectedBitTime in (CAN_250K_500K, CAN_125K_500K):
            # Data BR
            for j, i in enumerate(range(24, 28)):  # SJW
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (7 >> j) &
 1)
            for j, i in enumerate(range(16, 20)):  # TSEG2
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (7 >> j) &
 1)
            for j, i in enumerate(range(8, 13)):  # TSEG1
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (30 >> j)
& 1)
            for j, i in enumerate(range(0, 8)): #BRP
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, 1)
            # SSP
            for j, i in enumerate(range(24, 30)):  # TDCV
                ciTdc_word = set_bit(ciTdc_word, i, (tdcValue >> j)
& 1)
            for j, i in enumerate(range(16, 9)):  # TDCO
                ciTdc_word = set_bit(ciTdc_word, i, (31 >> j) & 1)
            for j, i in enumerate(range(16, 18)):  # TDCO
                ciTdc_word = set_bit(ciTdc_word, i, (
CAN_SSP_MODE_OFF >> j) & 1)
        elif selectedBitTime == CAN_250K_833K:
            # Data BR
            for j, i in enumerate(range(24, 28)):  # SJW
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (4 >> j) &
 1)
            for j, i in enumerate(range(16, 20)):  # TSEG2
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (4 >> j) &
 1)
            for j, i in enumerate(range(8, 13)):  # TSEG1
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (17 >> j)
& 1)
            for j, i in enumerate(range(0, 8)): #BRP
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, 1)
            # SSP
            for j, i in enumerate(range(24, 30)):  # TDCV
                ciTdc_word = set_bit(ciTdc_word, i, (tdcValue >> j)
& 1)
            for j, i in enumerate(range(16, 9)):  # TDCO
                ciTdc_word = set_bit(ciTdc_word, i, (18 >> j) & 1)
            for j, i in enumerate(range(16, 18)):  # TDCO
                ciTdc_word = set_bit(ciTdc_word, i, (
CAN_SSP_MODE_OFF >> j) & 1)
        elif selectedBitTime == CAN_250K_1M:
            # Data BR
            for j, i in enumerate(range(24, 28)):  # SJW
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (7 >> j) &
 1)
            for j, i in enumerate(range(16, 20)):  # TSEG2
```

```
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (7 >> j) &
 1)
            for j, i in enumerate(range(8, 13)):  # TSEG1
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (30 >> j)
& 1)
            for j, i in enumerate(range(0, 8)): #BRP
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, 1)
            # SSP
            for j, i in enumerate(range(24, 30)):  # TDCV
                ciTdc_word = set_bit(ciTdc_word, i, (tdcValue >> j)
& 1)
            for j, i in enumerate(range(16, 9)):  # TDCO
                ciTdc_word = set_bit(ciTdc_word, i, (31 >> j) & 1)
        elif selectedBitTime == CAN_250K_1M5:
            # Data BR
            for j, i in enumerate(range(24, 28)):  # SJW
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (5 >> j) &
 1)
            for j, i in enumerate(range(16, 20)):  # TSEG2
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (5 >> j) &
 1)
            for j, i in enumerate(range(8, 13)):  # TSEG1
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (18 >> j)
& 1)
            for j, i in enumerate(range(0, 8)): #BRP
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, 1)
            # SSP
            for j, i in enumerate(range(24, 30)):  # TDCV
                ciTdc_word = set_bit(ciTdc_word, i, (tdcValue >> j)
& 1)
            for j, i in enumerate(range(16, 9)):  # TDCO
                ciTdc_word = set_bit(ciTdc_word, i, (19 >> j) & 1)
        elif selectedBitTime == CAN_250K_2M:
            # Data BR
            for j, i in enumerate(range(24, 28)):  # SJW
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (3 >> j) &
 1)
            for j, i in enumerate(range(16, 20)):  # TSEG2
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (3 >> j) &
 1)
            for j, i in enumerate(range(8, 13)):  # TSEG1
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (14 >> j)
& 1)
            for j, i in enumerate(range(0, 8)): #BRP
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, 0)
            # SSP
            for j, i in enumerate(range(24, 30)):  # TDCV
                ciTdc_word = set_bit(ciTdc_word, i, (tdcValue >> j)
& 1)
            for j, i in enumerate(range(16, 9)):  # TDCO
                ciTdc_word = set_bit(ciTdc_word, i, (15 >> j) & 1)
        elif selectedBitTime == CAN_250K_3M:
            # Data BR
            for j, i in enumerate(range(24, 28)):  # SJW
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (2 >> j) &
 1)
            for j, i in enumerate(range(16, 20)):  # TSEG2
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (2 >> j) &
 1)
            for j, i in enumerate(range(8, 13)):  # TSEG1
```

```
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (8 >> j) &
 1)
            for j, i in enumerate(range(0, 8)): #BRP
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, 0)
            # SSP
            for j, i in enumerate(range(24, 30)):  # TDCV
                ciTdc_word = set_bit(ciTdc_word, i, (tdcValue >> j)
& 1)
            for j, i in enumerate(range(16, 9)):  # TDCO
                ciTdc_word = set_bit(ciTdc_word, i, (9 >> j) & 1)
        elif selectedBitTime == CAN_250K_4M:
            # Data BR
            for j, i in enumerate(range(24, 28)):  # SJW
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (1 >> j) &
 1)
            for j, i in enumerate(range(16, 20)):  # TSEG2
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (1 >> j) &
 1)
            for j, i in enumerate(range(8, 13)):  # TSEG1
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, (6 >> j) &
 1)
            for j, i in enumerate(range(0, 8)): #BRP
                ciDbtcfg_word = set_bit(ciDbtcfg_word, i, 0)
            # SSP
            for j, i in enumerate(range(24, 30)):  # TDCV
                ciTdc_word = set_bit(ciTdc_word, i, (tdcValue >> j)
& 1)
            for j, i in enumerate(range(16, 9)):  # TDCO
                ciTdc_word = set_bit(ciTdc_word, i, (7 >> j) & 1)
        else:
            return -1
        if self.verbose:
            print("(CONFIG) CiDBTCFG to write at {address}: {word}".
format(address=hex(cREGADDR_CiDBTCFG), word=hex(ciDbtcfg_word)))
        self.writeWord(cREGADDR_CiDBTCFG, ciDbtcfg_word)
        if self.verbose:
            print("(CONFIG) CiTDC to write at {address}: {word}".
format(address=hex(cREGADDR_CiTDC), word=hex(ciTdc_word)))
        self.writeWord(cREGADDR_CiTDC, ciTdc_word)
        if self.verbose:
            print("(CONFIG) CiDBTCFG read after config: {}".format(
hex(self.readWord(cREGADDR_CiDBTCFG))))
            print("(CONFIG) CiTDC read after config: {}".format(hex(
self.readWord(cREGADDR_CiTDC))))

    def eccEnable(self):
        byte = self.readByte(cREGADDR_ECCCON)
        byte |= 0x01
        self.writeByte(cREGADDR_ECCCON, byte)
        print('ECC enabled')

    def eccDisable(self):
        byte = self.readByte(cREGADDR_ECCCON)
        byte |= ~0x01
        self.writeByte(cREGADDR_ECCCON, byte)
        print('ECC disabled')

    def ramInit(self, d):
        txd = [d for k in range(0, self.SPI_DEFAULT_BUFFER_LENGTH)]
        for i in range(cRAMADDR_START, cRAM_SIZE / self.
SPI_DEFAULT_BUFFER_LENGTH, self.SPI_DEFAULT_BUFFER_LENGTH):
```

```
            self.writeByteArray(i, txd)
        print('RAM initialized')

    def calculateCRC16(self, data):
        # data is a byte array
        init = CRCBASE
        for byte in data:
            index = (init >> 8) ^ byte
            init = ((init << 8) ^ crc16_table[index]) & oxFFFF
        return init


    def receiveMessageGet(self, nBytes):  # rxd, n):
        n = 0

        # get FIFO registers
        address = cREGADDR_CiFIFOCON + (self.rxchannel *
    CiFIFO_OFFSET)
        # fifoReg = [reverse(w) for w in self.readWordArray(address,
     3)]
        fifoReg = [w for w in self.readWordArray(address, 3)]
        if self.verbose:
            print("(RX GET) fifo registers read at {address}: {
    fiforeg}".format(address=hex(address), fiforeg=[hex(a) for a in
    fifoReg]))

        # check that is a receive buffer
        ciFifoCon_word = fifoReg[0]
        if int(ciFifoCon_word >> 31) & 1: #TxEnable:
            return -2
        # get status
        ciFifoSta_word = fifoReg[1]

        # get address
        ciFifoUa_word = int(fifoReg[2] >> 24) + int((fifoReg[2] >>
    8) & 0x0000FF00)
        if self.verbose:
            print("(RX GET) UA in RX CiFIFOUA: {}".format(hex(
    ciFifoUa_word)))
        address = ciFifoUa_word + cRAMADDR_START
        n = nBytes + 8
        if self.verbose:
            print("(RX GET) RAM ADDRESS to read RX FIFO: {}".format(
    hex(address)))
        if int(ciFifoCon_word >> 29) & 1: #.rx.RxTimeStampEnable:
            n += 4
        if n % 4:
            n = n + 4 - (n % 4)

        # read rxObj
        if n > MAX_MSG_SIZE:
            n = MAX_MSG_SIZE
        print("# bytes to read: {}".format(n))
        ba = self.readByteArray(address, n)
        if self.verbose:
            print("(RX GET) Byte array read at {address}: {arr}".
    format(address=hex(address), arr=ba))

        id_word = (ba[0] << 24) + (ba[1] << 16) + (ba[2] << 8) + ba
    [3]
        #word = int(word, 16)
```

```
        #self.rxObj.id.word = word

        ctrl_word = (ba[4] << 24) + (ba[5] << 16) + (ba[6] << 8) +
ba[7]
        #word = int(word, 16)
        #self.rxObj.ctrl.word = word

        rxd = [ba[i] for i in range(12, nBytes + 12)]

        if self.verbose:
            print("(RX GET) ID word read: {}".format(hex(id_word)))
            print("(RX GET) CTRL word read: {}".format(hex(ctrl_word
)))
            print("(RX GET) data read: {}".format(rxd))

        # UINC channel
        self.receiveChannelUpdate()
        #return id_word, ctrl_word, rxd
        dlc = int(ctrl_word >> 24) & 0xF
        print("RX dlc: {}".format(self.dlcToDataBytes(dlc)))

        return rxd[0:self.dlcToDataBytes(dlc)]

    def receiveChannelUpdate(self):
        # set UINC
        address = cREGADDR_CiFIFOCON + (self.rxchannel *
CiFIFO_OFFSET) + 1
        #byte = 0xFF & self.readByte(address)
        if self.verbose:
            print("(RX Update) address to increase UINC: {}".format(
hex(address)))
        self.writeByte(address, 1)  # FF00 ???

    def receiveChannelEventGet(self):
        if self.rxchannel == CAN_FIFO_CH0:
            return -100
        address = cREGADDR_CiFIFOSTA + (self.rxchannel *
CiFIFO_OFFSET)
        byte = self.readByte(address)
        if self.verbose:
            print("(RX Event Get) Byte read from {address}: {byte}".
format(address=hex(address), byte=hex(byte)))
        self.rxFlags = (byte & 0xFF) & CAN_RX_FIFO_ALL_EVENTS
        if self.verbose:
            print("(RX Event Get) Byte masked with 0x0F: {}".format(
hex(self.rxFlags)))

    def transmitChannelEventGet(self):
        if self.txchannel == CAN_FIFO_CH0:
            return -100
        ciFifoSta = REG_CiFIFOSTA()
        #ciFifoSta.word = 0
        address = cREGADDR_CiFIFOSTA + (self.txchannel *
CiFIFO_OFFSET)
        byte = self.readByte(address)
        if self.verbose:
            print("(TX Event Get) Byte read from {address}: {byte}".
format(address=hex(address), byte=hex(byte)))
        self.txFlags = (byte & 0xFF) & CAN_TX_FIFO_ALL_EVENTS
        print("(TX Event Get) Byte masked with 0x1F: {}".format(hex(
self.txFlags)))
```

```
def dlcToDataBytes(self, dlc):
    if dlc < CAN_DLC_12:
        return dlc
    else:
        return CAN_DLC_dict[dlc]

def transmitChannelLoad(self, nbytes, flush, txd):
    address = cREGADDR_CiFIFOCON + (self.txchannel *
CiFIFO_OFFSET)
    fifoReg = [int(w) for w in self.readWordArray(address, 3)]
    #print("fiforef array: {}".format([hex(a) for a in fifoReg])
)
    if self.verbose:
        print("(TX Load) fifo registers read at {address}: {
fiforeg}".format(address=hex(address),fiforeg=[hex(a) for a in
fifoReg]))
    ciFifoCon_word = fifoReg[0]
    if not (int(ciFifoCon_word >> 31) & 1):  #ciFifoCon.tx.
TxEnable:
        return -2
    dataBytesInObject = self.dlcToDataBytes(self.txObj.ctrl.DLC)
    if dataBytesInObject < nbytes:
        return -3
    ciFifoSta_word = fifoReg[1]
    ciFifoUa_word = fifoReg[2]

    #aux = int(ciFifoUa_word >> 16) & 0xFFF
    #print("UA 2: {}".format(hex(aux)))
    #userAddress = ciFifoUa_word & 0xFFF
    #print("UA: {}".format(userAddress))

    userAddress = int(ciFifoUa_word >> 24) + (int(ciFifoUa_word
>> 8) & 0x0000FF00)
    if self.verbose:
        print("(TX LOAD) TX userAddress: {}".format(hex(
userAddress)))

    userAddress += cRAMADDR_START
    print("(TX LOAD) TX userAddress with offset: {}".format(hex(
userAddress)))
    txBuffer = [0] * (nbytes + 8)

    #print("tx obj id word: {}".format(hex(self.txObj.id.word)))

    """
    for k in range(0, nbytes):
        if k < 4:
            txBuffer[k] = int((self.txObj.id.word >> 8 * k) & 0
xFF)
        elif 4 <= k < 8:
            txBuffer[k] = int((self.txObj.ctrl.word >> 8 * (k -
4)) & 0xFF)
        else:
            txBuffer[k] = txd[k - 8] & 0xFF
    """

    txBuffer[0] = int(self.txObj.id.SID & 0xFF)
    txBuffer[1] = (int(self.txObj.id.SID >> 8) & 0x7) + (int(
self.txObj.id.EID << 3) & 0x1F)
    txBuffer[2] = int(self.txObj.id.EID >> 5) & 0xFF
```

```
    txBuffer[3] = int(self.txObj.id.EID >> 13 & 0x1F)

    if self.verbose:
        print("(TX LOAD) ID and SID: {}".format([hex(a) for a in
 txBuffer[0:4]]))

    txBuffer[4] = int(self.txObj.ctrl.DLC + int(self.txObj.ctrl.
IDE << 4) + int(self.txObj.ctrl.RTR << 5) + int(self.txObj.ctrl.
BRS << 6) + int(self.txObj.ctrl.FDF << 7))
    txBuffer[5] = int(self.txObj.ctrl.ESI + int(self.txObj.ctrl.
SEQ << 1))
    # txBuffer[6] and txBuffer[7] are 0

    for k in range(8, nbytes + 8):
        txBuffer[k] = txd[k-8] & 0xFF

    n = 0
    if nbytes % 4:
        n = 4 - (nbytes % 4)
        i = nbytes + 8
        for j in range(0, n):
            txBuffer[i + 8 + j] = 0

    if self.verbose:
        print("(TX LOAD) txbuffer to send: {}".format(txBuffer))
    self.writeByteArray(userAddress, txBuffer)
    self.state = "idle"
    #print("(TX LOAD) CiFIFOCON TX before calling update status:
 {}".format(hex(self.readWord(address))))

    self.transmitChannelUpdate(flush)


def transmitChannelUpdate(self, flush):
    address = cREGADDR_CiFIFOCON + (self.txchannel *
CiFIFO_OFFSET)
    if self.verbose:
        print("(TX Update) fifocon 2nd byte address: {}".format(
hex(address + 1)))
        print("(TX Update) CiFIFOCON TX before updating status:
{}".format(hex(self.readWord(address))))
        print("(TX Update) CiFIFOSTA  before updating: {}".
format(hex(self.readWord(cREGADDR_CiFIFOSTA + (self.txchannel *
CiFIFO_OFFSET)))))

    #aux = set_bit(aux, 0, 1)
    aux = set_bit(0, 0, 1)
    if flush:
        aux = set_bit(aux, 1, 1)
    self.writeByte(address + 1, aux)
    #w = self.readWord(address - 1)
    ##w = set_bit(w, 16, 1)
    #w = set_bit(w, 17, 1)
    #self.writeWord(address-1, w)
    if self.verbose:
        print("(TX Update) byte written at {a}: {aux}".format(
aux=aux, a=hex(address + 1)))
        print("(TX Update) CiFIFOSTA  after updating: {}".format
(hex(self.readWord(cREGADDR_CiFIFOSTA + (self.txchannel *
CiFIFO_OFFSET)))))
```

```
            print("(TX Update) CiFIFOCON TX after updating status:
{}".format(hex(self.readWord(address))))
        # self.transmitChannelEventGet()

    ### DEBUG: TRY TO DIRECTLY WRITE STATUS IN RX FIFOCON
    #if self.operationModeGet() == INTERNAL_LOOPBACK_MODE:
        #self.writeByte(cREGADDR_CiFIFOCON + (self.rxchannel *
CiFIFO_OFFSET) + 1, 1)
        #self.receiveChannelUpdate()

def dataBytesToDlc(self, n):
    try:
        return CAN_DLC_inv_dict[n]
    except KeyError:
        return CAN_DLC_0

def ramTest(self):
    # verify R/W
    # txd = [0] * MAX_DATA_BYTES
    # rxd = [0] * MAX_DATA_BYTES
    for length in range(4, MAX_DATA_BYTES + 1, 4):
        txd = [(randint(0, RAND_MAX) & 0xFF) for e in range(0,
length)]
        # rxd = [0xFF] * length
        if self.verbose:
            print("Data written on RAM: {}".format(txd))
        self.writeByteArray(cRAMADDR_START, txd)
        rxd = self.readByteArray(cRAMADDR_START, length)
        if self.verbose:
            print("Data read on RAM: {}".format(rxd))
        # good = False
        for i in range(0, length):
            good = txd[i] == rxd[i]
            if not good:
                print("Data mismatch!")
                self.reset()
                return -1
    #self.state = APP_STATE_TEST_REGISTER_ACCESS
    self.reset()
    return 1

def registerTest(self):
    for length in range(1, MAX_DATA_BYTES + 1):
        txd = [(randint(0, RAND_MAX) & 0x7F) for e in range(0,
length)]
        self.writeByteArray(cREGADDR_CiFLTOBJ, txd)
        rxd = self.readByteArray(cREGADDR_CiFLTOBJ, length)
        if self.verbose:
            print("Data written on Registers: {}".format(txd))
            print("Data read on Registers: {}".format(rxd))
        # good = False
        for i in range(0, length):
            good = txd[i] == rxd[i]
            if not good:
                print("Data mismatch!")
                self.reset()
                return -1
    # self.state = APP_STATE_INIT_TXOBJ
    self.reset()
    return 1
```

```
    def initTxObj(self):
        self.txObj.id.word = 0
        self.txObj.ctrl.word = 0

        self.txObj.id.SID = 0  # TX_RESPONSE_ID
        self.txObj.id.EID = 0

        self.txObj.ctrl.BRS = 1
        self.txObj.ctrl.DLC = self.txdlc # CAN_DLC_64
        self.txObj.ctrl.FDF = 1
        self.txObj.ctrl.IDE = 0
        self.txObj.ctrl.RTR = 0
        self.txObj.ctrl.SEQ = 1

        self.state = "idle"  # APP_STATE_RECEIVE

    def transmitMessageTasks(self, txd):
        self.txObj.id.word = 0
        self.txObj.ctrl.word = 0

        self.txObj.id.SID = 0x0  #self.txCounter  # TX_RESPONSE_ID
        self.txObj.id.EID = 0
        self.txObj.ctrl.BRS = 1
        self.txObj.ctrl.DLC = self.txdlc #CAN_DLC_64
        self.txObj.ctrl.FDF = 1
        self.txObj.ctrl.IDE = 0
        self.txObj.ctrl.RTR = 0
        self.txObj.ctrl.SEQ = 1
        #txd = range(0, self.dlcToDataBytes(self.txObj.ctrl.DLC))

        self.transmitChannelEventGet()
        if self.verbose:
            print("TX flags before trying to write: {}".format(self.
txFlags))

        if self.txFlags & CAN_TX_FIFO_NOT_FULL_EVENT:
            dlc = self.dlcToDataBytes(self.txObj.ctrl.DLC)
            if dlc < len(txd):
                if self.verbose:
                    print("Message too long! Cropping message")
                txd = txd[0:dlc]
            elif dlc > len(txd):
                if self.verbose:
                    print("Message too short, adding 0s")
                while len(txd) != dlc:
                    txd.append(0)
            self.transmitChannelLoad(self.dlcToDataBytes(self.txObj.
ctrl.DLC), True, txd)
            self.txCounter += 1
            if self.txCounter > 0x7FF:
                self.txCounter = 0

    def receiveMessageTasks(self):
        self.receiveChannelEventGet()
        if self.verbose:
            print("RX flags before trying to read: {}".format(self.
rxFlags))
        if self.rxFlags & CAN_RX_FIFO_NOT_EMPTY_EVENT:
        #if True:
            #rxd = self.receiveMessageGet(MAX_DATA_BYTES)
            rxd = self.receiveMessageGet(MAX_DATA_BYTES)
```

```
            return rxd
        else:
            return
```

Listing 1: canfdlib

A.2 Constants

Not all constants present in this file are used. Some are present with future update in mind.

```
cs = 3
max_speed_hz = 20000000
mode = 0
MSBFIRST = 0
LSBFIRST = 1

bitorder = MSBFIRST

RAND_MAX = 0x7fff
SPI_DEFAULT_BUFFER_LENGTH = 96
SPI_MAX_BUFFER_LENGTH = 30000
SPI_BAUDRATE = 8000000

ISO_CRC = 1

# SPI Instruction Set
cINSTRUCTION_RESET = 0b0000
cINSTRUCTION_READ = 0x03
cINSTRUCTION_READ_CRC = 0x0B
cINSTRUCTION_WRITE = 0x02
cINSTRUCTION_WRITE_CRC = 0x0A
cINSTRUCTION_WRITE_SAFE = 0x0C

# crc16 lookup table
crc16_table = (
    0x0000, 0x8005, 0x800F, 0x000A, 0x801B, 0x001E, 0x0014, 0x8011,
    0x8033, 0x0036, 0x003C, 0x8039, 0x0028, 0x802D, 0x8027, 0x0022,
    0x8063, 0x0066, 0x006C, 0x8069, 0x0078, 0x807D, 0x8077, 0x0072,
    0x0050, 0x8055, 0x805F, 0x005A, 0x804B, 0x004E, 0x0044, 0x8041,
    0x80C3, 0x00C6, 0x00CC, 0x80C9, 0x00D8, 0x80DD, 0x80D7, 0x00D2,
    0x00F0, 0x80F5, 0x80FF, 0x00FA, 0x80EB, 0x00EE, 0x00E4, 0x80E1,
    0x00A0, 0x80A5, 0x80AF, 0x00AA, 0x80BB, 0x00BE, 0x00B4, 0x80B1,
    0x8093, 0x0096, 0x009C, 0x8099, 0x0088, 0x808D, 0x8087, 0x0082,
    0x8183, 0x0186, 0x018C, 0x8189, 0x0198, 0x819D, 0x8197, 0x0192,
    0x01B0, 0x81B5, 0x81BF, 0x01BA, 0x81AB, 0x01AE, 0x01A4, 0x81A1,
    0x01E0, 0x81E5, 0x81EF, 0x01EA, 0x81FB, 0x01FE, 0x01F4, 0x81F1,
    0x81D3, 0x01D6, 0x01DC, 0x81D9, 0x01C8, 0x81CD, 0x81C7, 0x01C2,
    0x0140, 0x8145, 0x814F, 0x014A, 0x815B, 0x015E, 0x0154, 0x8151,
    0x8173, 0x0176, 0x017C, 0x8179, 0x0168, 0x816D, 0x8167, 0x0162,
    0x8123, 0x0126, 0x012C, 0x8129, 0x0138, 0x813D, 0x8137, 0x0132,
    0x0110, 0x8115, 0x811F, 0x011A, 0x810B, 0x010E, 0x0104, 0x8101,
    0x8303, 0x0306, 0x030C, 0x8309, 0x0318, 0x831D, 0x8317, 0x0312,
    0x0330, 0x8335, 0x833F, 0x033A, 0x832B, 0x032E, 0x0324, 0x8321,
    0x0360, 0x8365, 0x836F, 0x036A, 0x837B, 0x037E, 0x0374, 0x8371,
    0x8353, 0x0356, 0x035C, 0x8359, 0x0348, 0x834D, 0x8347, 0x0342,
    0x03C0, 0x83C5, 0x83CF, 0x03CA, 0x83DB, 0x03DE, 0x03D4, 0x83D1,
    0x83F3, 0x03F6, 0x03FC, 0x83F9, 0x03E8, 0x83ED, 0x83E7, 0x03E2,
    0x83A3, 0x03A6, 0x03AC, 0x83A9, 0x03B8, 0x83BD, 0x83B7, 0x03B2,
    0x0390, 0x8395, 0x839F, 0x039A, 0x838B, 0x038E, 0x0384, 0x8381,
    0x0280, 0x8285, 0x828F, 0x028A, 0x829B, 0x029E, 0x0294, 0x8291,
```

```
    0x82B3, 0x02B6, 0x02BC, 0x82B9, 0x02A8, 0x82AD, 0x82A7, 0x02A2,
    0x82E3, 0x02E6, 0x02EC, 0x82E9, 0x02F8, 0x82FD, 0x82F7, 0x02F2,
    0x02D0, 0x82D5, 0x82DF, 0x02DA, 0x82CB, 0x02CE, 0x02C4, 0x82C1,
    0x8243, 0x0246, 0x024C, 0x8249, 0x0258, 0x825D, 0x8257, 0x0252,
    0x0270, 0x8275, 0x827F, 0x027A, 0x826B, 0x026E, 0x0264, 0x8261,
    0x0220, 0x8225, 0x822F, 0x022A, 0x823B, 0x023E, 0x0234, 0x8231,
    0x8213, 0x0216, 0x021C, 0x8219, 0x0208, 0x820D, 0x8207, 0x0202
)

# Register Addresses
# can_fd_ubp
cREGADDR_CiCON = 0x000
cREGADDR_CiNBTCFG = 0x004
cREGADDR_CiDBTCFG = 0x008
cREGADDR_CiTDC = 0x00C

cREGADDR_CiTBC = 0x010
cREGADDR_CiTSCON = 0x014
cREGADDR_CiVEC = 0x018
cREGADDR_CiINT = 0x01C
cREGADDR_CiINTFLAG = cREGADDR_CiINT
cREGADDR_CiINTENABLE = (cREGADDR_CiINT+2)

cREGADDR_CiRXIF = 0x020
cREGADDR_CiTXIF = 0x024
cREGADDR_CiRXOVIF = 0x028
cREGADDR_CiTXATIF = 0x02C

cREGADDR_CiTXREQ = 0x030
cREGADDR_CiTREC = 0x034
cREGADDR_CiBDIAG0 = 0x038
cREGADDR_CiBDIAG1 = 0x03C

cREGADDR_CiTEFCON = 0x040
cREGADDR_CiTEFSTA = 0x044
cREGADDR_CiTEFUA = 0x048
cREGADDR_CiFIFOBA = 0x04C

cREGADDR_CiFIFOCON = 0x050
cREGADDR_CiFIFOSTA = 0x054
cREGADDR_CiFIFOUA = 0x058
CiFIFO_OFFSET = (3*4)

CiFILTER_OFFSET = (2*4)

# MCP2517 Specific
cREGADDR_OSC = 0xE00
cREGADDR_IOCON = 0xE04
cREGADDR_CRC = 0xE08
cREGADDR_ECCCON = 0xE0C
cREGADDR_ECCSTA = 0xE10

# RAM addresses
cRAM_SIZE = 2048
cRAMADDR_START = 0x400
cRAMADDR_END = (cRAMADDR_START+cRAM_SIZE)

# CAN Operation Modes
NORMAL_MODE = 0x00
SLEEP_MODE = 0x01
INTERNAL_LOOPBACK_MODE = 0x02
```

```
LISTEN_ONLY_MODE = 0x03
CONFIGURATION_MODE = 0x04
EXTERNAL_LOOPBACK_MODE = 0x05
CLASSIC_MODE = 0x06
RESTRICTED_MODE = 0x07
INVALID_MODE = 0xFF

# Message IDs
FILE_START_ID = 0xd0
FILE_STOP_ID = 0xdf
FILE_DATA_ID = 0xda
TX_REQUEST_ID = 0x300
TX_RESPONSE_ID = 0x301
BUTTON_STATUS_ID = 0x201
LED_STATUS_ID = 0x200
BITTIME_SET_ID = 0x100
PAYLOAD_ID = 0x101
BITTIME_CFG_GET_ID = 0x600
BITTIME_CFG_125K_ID = 0x601
BITTIME_CFG_250K_ID = 0x602
BITTIME_CFG_500K_ID = 0x603
BITTIME_CFG_1M_ID = 0x604

#CAN FIFO channels
CAN_FIFO_CH0 = 0
CAN_FIFO_CH1 = 1
CAN_FIFO_CH2 = 2
CAN_FIFO_CH3 = 3
CAN_FIFO_CH4 = 4
CAN_FIFO_CH5 = 5
CAN_FIFO_CH6 = 6
CAN_FIFO_CH7 = 7
CAN_FIFO_CH8 = 8
CAN_FIFO_CH9 = 9
CAN_FIFO_CH10 = 10
CAN_FIFO_CH11 = 11
CAN_FIFO_CH12 = 12
CAN_FIFO_CH13 = 13
CAN_FIFO_CH14 = 14
CAN_FIFO_CH15 = 15
CAN_FIFO_CH16 = 16
CAN_FIFO_CH17 = 17
CAN_FIFO_CH18 = 18
CAN_FIFO_CH19 = 19
CAN_FIFO_CH20 = 20
CAN_FIFO_CH21 = 21
CAN_FIFO_CH22 = 22
CAN_FIFO_CH23 = 23
CAN_FIFO_CH24 = 24
CAN_FIFO_CH25 = 25
CAN_FIFO_CH26 = 26
CAN_FIFO_CH27 = 27
CAN_FIFO_CH28 = 28
CAN_FIFO_CH29 = 29
CAN_FIFO_CH30 = 30
CAN_FIFO_CH31 = 31
CAN_FIFO_TOTAL_CHANNELS = 32

cREGADDR_CiFLTCON = (cREGADDR_CiFIFOCON+(CiFIFO_OFFSET*
    CAN_FIFO_TOTAL_CHANNELS))
cREGADDR_CiFLTOBJ = (cREGADDR_CiFLTCON+CAN_FIFO_TOTAL_CHANNELS)
```

```
cREGADDR_CiMASK = (cREGADDR_CiFLTOBJ+4)

# Application states
APP_STATE_INIT = 0
APP_STATE_REQUEST_CONFIG = 1
APP_STATE_WAIT_FOR_CONFIG = 2
APP_STATE_INIT_TXOBJ = 3
APP_STATE_TRANSMIT = 4
APP_STATE_RECEIVE = 5
APP_STATE_PAYLOAD = 6
APP_STATE_TEST_RAM_ACCESS = 7
APP_STATE_TEST_REGISTER_ACCESS = 8

CAN_PLSIZE_64 = 7

CAN_500K_1M = 0
CAN_500K_2M = 1
CAN_500K_3M = 2
CAN_500K_4M = 3
CAN_500K_5M = 4
CAN_500K_6M7 = 5
CAN_500K_8M = 6
CAN_500K_10M = 7
CAN_250K_500K = 8
CAN_250K_833K = 9
CAN_250K_1M = 10
CAN_250K_1M5 = 11
CAN_250K_2M = 12
CAN_250K_3M = 13
CAN_250K_4M = 14
CAN_1000K_4M = 15
CAN_1000K_8M = 16
CAN_125K_500K = 17

CAN_SYSCLK_40M = 0
CAN_SYSCLK_20M = 1

CAN_SSP_MODE_OFF = 0
CAN_SSP_MODE_MANUAL = 1
CAN_SSP_MODE_AUTO = 2

MAX_MSG_SIZE = 76
MAX_DATA_BYTES = 64

MAX_TXQUEUE_ATTEMPTS = 50

CAN_RX_FIFO_NO_EVENT = 0
CAN_RX_FIFO_ALL_EVENTS = 0x0F
CAN_RX_FIFO_NOT_EMPTY_EVENT = 0x01
CAN_RX_FIFO_HALF_FULL_EVENT = 0x02
CAN_RX_FIFO_FULL_EVENT = 0x04
CAN_RX_FIFO_OVERFLOW_EVENT = 0x08

CAN_TX_FIFO_NO_EVENT = 0
CAN_TX_FIFO_ALL_EVENTS = 0x17
CAN_TX_FIFO_NOT_FULL_EVENT = 0x01
CAN_TX_FIFO_HALF_FULL_EVENT = 0x02
CAN_TX_FIFO_EMPTY_EVENT = 0x04
CAN_TX_FIFO_ATTEMPTS_EXHAUSTED_EVENT = 0x10

CAN_ERROR_FREE_STATE = 0,
```

```
CAN_ERROR_ALL = 0x3F,
CAN_TX_RX_WARNING_STATE = 0x01,
CAN_RX_WARNING_STATE = 0x02,
CAN_TX_WARNING_STATE = 0x04,
CAN_RX_BUS_PASSIVE_STATE = 0x08,
CAN_TX_BUS_PASSIVE_STATE = 0x10,
CAN_TX_BUS_OFF_STATE = 0x20

CAN_DLC_0 = 0
CAN_DLC_1 = 1
CAN_DLC_2 = 2
CAN_DLC_3 = 3
CAN_DLC_4 = 4
CAN_DLC_5 = 5
CAN_DLC_6 = 6
CAN_DLC_7 = 7
CAN_DLC_8 = 8
CAN_DLC_12 = 9
CAN_DLC_16 = 10
CAN_DLC_20 = 11
CAN_DLC_24 = 12
CAN_DLC_32 = 13
CAN_DLC_48 = 14
CAN_DLC_64 = 15

CAN_DLC_dict = {
    CAN_DLC_12 : 12,
    CAN_DLC_16 : 16,
    CAN_DLC_20 : 20,
    CAN_DLC_24 : 24,
    CAN_DLC_32 : 32,
    CAN_DLC_48 : 48,
    CAN_DLC_64 : 64
}

CAN_DLC_inv_dict = {
    4 : CAN_DLC_4,
    8 : CAN_DLC_8,
    12 : CAN_DLC_12,
    16 : CAN_DLC_16,
    20 : CAN_DLC_20,
    24 : CAN_DLC_24,
    32 : CAN_DLC_32,
    48 : CAN_DLC_48,
    64 : CAN_DLC_64
}

# Bit time configurations const
BitTimeConfig125K = [1, 0x11, 500]
BitTimeConfig250K = [7, 0x08, 500, 833, 1000, 1500, 2000, 3076,
    4000]
BitTimeConfig500K = [8, 0x00, 1000, 2000, 3076, 4000, 5000, 6666,
    8000, 10000]
BitTimeConfig1M = [2, 0x0f, 4000, 8000]

all500k = (CAN_500K_1M, CAN_500K_2M, CAN_500K_3M, CAN_500K_4M,
    CAN_500K_5M, CAN_500K_6M7, CAN_500K_8M, CAN_500K_10M)
all250k = (CAN_250K_500K, CAN_250K_833K,  CAN_250K_1M, CAN_250K_1M5,
    CAN_250K_2M, CAN_250K_3M, CAN_250K_4M)
all1000k = (CAN_1000K_4M, CAN_1000K_8M)
```

Listing 2: Constants

A.3 Registers

```
import ctypes
c_uint32 = ctypes.c_uint32
c_bool = ctypes.c_bool
endianType = ctypes.LittleEndianStructure
union = ctypes.Union

class REG_CAN_CONFIG(endianType):
    _fields_ = [
        ("TxBandWidthSharing", c_uint32, 4),
        ("TXQEnable", c_uint32, 1),
        ("StoreInTEF", c_uint32, 1),
        ("SystemErrorToListenOnly", c_uint32, 1),
        ("EsiInGatewayMode", c_uint32, 1),
        ("RestrictReTxAttempts", c_uint32, 1),
        ("BitRateSwitchDisable", c_uint32, 1),
        ("WakeUpFilterTime", c_uint32, 2),
        ("WakeUpFilterEnable", c_uint32, 1),
        ("ProtocolExceptionEventDisable", c_uint32, 1),
        ("IsoCrcEnable", c_uint32, 1),
        ("DNetFilterCount", c_uint32, 5),
              ]

class CAN_TX_FIFO_CONFIG(endianType):
    _fields_ = [
        ("RTREnable", c_uint32, 1),
        ("TxPriority", c_uint32, 5),
        ("TxAttempts", c_uint32, 2),
        ("FifoSize", c_uint32, 5),
        ("PayLoadSize", c_uint32, 3)]

class CAN_RX_FIFO_CONFIG(endianType):
    _fields_ = [
        ("RxTimeStampEnable",  c_uint32,1),
        ("FifoSize", c_uint32, 5),
        ("PayLoadSize", c_uint32, 3)]

class CAN_MSGOBJ_ID(endianType):
    _fields_ = [
        ("unimplemented1", c_uint32, 2),
        ("SID11", c_uint32, 1),
        ("EID", c_uint32, 18),
        ("SID", c_uint32, 11),
        ]


class CAN_TX_MSGOBJ_CTRL(endianType):
    _fields_ = [
        ("unimplemented1", c_uint32, 16),
        ("SEQ", c_uint32, 7),
        ("ESI", c_uint32, 1),
        ("FDF", c_uint32, 1),
        ("BRS", c_uint32, 1),
        ("RTR", c_uint32, 1),
        ("IDE", c_uint32, 1),
        ("DLC", c_uint32, 4),
        ]

class CAN_TX_MSGOBJ(endianType):
    _fields_ = [
```

```
        ("id", CAN_MSGOBJ_ID),
        ("ctrl", CAN_TX_MSGOBJ_CTRL),
        ("CAN_MSG_TIMESTAMP", c_uint32)
    ]

class CAN_RX_MSGOBJ_CTRL(endianType):
    _fields_ = [
        ("unimplemented2", c_uint32, 16),
        ("FilterHit", c_uint32, 5),
        ("unimplemented1", c_uint32, 2),
        ("ESI", c_uint32, 1),
        ("FDF", c_uint32, 1),
        ("BRS", c_uint32, 1),
        ("RTR", c_uint32, 1),
        ("IDE", c_uint32, 1),
        ("DLC", c_uint32, 4),
        ]
class CAN_RX_MSGOBJ(endianType):
    _fields_ = [
        ("id", CAN_MSGOBJ_ID),
        ("ctrl", CAN_RX_MSGOBJ_CTRL),
        ("CAN_MSG_TIMESTAMP", c_uint32)
    ]
```

Listing 3: Registers

A.4 Tests

```
import binascii
import ft
from constants import *

ft232h = None
spi = None

def init():
    ft.use_FT232H()
    global ft232h
    global spi
    ft232h = ft.FT232H()
    spi = ft.SPI(ft232h, cs=3, max_speed_hz=20000000, mode=0,
    bitorder=ft.MSBFIRST)

def spi_reset():
    spiTransmitBuffer = []
    spiTransmitBuffer.append(cINSTRUCTION_RESET << 4)
    spiTransmitBuffer.append(0)
    spi._assert_cs()
    spi.write(spiTransmitBuffer)
    spi._deassert_cs()

def test1():
    init()
    spi_reset()
    spiTransmitBuffer = []
    address = 0xE00
    spiTransmitBuffer.append((cINSTRUCTION_READ << 4) + ((address >>
     8) & 0xF))
    spiTransmitBuffer.append(address & 0xFF)
    spi._assert_cs()
```

```
    spi.write(spiTransmitBuffer)
    response = spi.read(4)
    spi._deassert_cs()

    result = 'Reading OSC register with SPI. Result:\n32-bit word: {
    word}\nBytes: {bytes}'.format(bytes=list(response), word=
    binascii.hexlify(response))
    return result

def test2():
    init()
    spi_reset()
    spiTransmitBuffer = []
    address = 0x000
    spiTransmitBuffer.append((cINSTRUCTION_READ << 4) + ((address >>
     8) & 0xF))
    spiTransmitBuffer.append(address & 0xFF)
    spi._assert_cs()
    spi.write(spiTransmitBuffer)
    response_0 = spi.read(4)
    spi._deassert_cs()

    result = 'CiCON register before writing:\n32-bit word: {word}\
    nBytes: {bytes}\n'.format(bytes=list(response_0),word=binascii.
    hexlify(response_0))

    spiTransmitBuffer = []
    data = [0, 0, 0, 0]
    result = result + "Data to write (4 bytes): {}\n".format(data)

    addressW = 0x000
    spiTransmitBuffer.append((cINSTRUCTION_WRITE << 4) + ((addressW
    >> 8) & 0xF))
    spiTransmitBuffer.append(addressW & 0xFF)
    spiTransmitBuffer = spiTransmitBuffer + data

    spi._assert_cs()
    spi.write(spiTransmitBuffer)
    spi._deassert_cs()

    spiTransmitBuffer = []
    spiTransmitBuffer.append((cINSTRUCTION_READ << 4) + ((address >>
     8) & 0xF))
    spiTransmitBuffer.append(address & 0xFF)

    spi._assert_cs()
    spi.write(spiTransmitBuffer)
    response_1 = spi.read(4)
    spi._deassert_cs()

    result = result + 'CiCON register modified:\n32-bit word: {word
    }\nBytes: {bytes}\n'.format(bytes=list(response_1),word=binascii
    .hexlify(response_1))
    return result

def test3(canfd):
    canfd.reset()
    address = 0x000
    word = canfd.readWord(address)
    result = 'Reading CiCON: {}\n:'.format(word)
```

```
    write_word = 0x600798F4
    result = result + "Word to write: {}\n".format(write_word)
    canfd.writeWord(address, write_word)
    word = canfd.readWord(address)
    result = result + 'Reading CiCON with {write_word} written on it
    : {word}\n'.format(write_word=write_word, word=word)

    canfd.reset()
    result = result + "Resetting...\n"

    write_byte = 0x6F
    canfd.writeByte(address, write_byte)
    word = canfd.readWord(address)
    result = result + 'Reading CiCON with {write_byte} written on
    its 1st byte: {word}'.format(write_byte=write_byte, word=word)

    canfd.reset()
    result = result + "Resetting...\n"

    write_byte_array = [0x60, 0x07, 0x98, 0xF4]
    canfd.writeByteArray(address, write_byte_array)
    word = canfd.readWord(address)
    result = result + 'Reading CiCON with {write_byte_array} array
    written on it (4 bytes): {word}\n'.format(write_byte_array=
    write_byte_array, word=word)

    canfd.reset()
    result = result + "Resetting...\n"

    write_word_array = [0x600798F4, 0x7f0f3eff]
    canfd.writeWordArray(address, write_word_array)
    word = canfd.readWordArray(address, 2)
    result = result + 'Reading CiCON and CiNBTCFG with {
    write_word_array} written on it: {word}\n'.format(
    write_word_array=write_word_array, word=word)

    return result

def test4(canfd):
    canfd.reset()
    result = canfd.ramTest()
    if result == -1:
        result = "RAM test failed!\n"
    else:
        result = 'RAM test succesful!\n'
    return result

def test5(canfd):
    canfd.reset()
    result = canfd.registerTest()
    if result == -1:
        result = "Register test failed!\n"
    else:
        result = 'Register test succesful!\n'
    return result

def test6(canfd):
    canfd.reset()
    mode = canfd.operationModeGet()
    result = "Mode after reset: {}\n".format(mode)
    result = result + 'selecting NORMAL_MODE\n'
```

```
    canfd.operationModeSelect(NORMAL_MODE)
    mode = canfd.operationModeGet()
    result = result + "Device mode: {}\n".format(mode) + 'Selecting
    INTERNAL_LOOPBACK_MODE\n'

    canfd.operationModeSelect(INTERNAL_LOOPBACK_MODE)
    mode = canfd.operationModeGet()
    result = result + "Device mode: {}\n".format(mode) + 'Selecting
    CONFIGURATION_MODE\n'

    canfd.operationModeSelect(CONFIGURATION_MODE)
    mode = canfd.operationModeGet()
    result = result +  "Device mode: {}\n".format(mode) + '
    Initialazing...\n'

    canfd.initialize()
    mode = canfd.operationModeGet()
    result = result + "Device mode after init: {}\n".format(mode)
    return result

def test7(canfd):
        canfd.initialize()
    canfd.operationModeSelect(EXTERNAL_LOOPBACK_MODE)
    canfd.txdlc = 8
    txd = [0, 0xA, 0xF1, 1, 13, 7, 253, 27]
    result = 'Message to transmit: {}\n'.format(txd)
    canfd.transmitMessageTasks(txd)
    # receive message
    rxd = canfd.receiveMessageTasks()
    result = result +"Received message: {}\n".format(rxd)
    return result


test_dict = {
    'test1': test1,
    'test2': test2,
    'test3': test3,
    'test4': test4,
    'test5': test5,
    'test6': test6,
    'test7': test7
}
```

Listing 4: Tests

A.5 GUI

```
import canfdlib
from mttkinter import mtTkinter as tk
import tkMessageBox
from Adafruit_GPIO.FT232H import FT232H
from gui_constants import *
from constants import *
import tests

class canfdgui():
    def __init__(self, inputDict=None, debug=False):
        self.inputDict = inputDict
        self.debug = debug
```

```
        self.rxd = []
        self.txd = []

        self.window = tk.Tk()
        self.window.title(window_title)
        self.window.geometry(window_size)

        self.right_frame = tk.Frame(self.window, width=450, height
=100)
        self.left_frame = tk.Frame(self.window, width=250, height
=100)
        self.corner_frame = tk.Frame(self.window, width=100, height
=20)
        self.extra_frame = tk.Frame(self.window, width=30, height
=100)
        self.window.grid_columnconfigure(1, weight=1)
        self.right_frame.grid(row=0, column=1, sticky="nsew")
        self.left_frame.grid(row=0, column=0, sticky="nsew")
        self.corner_frame.grid(row=1,column=0,sticky="sw")
        self.extra_frame.grid(row=0, column=2)

        # textbot for rx
        self.rx_box_scrollbar = tk.Scrollbar(self.left_frame)
        self.rx_box_text = tk.Text(self.left_frame, height=
rx_textbox_height, width=rx_textbox_width)
        self.rx_box_scrollbar.grid(column=1, row=0,  sticky=tk.N+tk.
S+tk.W)
        self.rx_box_text.grid(column=0, row=0)
        self.rx_box_scrollbar.config(command=self.rx_box_text.yview)
        self.rx_box_text.config(yscrollcommand=self.rx_box_scrollbar
.set)

        # rx button
        self.receive_start_button = tk.Button(self.right_frame, text
="Start RX", command = self.receive)
        self.receive_start_button.grid(column=0,row=0, pady=5)

        # clear rx box windows button
        self.clear_button = tk.Button(self.right_frame, text="Clear"
, command=self.clear)
        self.clear_button.grid(column=0, row=1, pady=5)

        # tx button
        self.transmit_start_button = tk.Button(self.corner_frame,
text="Start TX", command=self.transmit)
        self.transmit_start_button.grid(column=2,row=0)

        # tx message
        self.tx_msg = tk.Entry(self.corner_frame, width=tx_msg_width
)
        self.tx_msg.grid(column=1,row=0)

        # tx label
        self.txlbl = tk.Label(self.corner_frame, text="TX Message:")
        self.txlbl.grid(column=0, row=0)

        # rx channel button
        self.rx_channel_button = tk.Button(self.right_frame, text="
Set RX channel", command=self.setRXchannel)
        self.rx_channel_button.grid(column=1, row=3, pady=5)
```

```
        # rx channel
        self.rx_channel = tk.Entry(self.right_frame, width=10)
        self.rx_channel.grid(column=1, row=4, pady=5)

        # tx channel button
        self.tx_channel_button = tk.Button(self.right_frame, text="
Set TX channel", command=self.setTXchannel)
        self.tx_channel_button.grid(column=0, row=3, pady=5)

        # tx channel
        self.tx_channel = tk.Entry(self.right_frame, width=10)
        self.tx_channel.grid(column=0, row=4, pady=5)

        # reset button
        self.reset_button = tk.Button(self.right_frame, text="Reset
Device", command=self.reset)
        self.reset_button.grid(column=0,row=2, pady=5)

        # opMode droplist and button
        OPTIONS = ["NORMAL_MODE","SLEEP_MODE","
INTERNAL_LOOPBACK_MODE","LISTEN_ONLY_MODE","CONFIGURATION_MODE",
"EXTERNAL_LOOPBACK_MODE","CLASSIC_MODE","RESTRICTED_MODE","
INVALID_MODE"]

        self.droplist = tk.StringVar(self.left_frame)
        self.droplist.set(OPTIONS[0])  # default value

        w = tk.OptionMenu(self.right_frame, self.droplist, *OPTIONS)
        w.grid(column=1,row=1, padx=5)

        self.opmode_button = tk.Button(self.right_frame, text="Set
config mode", command=self.changemode)
        self.opmode_button.grid(column=1,row=2, pady=5)

        # stop button
        self.stop_button = tk.Button(self.right_frame, text="STOP",
command=self.stop)
        self.stop_button.grid(column=1, row=0, pady=5)

        # connect button
        self.connect_button = tk.Button(self.extra_frame, text="
CONNECT", command=self.connect)
        self.connect_button.grid(column=0, row=2, pady=5) # enlarge
        self.connect_button.config(height=3, width=15)

        # dlc droplist and button
        OPTIONS_dlc = ["CAN_DLC_0","CAN_DLC_1","CAN_DLC_2","
CAN_DLC_3","CAN_DLC_4","CAN_DLC_5","CAN_DLC_6","CAN_DLC_7","
CAN_DLC_8","CAN_DLC_12","CAN_DLC_16","CAN_DLC_20","CAN_DLC_24","
CAN_DLC_32","CAN_DLC_48","CAN_DLC_64"]

        self.droplist_dlc = tk.StringVar(self.extra_frame)
        self.droplist_dlc.set(OPTIONS_dlc[-1])  # default value

        w_dlc = tk.OptionMenu(self.extra_frame, self.droplist_dlc, *
OPTIONS_dlc)
        w_dlc.grid(column=0, row=0, padx=5)

        self.dlc_button = tk.Button(self.extra_frame, text="Set DLC"
, command=self.changedlc)
```

```
        self.dlc_button.grid(column=0, row=1, pady=5)

        self.canfd = None

        if self.debug:
           # self.window.after(1000, self.dummy_main)
            pass
        else:
            try:
                self.connect()
            except RuntimeError:
                self.rx_box_text.insert(tk.END, "Device not ready.
Connect manually." + '\n')
            finally:
                #self.window.after(10, self.main)
                self.main()
                self.window.mainloop()

    def connect(self):
        if self.debug:
            print "Debug mode"
            self.rx_box_text.insert(tk.END, "Debug mode" + '\n')
        else:
            #print(self.inputDict)
            try:
                if self.inputDict["ft232h"] is None:
                    ft232h = FT232H()
                else:
                    ft232h = self.inputDict["ft232h"]
                cs = self.inputDict["cs"]
                max_speed_hz = self.inputDict["max_speed_hz"]
                mode = self.inputDict["mode"]
                bitorder = self.inputDict["bitorder"]
                SPI_DEFAULT_BUFFER_LENGTH = self.inputDict["
SPI_DEFAULT_BUFFER_LENGTH"]
                SPI_MAX_BUFFER_LENGTH = self.inputDict["
SPI_MAX_BUFFER_LENGTH"]
                SPI_BAUDRATE = self.inputDict["SPI_BAUDRATE"]
                self.canfd = canfdlib.CANFD_SPI(ft232h, cs,
max_speed_hz, mode, bitorder, SPI_DEFAULT_BUFFER_LENGTH,
SPI_MAX_BUFFER_LENGTH, SPI_BAUDRATE)
                self.canfd.initialize()
                self.rx_box_text.insert(tk.END, "Device connected
succesfully!" + '\n')
            except:
                self.rx_box_text.insert(tk.END, "Device not found."
+ '\n')

    def init(self):
        self.canfd.state = APP_STATE_INIT
        # call initialize() from button
        self.rx_box_text.insert(tk.END, 'Initializing device...' + '
\n')

    def receive(self):
        # start receiving
        self.canfd.state = APP_STATE_RECEIVE


    def transmit(self):
        input = self.tx_msg.get()
```

```
        if 'test' in input:
            self.launchTest(input)
        else:
            self.txd = [int(value) for value in input.split(',')]
            self.canfd.state = APP_STATE_TRANSMIT

    def setRXchannel(self):
        # set rx channel and reset
        try:
        #if self.rx_channel.get() in range(0, 32):
            channel = int(self.rx_channel.get())
            if channel in range(0, 32):
                self.canfd.state = "idle"
                self.canfd.rxchannel = channel
                self.rx_box_text.insert(tk.END, "RX channel set to:
{}".format(self.canfd.rxchannel) + '\n')
                self.canfd.reset()
       #else:
        except ValueError:
            tkMessageBox.showinfo("Warning", "Not a valid channel.\
nTry a number in range(0, 32)")

    def setTXchannel(self):
        # set tx channel and reset
       # if self.tx_channel.get() in range(0,32):
        try:
            channel = int(self.tx_channel.get())
            if channel in range(0, 32):
                self.canfd.state = "idle"
                self.canfd.txchannel = channel
                self.rx_box_text.insert(tk.END, "TX channel set to:
{}".format(self.canfd.txchannel) + '\n')
                self.canfd.reset()
        #else:
        except ValueError:
            tkMessageBox.showinfo("Warning", "Not a valid channel.\
nTry a number in range(0,32)")

    def pause(self):
        # put state to idle  and do nothing
        self.canfd.state = "idle"

    def stop(self):
        # put state to stop, and abort execution
        #self.canfd.operationModeSelect(SLEEP_MODE)
        self.rx_box_text.insert(tk.END, "Stopping..." + '\n')
        self.canfd.state = "stop"

    def clear(self):
        # clear window
        self.rx_box_text.delete('1.0', tk.END)

    def reset(self):
        self.canfd.state = "idle"
        self.canfd.reset()
        self.rx_box_text.insert(tk.END, "Resetting..." + '\n')

    def changemode(self):
        self.canfd.state = "idle"
        mode = mode_dict[self.droplist.get()]
        print(self.canfd.operationModeGet())
```

```
        self.rx_box_text.insert(tk.END, "Previous opMode: {mode} - {
    value}".format(value=self.canfd.operationModeGet(), mode=
    mode_dict_inv[self.canfd.operationModeGet()]) + '\n')
        self.canfd.operationModeSelect(mode)
        self.rx_box_text.insert(tk.END, "opMode changed to : {mode}
    - {value}".format(value=self.canfd.operationModeGet(), mode=
    mode_dict_inv[self.canfd.operationModeGet()]) + '\n')
        #self.canfd.reset()

    def changedlc(self):
        self.canfd.state = "idle"
        self.rx_box_text.insert(tk.END, "Previous mode: {mode} - {
    value}".format(value=self.canfd.txdlc, mode=dlc_inv_dict[self.
    canfd.txdlc]) + '\n')
        self.canfd.txdlc = dlc_dict[self.droplist_dlc.get()]
        self.rx_box_text.insert(tk.END, "Mode changed to: {mode} - {
    value}".format(value=self.canfd.txdlc, mode=dlc_inv_dict[self.
    canfd.txdlc]) + '\n')

    def launchTest(self, test):
        test_num = int(test[-1])
        #print(test_num)
        try:
            if test_num < 3:
                result = tests.test_dict[test]()
            else:
                result = tests.test_dict[test](self.canfd)
            self.rx_box_text.insert(tk.END, result + '\n')
            self.canfd.reset()
        except KeyError:
            self.rx_box_text.insert(tk.END, 'Test not found' + '\n')

    def main(self):
        # put main function here
        if self.canfd != None:
            if self.canfd.state == APP_STATE_INIT:
                # Initialize
                self.canfd.initialize()
                self.canfd.state = "idle"

            elif self.canfd.state == APP_STATE_TEST_RAM_ACCESS:
                ram_test_result = self.canfd.ramTest()
                if ram_test_result == -1:
                    self.rx_box_text.insert(tk.END, 'RAM test failed
    !' + '\n')
                else:
                    self.rx_box_text.insert(tk.END, 'RAM test
    successful!' + '\n')
                self.canfd.state = "idle"

            elif self.canfd.state == APP_STATE_TEST_REGISTER_ACCESS:
                register_test_result = self.canfd.registerTest()
                if register_test_result == -1:
                    self.rx_box_text.insert(tk.END, 'Register test
    failed!' + '\n')
                else:
                    self.rx_box_text.insert(tk.END, 'Register test
    successful!' + '\n')
                self.canfd.state = "idle"

            elif self.canfd.state == APP_STATE_INIT_TXOBJ:
```

```
                self.canfd.initTxObj()

            elif self.canfd.state == APP_STATE_RECEIVE:
                result = self.canfd.receiveMessageTasks()
                print(result)
                #self.rxd = result
                # print results
                if result is not None:
                    if self.rxd != result:
                        self.rxd = result
                        self.rx_box_text.insert(tk.END, '{}'.format(
    self.rxd) + '\n')
                #self.canfd.state = "idle"

            elif self.canfd.state == APP_STATE_TRANSMIT:
                self.canfd.transmitMessageTasks(self.txd)
                self.canfd.state = "idle"
                self.canfd.txd = None

            elif self.canfd.state == APP_STATE_REQUEST_CONFIG:
                self.canfd.operationModeSelect(CONFIGURATION_MODE)
                self.canfd.state = APP_STATE_WAIT_FOR_CONFIG

            elif self.canfd.state == APP_STATE_WAIT_FOR_CONFIG:
                opMode = self.canfd.operationModeGet()
                if opMode != CONFIGURATION_MODE:
                    self.canfd.state = APP_STATE_WAIT_FOR_CONFIG
                else:
                    self.canfd.state = APP_STATE_INIT

            elif self.canfd.state == "idle":
                pass

            elif self.canfd.state == "stop":
                pass
        self.window.after(10,self.main)

    def dummy_main(self):
        print("Debug mode")
        self.window.after(1000, self.dummy_main)

try:
    ft232h = FT232H()
except RuntimeError:
    print("Device not found.")
    ft232h = None

inputDict = {"ft232h":ft232h,
             "cs":cs,
             "max_speed_hz":max_speed_hz,
             "mode":mode,
             "bitorder":bitorder,
             "SPI_DEFAULT_BUFFER_LENGTH":SPI_DEFAULT_BUFFER_LENGTH,
             "SPI_MAX_BUFFER_LENGTH":SPI_MAX_BUFFER_LENGTH,
             "SPI_BAUDRATE":SPI_BAUDRATE}

a = canfdgui(inputDict)
```

Listing 5: GUI

A.6 RAM test

```
def ramTest(self):
    # verify R/W
    for length in range(4, MAX_DATA_BYTES + 1, 4):
        txd = [(randint(0, RAND_MAX) & 0xFF) for e in range(0,
    length)]
        #print("Data written on RAM: {}".format(txd))
        self.writeByteArray(cRAMADDR_START, txd)
        rxd = self.readByteArray(cRAMADDR_START, length)
        #print("Data read on RAM: {}".format(rxd))
        for i in range(0, length):
            good = txd[i] == rxd[i]
            if not good:
                print("Data mismatch!")
                return -1
    return 1
result = canfd.ramTest()
if result == -1:
    print("RAM test failed!")
else:
    print ('RAM test succesful!')

>> Data written on RAM: [38, 120, 168, 171]
>> Data read on RAM: [38, 120, 168, 171]
>> Data written on RAM: [140, 52, 197, 60, 5, 14, 137, 142]
>> Data read on RAM: [140, 52, 197, 60, 5, 14, 137, 142]
>> Data written on RAM: [220, 71, 136, 33, 168, 87, 96, 78, 161, 89,
     84, 231]
>> Data read on RAM: [220, 71, 136, 33, 168, 87, 96, 78, 161, 89,
    84, 231]
>> Data written on RAM: [174, 215, 219, 197, 227, 165, 122, 129,
    255, 21, 15, 95, 153, 177, 113, 188]
>> Data read on RAM: [174, 215, 219, 197, 227, 165, 122, 129, 255,
    21, 15, 95, 153, 177, 113, 188]
>> Data written on RAM: [93, 29, 106, 181, 146, 167, 14, 155, 34,
    186, 91, 70, 217, 253, 101, 147, 248, 178, 56, 61]
>> Data read on RAM: [93, 29, 106, 181, 146, 167, 14, 155, 34, 186,
    91, 70, 217, 253, 101, 147, 248, 178, 56, 61]
>> Data written on RAM: [211, 109, 137, 254, 16, 126, 97, 124, 198,
    8, 113, 138, 116, 217, 169, 181, 204, 77, 5, 98, 137, 36, 83,
    52]
>> Data read on RAM: [211, 109, 137, 254, 16, 126, 97, 124, 198, 8,
    113, 138, 116, 217, 169, 181, 204, 77, 5, 98, 137, 36, 83, 52]
>> Data written on RAM: [87, 196, 138, 133, 37, 128, 9, 26, 20, 104,
     217, 155, 144, 77, 155, 67, 89, 224, 239, 19, 121, 2, 203, 177,
     152, 171, 20, 44]
>> Data read on RAM: [87, 196, 138, 133, 37, 128, 9, 26, 20, 104,
    217, 155, 144, 77, 155, 67, 89, 224, 239, 19, 121, 2, 203, 177,
    152, 171, 20, 44]
>> Data written on RAM: [69, 101, 67, 152, 83, 177, 102, 30, 51, 61,
     242, 81, 50, 65, 155, 237, 85, 154, 105, 250, 76, 33, 73, 15,
    65, 226, 157, 132, 219, 51, 172, 201]
>> Data read on RAM: [69, 101, 67, 152, 83, 177, 102, 30, 51, 61,
    242, 81, 50, 65, 155, 237, 85, 154, 105, 250, 76, 33, 73, 15,
    65, 226, 157, 132, 219, 51, 172, 201]
>> Data written on RAM: [197, 93, 86, 253, 53, 197, 88, 34, 217, 9,
    246, 78, 156, 43, 112, 74, 200, 54, 121, 229, 110, 206, 103,
    242, 245, 164, 107, 61, 12, 58, 78, 232, 115, 78, 124, 95]
```

```
>> Data read on RAM: [197, 93, 86, 253, 53, 197, 88, 34, 217, 9,
    246, 78, 156, 43, 112, 74, 200, 54, 121, 229, 110, 206, 103,
    242, 245, 164, 107, 61, 12, 58, 78, 232, 115, 78, 124, 95]
>> Data written on RAM: [125, 238, 64, 48, 247, 146, 114, 137, 148,
    3, 124, 106, 118, 14, 132, 192, 46, 175, 209, 185, 149, 244, 74,
     29, 161, 232, 188, 127, 195, 73, 83, 112, 70, 92, 153, 42, 227,
     19, 224, 182]
>> Data read on RAM: [125, 238, 64, 48, 247, 146, 114, 137, 148, 3,
    124, 106, 118, 14, 132, 192, 46, 175, 209, 185, 149, 244, 74,
    29, 161, 232, 188, 127, 195, 73, 83, 112, 70, 92, 153, 42, 227,
    19, 224, 182]
>> Data written on RAM: [192, 44, 245, 205, 34, 239, 158, 230, 232,
    90, 136, 216, 193, 169, 234, 187, 51, 225, 64, 210, 26, 187, 92,
     10, 254, 84, 50, 130, 188, 49, 39, 140, 75, 73, 247, 78, 62,
    173, 95, 69, 209, 82, 217, 182]
>> Data read on RAM: [192, 44, 245, 205, 34, 239, 158, 230, 232, 90,
     136, 216, 193, 169, 234, 187, 51, 225, 64, 210, 26, 187, 92,
    10, 254, 84, 50, 130, 188, 49, 39, 140, 75, 73, 247, 78, 62,
    173, 95, 69, 209, 82, 217, 182]
>> Data written on RAM: [191, 82, 114, 84, 213, 31, 158, 43, 123,
    157, 64, 89, 252, 255, 135, 216, 65, 96, 230, 87, 194, 91, 28,
    93, 115, 96, 188, 144, 37, 164, 69, 244, 147, 20, 71, 66, 106,
    133, 158, 187, 243, 79, 11, 142, 228, 202, 143, 196]
>> Data read on RAM: [191, 82, 114, 84, 213, 31, 158, 43, 123, 157,
    64, 89, 252, 255, 135, 216, 65, 96, 230, 87, 194, 91, 28, 93,
    115, 96, 188, 144, 37, 164, 69, 244, 147, 20, 71, 66, 106, 133,
    158, 187, 243, 79, 11, 142, 228, 202, 143, 196]
>> Data written on RAM: [77, 8, 3, 120, 67, 240, 238, 86, 162, 194,
    248, 181, 175, 170, 210, 111, 138, 180, 45, 201, 201, 24, 141,
    217, 213, 139, 193, 188, 55, 47, 31, 158, 27, 179, 163, 134, 64,
     241, 249, 147, 2, 70, 168, 200, 100, 117, 34, 59, 4, 167, 132,
    125]
>> Data read on RAM: [77, 8, 3, 120, 67, 240, 238, 86, 162, 194,
    248, 181, 175, 170, 210, 111, 138, 180, 45, 201, 201, 24, 141,
    217, 213, 139, 193, 188, 55, 47, 31, 158, 27, 179, 163, 134, 64,
     241, 249, 147, 2, 70, 168, 200, 100, 117, 34, 59, 4, 167, 132,
    125]
>> Data written on RAM: [116, 197, 134, 198, 181, 86, 240, 213, 201,
     183, 12, 210, 63, 37, 85, 10, 172, 168, 244, 136, 243, 83, 50,
    67, 207, 172, 60, 15, 225, 208, 113, 223, 38, 198, 157, 255, 98,
     172, 85, 135, 193, 170, 161, 125, 229, 71, 207, 71, 125, 33,
    106, 79, 93, 209, 96, 19]
>> Data read on RAM: [116, 197, 134, 198, 181, 86, 240, 213, 201,
    183, 12, 210, 63, 37, 85, 10, 172, 168, 244, 136, 243, 83, 50,
    67, 207, 172, 60, 15, 225, 208, 113, 223, 38, 198, 157, 255, 98,
     172, 85, 135, 193, 170, 161, 125, 229, 71, 207, 71, 125, 33,
    106, 79, 93, 209, 96, 19]
>> Data written on RAM: [60, 233, 147, 193, 65, 255, 12, 122, 69,
    117, 157, 32, 209, 46, 185, 2, 177, 179, 226, 3, 39, 40, 22,
    126, 5, 223, 141, 31, 50, 78, 35, 170, 227, 54, 246, 218, 44,
    12, 82, 90, 221, 148, 246, 66, 62, 5, 186, 215, 31, 29, 174,
    141, 32, 231, 52, 179, 128, 33, 182, 231]
>> Data read on RAM: [60, 233, 147, 193, 65, 255, 12, 122, 69, 117,
    157, 32, 209, 46, 185, 2, 177, 179, 226, 3, 39, 40, 22, 126, 5,
    223, 141, 31, 50, 78, 35, 170, 227, 54, 246, 218, 44, 12, 82,
    90, 221, 148, 246, 66, 62, 5, 186, 215, 31, 29, 174, 141, 32,
    231, 52, 179, 128, 33, 182, 231]
>> Data written on RAM: [205, 38, 116, 28, 233, 202, 164, 192, 225,
    178, 32, 206, 195, 79, 116, 191, 146, 243, 38, 4, 83, 135, 168,
    183, 163, 49, 206, 178, 247, 13, 20, 112, 193, 37, 253, 27, 200,
```

```
     214, 135, 243, 134, 143, 1, 179, 18, 117, 147, 73, 17, 220, 19,
     71, 235, 56, 164, 103, 223, 221, 39, 119, 111, 34, 37, 86]
>> Data read on RAM: [205, 38, 116, 28, 233, 202, 164, 192, 225,
    178, 32, 206, 195, 79, 116, 191, 146, 243, 38, 4, 83, 135, 168,
    183, 163, 49, 206, 178, 247, 13, 20, 112, 193, 37, 253, 27, 200,
     214, 135, 243, 134, 143, 1, 179, 18, 117, 147, 73, 17, 220, 19,
     71, 235, 56, 164, 103, 223, 221, 39, 119, 111, 34, 37, 86]
>> RAM test succesful!
```

Listing 6: RAM test