Microprocessors

Serial buses - UART, SPI, I2C

Stanislav Vítek
Department of Radioelectronics
Czech Technical University in Prague

UART - Universal Asynchronous Receiver Transmitter

- Asynchronous serial communication in which the data format and transmission speeds are configurable.
- It sends data bits one by one, from the least significant to the most significant, framed by start and stops bits so that the communication channel handles precise timing.
- A driver circuit external to the UART handles the electric signaling levels.
- UART interfaces have a maximum data rate of around 5 Mbps.
- There is also some protocol overhead in the form of start, stop, and parity bits.

UART interface

- TxD Transmit Data
- RxD Receive Data

The polarity of the signals is +5 V for mark or high and 0 V for space or low. However, serial data transmission along the wire in an RS232 transmission interface requires −15 to −12 V for mark and +12 to +15 V for space. Line drivers are used to convert the logic levels required by the UART to those needed for the RS232 interface pins TD and RD.

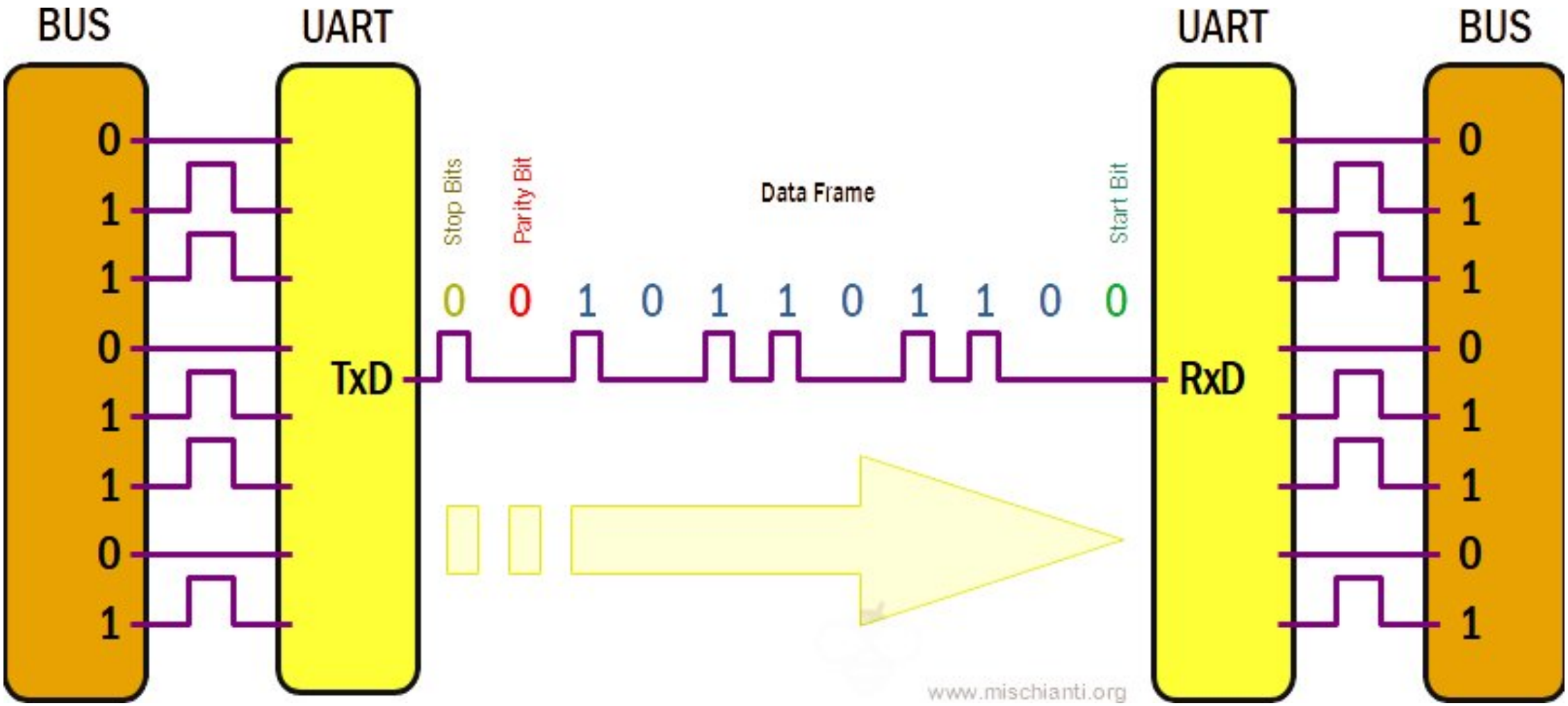
UART properties

- It can operate between devices in 3 ways:
  - Simplex = data transmission in one direction
  - Half-duplex = data transmission in either direction but not simultaneously
  - Full-duplex = data transmission in both directions simultaneously

Distance

A UART can work over long distances with appropriate line drivers: from 15 meters (m) for the RS-232 serial data bus to 1000m for RS-485 or RS-422 interfaces. The distance is also influenced by cable quality and baud rate.

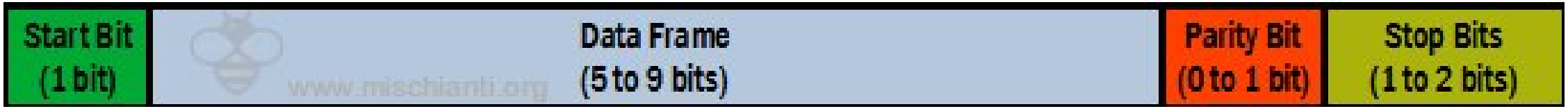
UART communication



UART communication

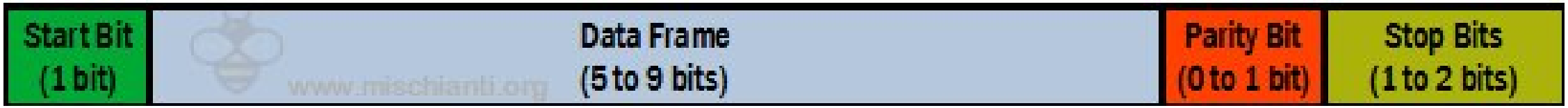
- Transmitting UART converts parallel data from the primary device (e.g. CPU) into serial form and transmits it to receiving UART. It will then convert the serial data back into parallel data for the receiving device.
- As UART has no clocks, UART adds start and stop bits that are being transferred.
- This helps the receiving UART know when to start reading bits as the bits represent the start and the end of the data packet. When the receiving UART detects a start bit, it will read the bits at the BAUD rate.
- UART data transmission speed is called BAUD Rate and is set to 115,200 by default (BAUD rate is based on symbol transmission rate but is similar to bit rate).

UART packet 1/2



- **Start Bit:** UART data transmission line is usually held at a high voltage level when it's not transmitting data. The transmitting UART pulls the transmission line from high to low for one clock cycle to start the transfer of data.
- Data Frame: contains the actual data being transferred. If a parity bit is used, it can be 5 bits up to 8 bits long. The data frame can be 9 bits long if no parity bit is used. In most cases, the data is first sent with the least significant bit.

UART packet 2/2



- **Parity:** describes the evenness or oddness of a number. The parity bit allows the receiving UART to tell if any data has changed during transmission. It counts the number of bits with a value of 1 and checks if the total is an even or odd number.
- **Stop Bits:** to signal the end of the data packet, the sending UART drives the data transmission line from a low voltage to a high voltage for 1 to two 2 bits duration.

UART on RP2

- UART is described by the machine.UART class.
- RP2040 has two UART peripherals (UART0 and UART1).
  - Programmable data length (5-8 bits) and the number of stop bits (1 or 2).
  - FIFO in both directions up to 32 bytes.
- Interrupts can be used to monitor data arrival or departure, device status, communication error, or data reception timeout.
- Both devices can be configured on various pairs of TX and RX pins.
  - UART0: GP0-GP1, GP12-GP13, GP16-GP17
  - UART1: GP4-GP5, GP8-GP9

UART

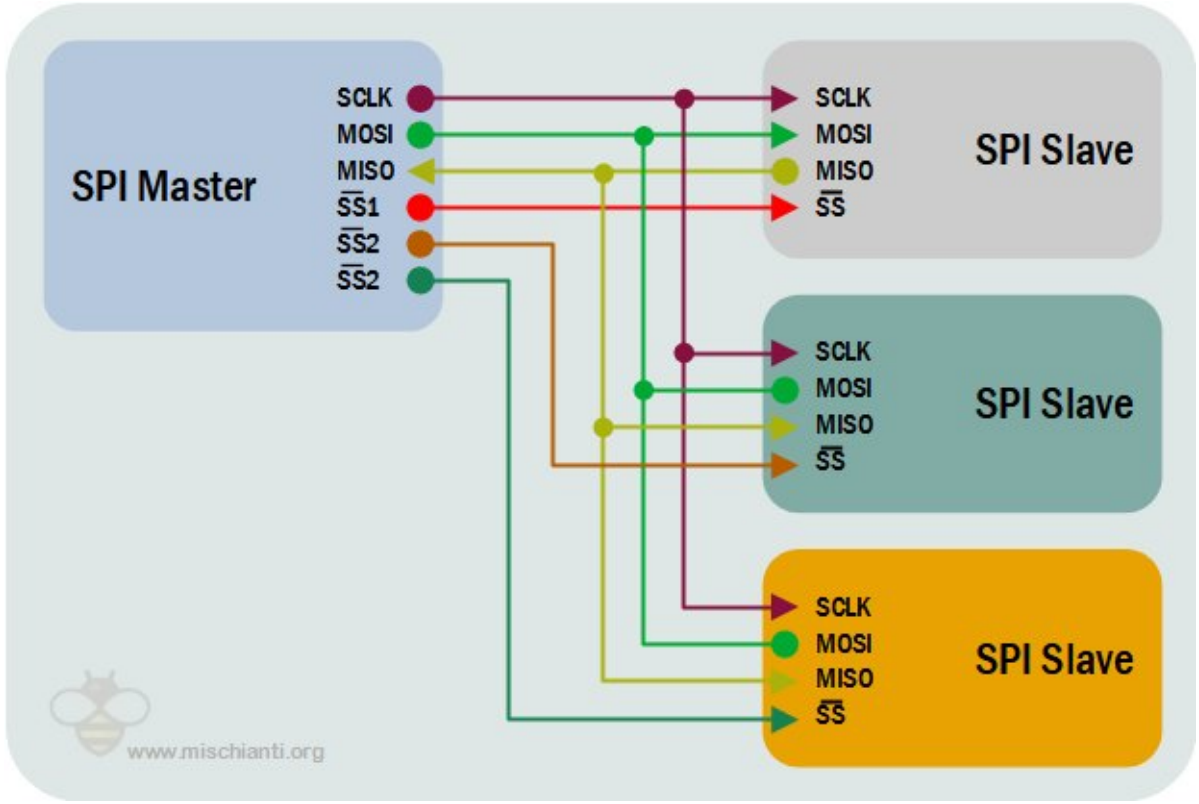
```
from machine import Pin, UART
import time

uart = UART(1, baudrate=9600, tx=Pin(4), rx=Pin(5))
uart.init(bits=8, parity=None, stop=2)

led = Pin("LED", Pin.OUT)

while True:
    uart.write('t')
    if uart.any():
        data = uart.read()
        if data== b'm':
            led.toggle()
    time.sleep(1)
```

SPI - Serial peripheral bus

- Developed by Motorola, it has no official specification; used by many companies; 4 different communication modes (SPI 0 … SPI 3)
- SPI dedicated to fast communication among IC chips, so that it is used just only on printed circuit boards, or for communication with nearby peripherals
- Designed for synchronous serial communication between the microcontroller and peripheral circuits
  - A/D converters (programming, not data transmission - Intersil KAD5512, …), slower D/A converters (Maxim)
  - Potentiometers (Intersil ISL22424, …)
  - MMC and SD cards, EEPROMs
  - FLASH memories (BIOS on PC mainboards, connected via ICH)
  - EF lens Canon

SPI

- Serial bus, full duplex
- Single master, multiple slave
- MISO, MOSI, SCLK, SS/CS
- Typical speed of 10 Mbps
- Distance up to 10 meters
- Simple implementation
  - No need for addressing
- Disadvantages
  - More devices = more wires
  - No data control



SPI

- If the processor has no hardware support for SPI interface, it can be implemented in software using four pins of one of its ports
- Data are latched on an edge of the clock signal SCK (master) and updated on opposite edge of SCK (slave)
- Signals:
  - SCK: Serial Clock, generated by Master
  - MOSI/SDI: Master Output Slave Input/Serial Data In
  - MISO/SDO: Master Input Slave Output/Serial Data Out
  - CS: Chip Select – selects the device, substitute an addressing on master sometimes called SS (Slave Select)
- 1 master / 1 or more slave devices, multimaster possible but complicated

SPI in MicroPython

- The SPI bus is described by the machine.SPI class, supporting both software and hardware implementations.

```
from machine import SPI, Pin

spi = SPI(0, baudrate=400000)        # Create SPI peripheral 0 at frequency of 400kHz.
                                     # Depending on the use case, extra parameters may be required
                                     # to select the bus characteristics and/or pins to use.
cs = Pin(4, mode=Pin.OUT, value=1)   # Create chip-select on pin 4.

try:
  cs(0)                              # Select peripheral.
  spi.write(b"12345678")             # Write 8 bytes, and don't care about received data.
finally:
   cs(1)                             # Deselect peripheral.
```

SPI - constructors and methods

- Hardware SPI

```
class machine.SPI(id, ...)
```

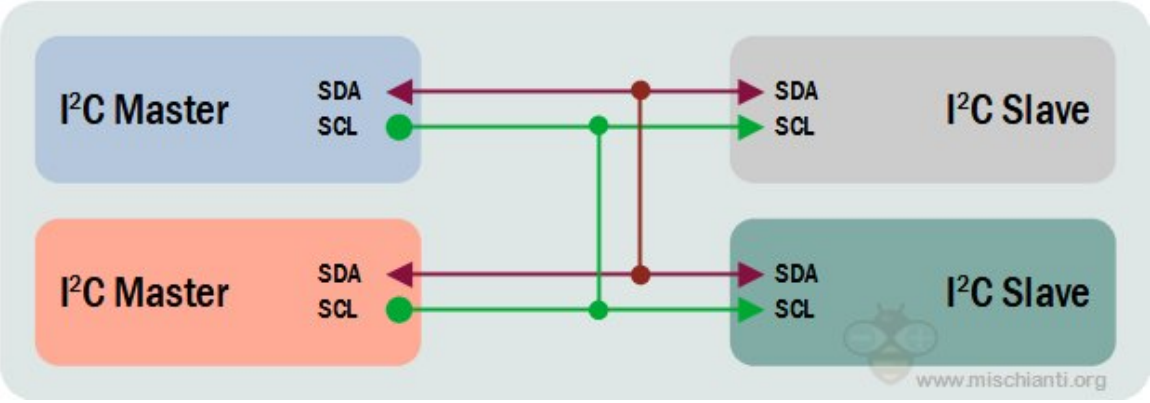
- Software SoftSPI

```
class machine.SoftSPI(baudrate=500000, *, polarity=0, phase=0, bits=8,
                      firstbit=MSB, sck=None, mosi=None, miso=None)
```

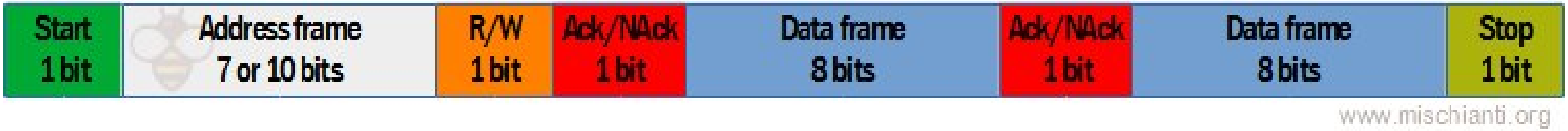
- init init(), deinit()
- write read(), readinto()
- read write(), write_readinto()

I2C

- Serial synchronous bus, half duplex
  - Multimaster, multislave, two wires: SDA, SCL
  - Supports speeds of 100 kbps, 400 kbps, and 3.4 Mbps (or 10 kbps and 1 Mbps)
  - Typically used for shorter distances, with a maximum of around 30 cm.



I2C Protocol



- START: SDA transitions from HIGH to LOW before SCL transitions from HIGH to LOW.
- Stop: SDA transitions from LOW to HIGH after SCL transitions from LOW to HIGH.
- Address Frame: A 7 or 10-bit address of the slave with which the master wants to communicate.
- Read/Write Bit: A bit that determines whether the master is sending data to the slave (LOW) or requesting data from it (HIGH).
- ACK/NACK Bit: After each frame in the message, there is an acknowledgment/non-acknowledgment bit. If the address or data frame was successfully received, the receiving device returns an ACK bit to the sender.

I2C in MicroPython

- The I2C bus is described by the machine.I2C class.

```
from machine import I2C

i2c = I2C(freq=400000)           # create I2C peripheral at frequency of 400kHz
                                 # depending on the port, extra parameters may be required
                                 # to select the peripheral and/or pins to use

i2c.scan()                       # scan for peripherals, returning a list of 7-bit addresses

i2c.writeto(42, b'123')          # write 3 bytes to peripheral with 7-bit address 42
i2c.readfrom(42, 4)              # read 4 bytes from peripheral with 7-bit address 42

i2c.readfrom_mem(42, 8, 3)       # read 3 bytes from memory of peripheral 42,
                                 #   starting at memory-address 8 in the peripheral
i2c.writeto_mem(42, 2, b'\x10') # write 1 byte to memory of peripheral 42
                                 #   starting at address 2 in the peripheral
```

I2C - constructors and methods

- Hardware I2C

```
class machine.I2C(id, *, scl, sda, freq=400000)
```

- Software SoftI2C

```
class machine.SoftI2C(scl, sda, *, freq=400000, timeout=50000)
```

- Initialization init() and deinitialization deinit()
- Scanning scan()
  - Scans all I2C addresses between 0x08 and 0x77, inclusive, and returns a list of those that respond.
  - A device responds if, after sending its address (including the write bit), it pulls the SDA line.

I2C - Low-Level Functions of the SoftI2C Class

- start() - START condition (SDA transitions to LOW while SCL is HIGH).
- stop() - STOP condition (SDA transitions to HIGH while SCL is HIGH).
- readinto() - reads data from the bus and stores it in a buffer.
  - The number of bytes read corresponds to the length of the buffer.
  - After receiving all bytes except the last one, an ACK is sent on the bus. By sending NACK or ACK after receiving the last byte, you can determine whether to read data in a subsequent call.
- write() - writes data from the buffer to the bus.
  - It checks if an ACK is received after each byte, and in the case of receiving NACK, it interrupts the write of the remaining data. The function returns the number of received ACKs.

Standard I2C operations - read

```
I2C.readfrom(addr, nbytes, stop=True, /)
```

- readfrom() - reads nbytes from the peripheral specified by addr.
  - If stop is true, a STOP condition is generated at the end of the transfer.
  - It returns a bytes object with the read data.

```
I2C.readfrom_into(addr, buf, stop=True, /)
```

- readfrom_into() - reads into buf from the peripheral specified by addr.
  - The number of bytes read will match the length of buf.
  - If stop is true, a STOP condition is generated at the end of the transfer.

Standard I2C operations - write

- writeto() - writes bytes from buf to the peripheral specified by addr. If a NACK is received after writing a byte from buf, the remaining bytes will not be sent. If stop is true, a STOP condition is generated at the end of the transfer, even if a NACK is received. The function returns the number of received ACKs.

```
I2C.writeto(addr, buf, stop=True, /)
```

- writevto() - writes bytes contained in the vector to the peripheral specified by addr. The vector should be a tuple or list of objects following the buffer protocol. Addr is sent once, and then bytes from each object in the vector are successively written. Objects in the vector can have zero length, in which case they do not contribute to the output.

```
I2C.writevto(addr, vector, stop=True, /)
```