Programmering af Mobile Robotter

RB1-PMR – Module 9: Data Communication

Jes Hundevadt Jepsen | jegj@mmmi.sdu.dk
SDU Drone Center, MMMI, University of Southern Denmark, Odense, Denmark

Agenda

- Recap of last module

- Communication interfaces
  - General introduction to data communication
    - Parallel / Serial
    - Synchronous/ Asynchronous
    - Simplex / Duplex

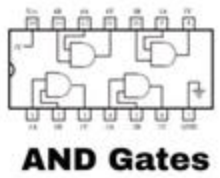
  - Interfaces
    - I^2C (Inter-Integrated Circuit)
    - Serial Peripheral Interface (SPI)
    - Universal Asynchronous Receiver-Transmitter (UART)

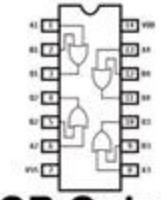
- Introduction to:
  - **Portfolio 4:** Controller for a Mobile Robot (Open-ended Mini Project), and
  - **Extra Credit Activities 4:** Present your Mini Project

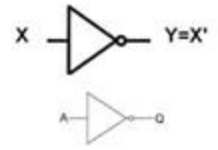
- **Remember:** Fokusgruppeinterview (25/11, 8.15 - 9.15)

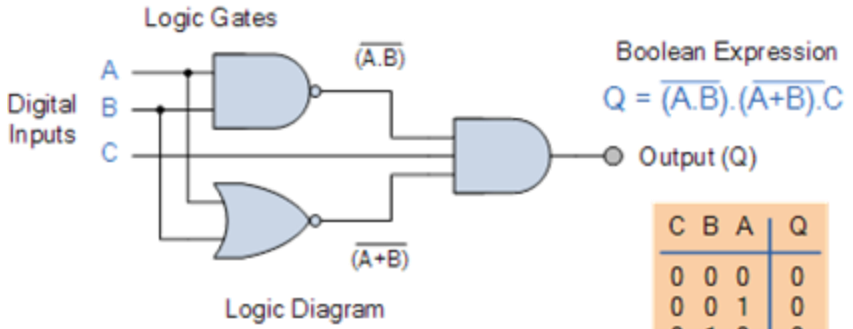
Recap

- Logic gates
  - NOT, AND, OR, NAND, NOR, XOR, XNOR
- Combinational Logic Circuits
  - Logic diagram
  - Truth Table
  - Boolean expression
  - Common examples
    - Multiplexers, decode/encoder, full adder
- The Laws of Boolean Algebra
  - How to simplify logic circuits



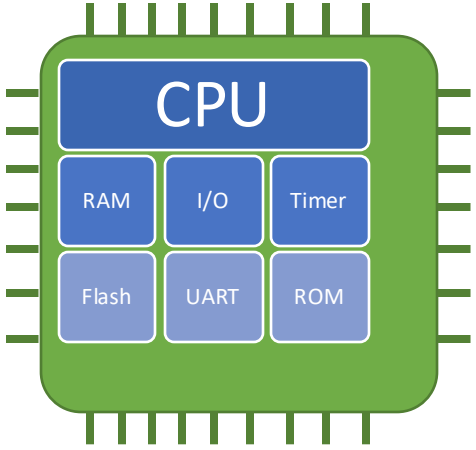


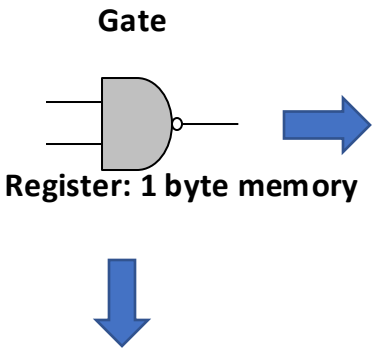


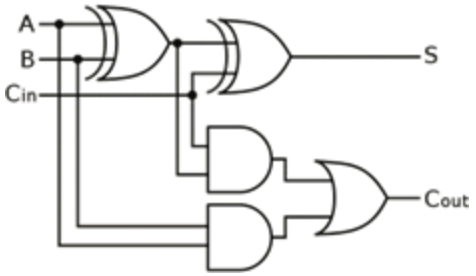


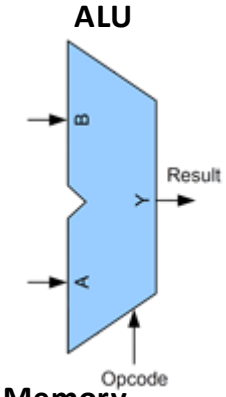
Typical Truth Table

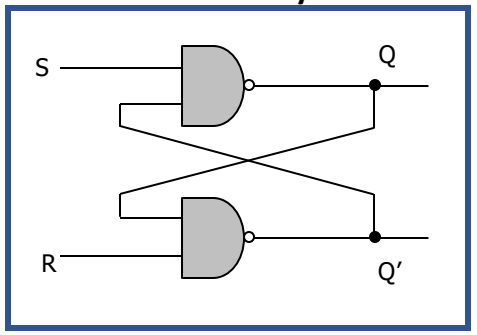
C	B	A	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

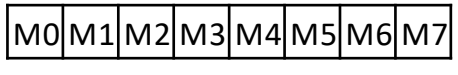












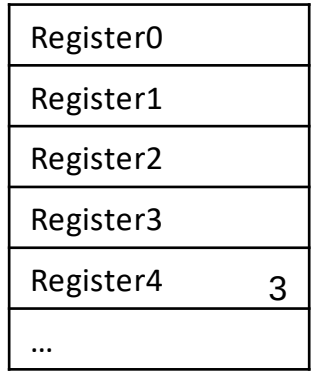


Illustration: Datateknik course, W2-W4 (T540050101, by Ali Sahafi)

Communication interfaces

Communication (Module 1)

"The process of exchanging information, ideas, thoughts, or messages between individuals, groups, or systems through a shared medium"

*= requires a **sender**, a message, and a **receiver** using the same common understanding*

Humans?

- Language, writing/symbols, appearance/body language, etc.

Machines?

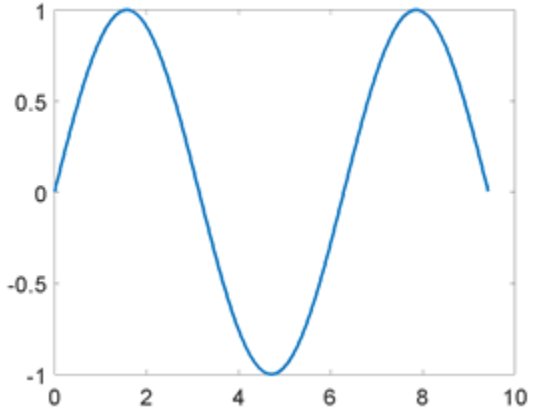
- Protocols, writing/symbols, data, etc.
  - → **signals**

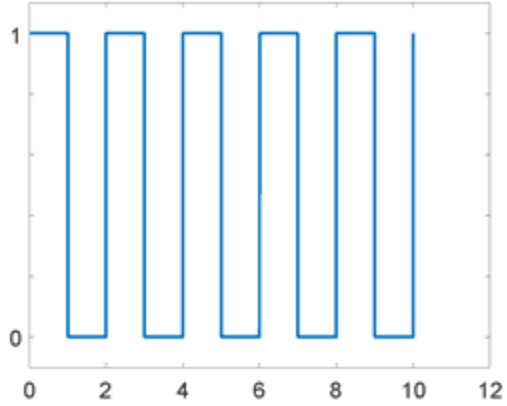
Signals (Module 1)

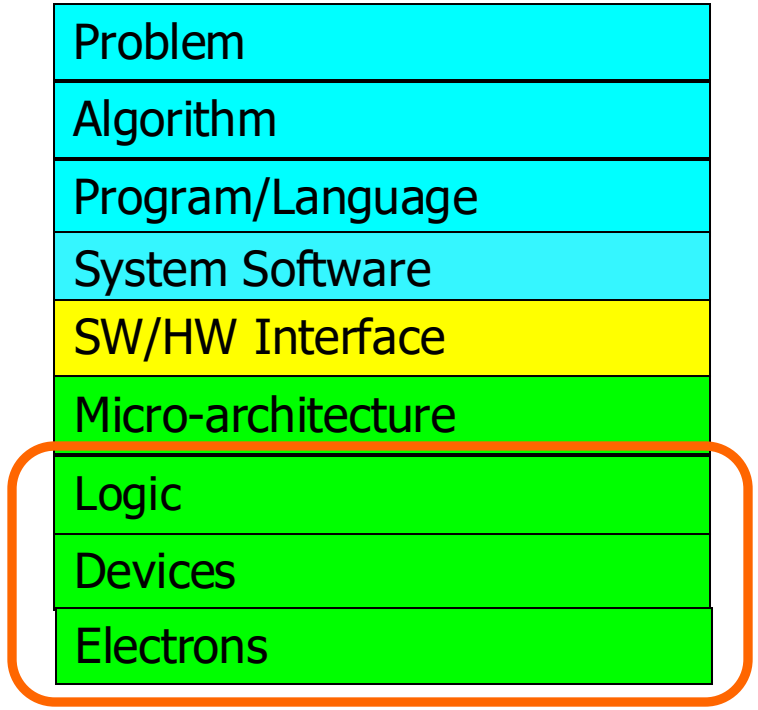
"An electrical or electromagnetic current that is used for carrying data from one device or network to another"

- There are two main types of signal:

Analog	Digital
Analog signal is continuous in time and amplitude.	Digital signals are discrete in time and amplitude.
A continuous range of values with an infinite number of possible values.	Binary values (**high (1)** and **low (0)**) to represent information.
Examples: Analog sensors (temperature, light, etc.), radio signals,	**Examples:** Computer data, CDs, DVDs







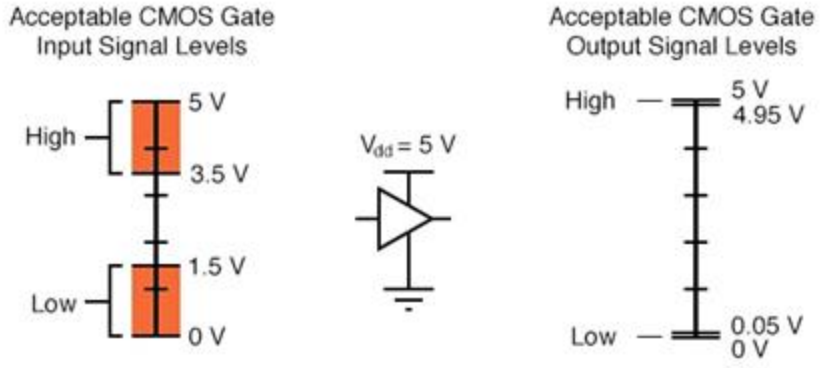
In principle analog could be fine… **but it is not…**

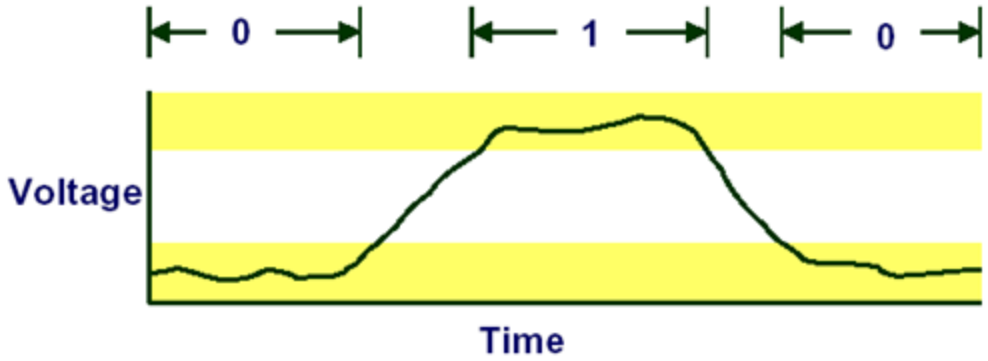
Logic Signal Voltage Levels

- Logic gate circuits are designed to input and output only two types of signals: **High (1) / Low (0)**

- **Example:** "Acceptable" signal voltages range
  - From 0 volts to < 0.8 - 1.5 volts for a "low" logic state
  - From > 2 - 3.5 volts to 5 volts for a "high" logic state

- Ranging between low and high is considered as uncertain



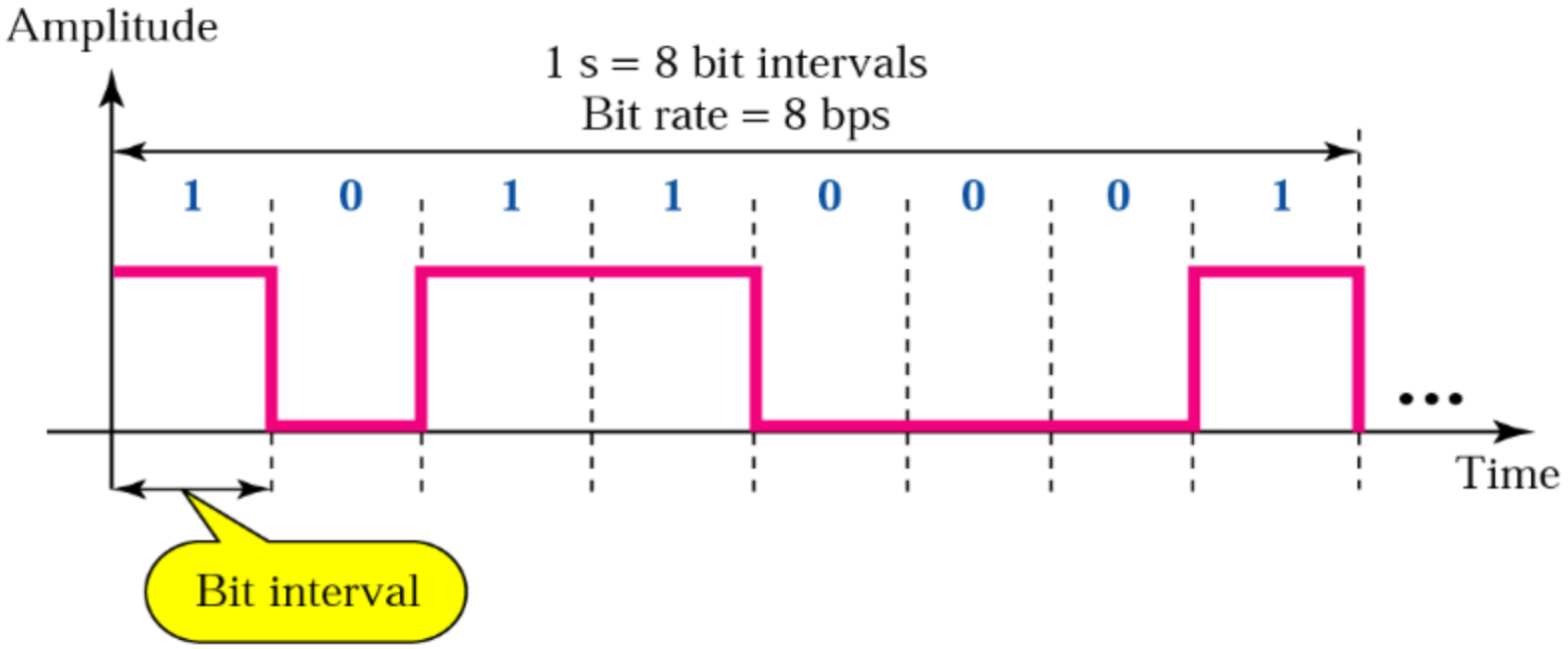


Images: https://www.allaboutcircuits.com/textbook/digital/chpt-3/logic-signal-voltage-levels/
https://people.engr.tamu.edu/djimenez/classes/312/lecture2.html

Data (Module 1)

"A sequence of bits, either 0 or 1, bits arranged in complex or specific patterns"

Hard for the human to read…



* bits per second(bps)

Images: https://bmsce.ac.in/Content/CS/unit1_CN_LJJ.pdf

Data

"A sequence of bits, either 0 or 1, bits arranged in complex or specific patterns"

…can be transferred using two methods:

Illustration: Datateknik course, W8 (T540050101, by Ali Sahafi)

Data

"A sequence of bits, either 0 or 1, bits arranged in complex or specific patterns"

…can be transferred using two methods:

- Parallel
  - All the bits are transferred simultaneously
  - High speed
  - Need lot of wires:
    - Impractical for long distances
  - **Example:** Old printers

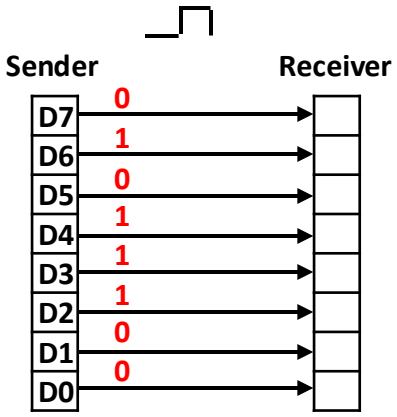
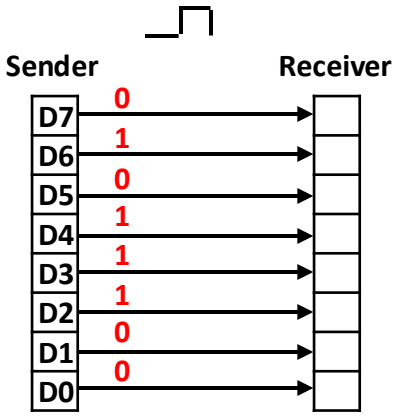


Illustration: Datateknik course, W8 (T540050101, by Ali Sahafi)

Data

"A sequence of bits, either 0 or 1, bits arranged in complex or specific patterns"



…can be transferred using two methods:

- Parallel
  - All the bits are transferred simultaneously
  - High speed
  - Need lot of wires:
    - Impractical for long distances
  - **Example:** Old printers

- Serial
  - Bits are sent sequentially
  - Theoretically lower communication speed
  - Use common wire for all bits
  - **Example:** USB, Morse code

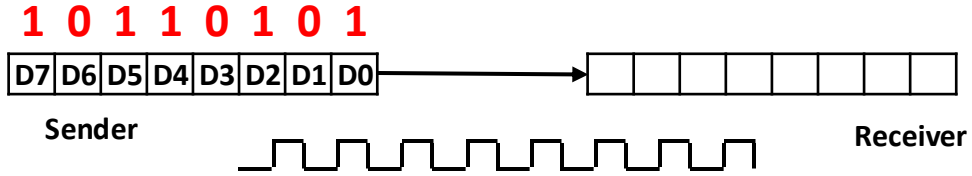


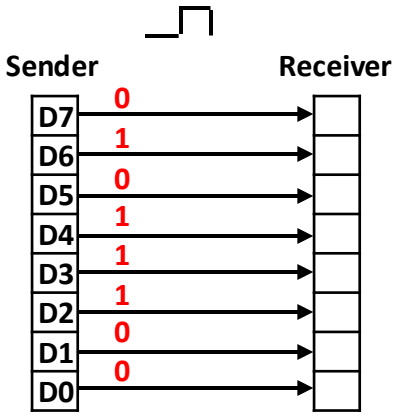
Illustration: Datateknik course, W8 (T540050101, by Ali Sahafi)

Data

"A sequence of bits, either 0 or 1, bits arranged in complex or specific patterns"

…can be transferred using two methods:

- ~~Parallel~~
  - ~~All the bits are transferred simultaneously~~
  - ~~High speed~~
  - ~~Need lot of wires:~~
    - ~~Impractical for long distances~~
  - ~~**Example:** Old printers~~



- **Serial**
  - Bits are sent sequentially
  - Theoretically lower communication speed
  - Use common wire for all bits
  - **Example:** USB, Morse code

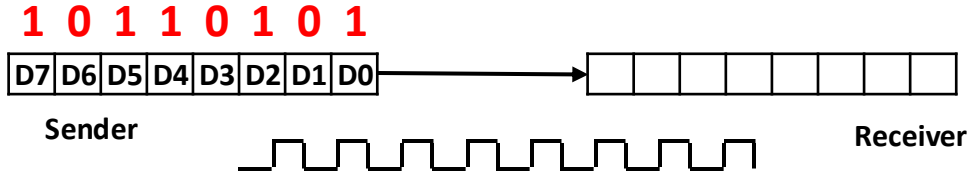


Illustration: Datateknik course, W8 (T540050101, by Ali Sahafi)

Data

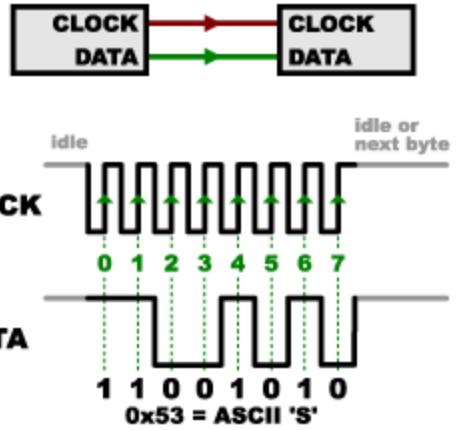
"A sequence of bits, either 0 or 1, bits arranged in complex or specific patterns"

*…serial data transfer can be be done using <u>two **methods**</u>:*

Illustration: Datateknik course, W8 (T540050101, by Ali Sahafi)

Data

"A sequence of bits, either 0 or 1, bits arranged in complex or specific patterns"



…serial data transfer can be be done using <u>two methods</u>:

- **Synchronous**
  - There is **a common clock line** between the Transmitter and Receiver
  - Data are captured at clock transition

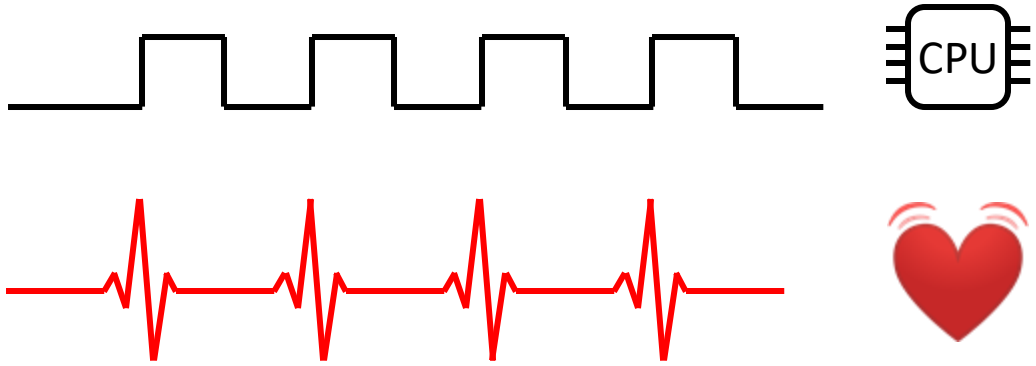


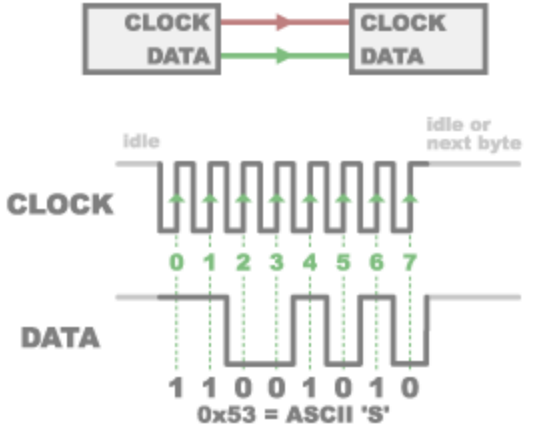
Illustration: Datateknik course, W8 (T540050101, by Ali Sahafi)

Data

"A sequence of bits, either 0 or 1, bits arranged in complex or specific patterns"

…serial data transfer can be be done using <u>two methods</u>:

- Synchronous
  - There is a common clock line between the Transmitter and Receiver
  - Data are captured at clock transition



- **Asynchronous**
  - **No common clock** line between Transmitter and Receiver
  - Data are captured using internal clock <u>based on predefined communication speed</u>

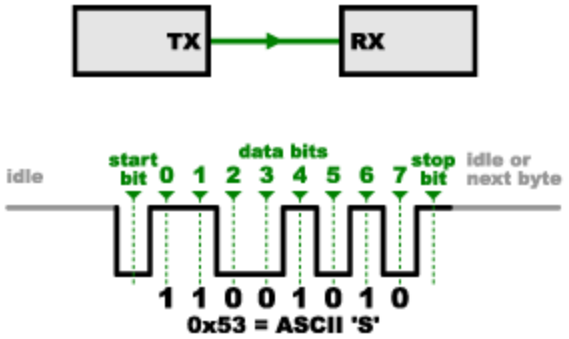


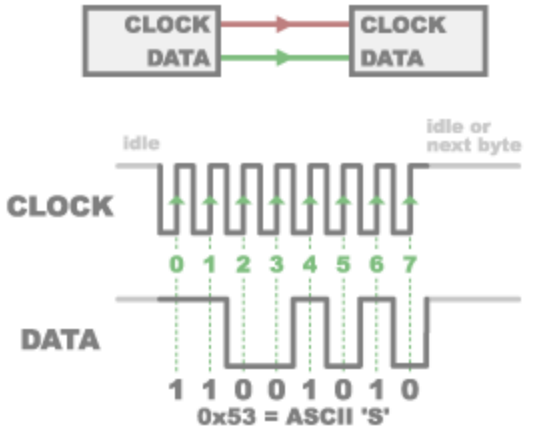
Illustration: Datateknik course, W8 (T540050101, by Ali Sahafi)

Data

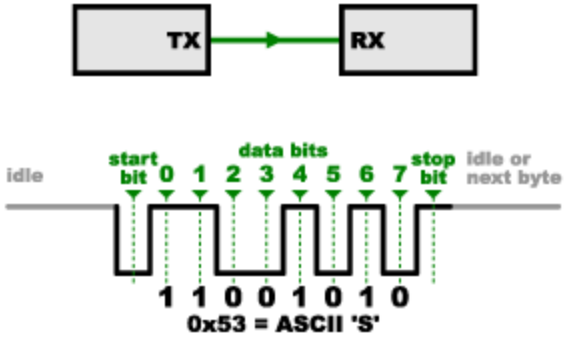
"A sequence of bits, either 0 or 1, bits arranged in complex or specific patterns"

…serial data transfer can be be done using <u>two methods</u>:

- **Synchronous ?**
  - There is a common clock line between the Transmitter and Receiver
  - Data are captured at clock transition



- **Asynchronous ?**
  - **No common clock** line between Transmitter and Receiver
  - Data are captured using internal clock <u>based on predefined communication speed</u>



<u>Examples?</u>

Illustration: Datateknik course, W8 (T540050101, by Ali Sahafi)

Data

"A sequence of bits, either 0 or 1, bits arranged in complex or specific patterns"

*…serial data transfer can be be done using <u>two **modes**</u>:*

Illustration: Datateknik course, W8 (T540050101, by Ali Sahafi)

Data

"A sequence of bits, either 0 or 1, bits arranged in complex or specific patterns"

…serial data transfer can be be done using two modes:

- **Duplex**: Data can be transmitted and Received

Illustration: Datateknik course, W8 (T540050101, by Ali Sahafi)

Data

"A sequence of bits, either 0 or 1, bits arranged in complex or specific patterns"

…serial data transfer can be be done using <u>two modes</u>:

- **Duplex**: Data can be transmitted and Received
  - **<u>Full Duplex:</u>** Data can be sent and received simultaneously
    - Needs two wires for data

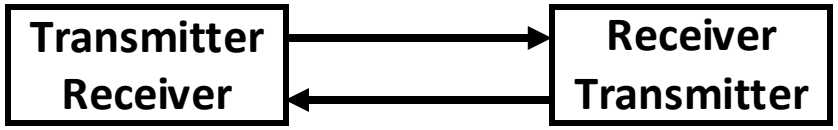


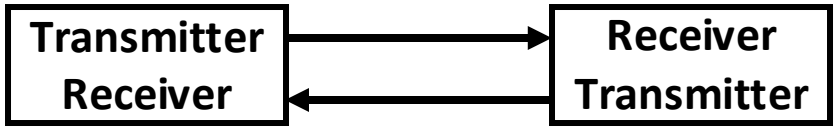
Illustration: Datateknik course, W8 (T540050101, by Ali Sahafi)

Data

"A sequence of bits, either 0 or 1, bits arranged in complex or specific patterns"

…serial data transfer can be be done using <u>two modes</u>:

- **Duplex**: Data can be transmitted and Received
  - Full Duplex: Data can be sent and received simultaneously
    - Needs two wires for data



  - **<u>Half Duplex:</u>** Data can not be sent and received simultaneously
    - Needs one shared wire for data

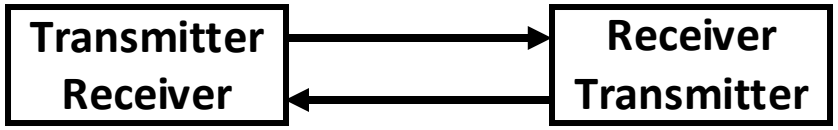
Illustration: Datateknik course, W8 (T540050101, by Ali Sahafi)

Data

"A sequence of bits, either 0 or 1, bits arranged in complex or specific patterns"

…serial data transfer can be be done using <u>two modes</u>:

- Duplex: Data can be transmitted and Received
  - Full Duplex: Data can be sent and received simultaneously
    - Needs two wires for data



  - Half Duplex: Data can not be sent and received simultaneously
    - Needs one shared wire for data

- **Simplex:** Data is only transmitted in one way

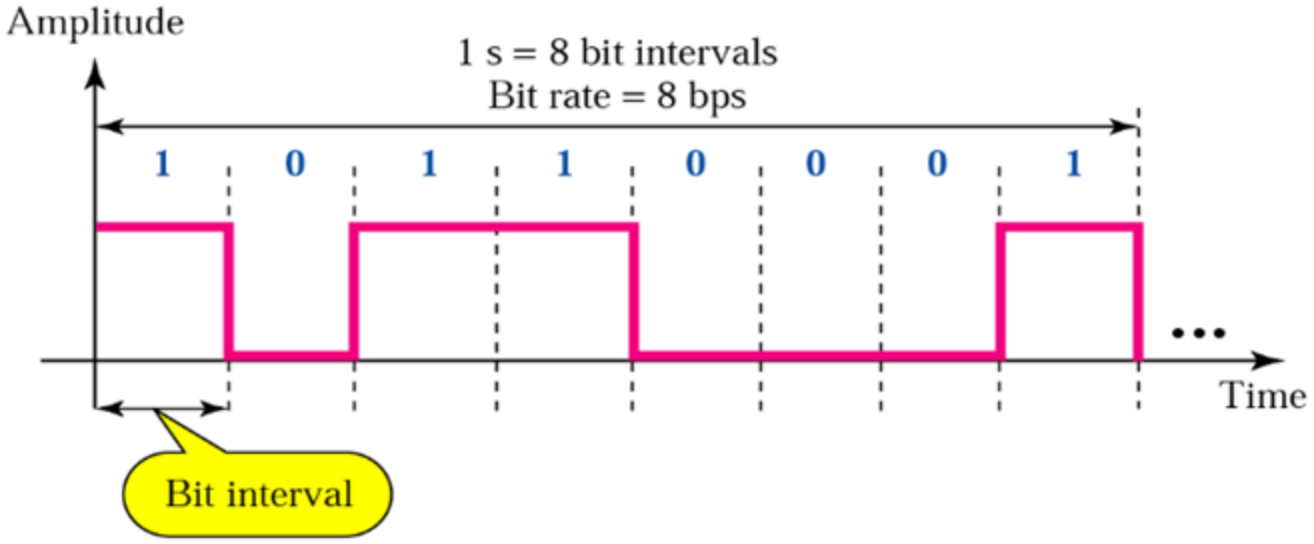
Illustration: Datateknik course, W8 (T540050101, by Ali Sahafi)

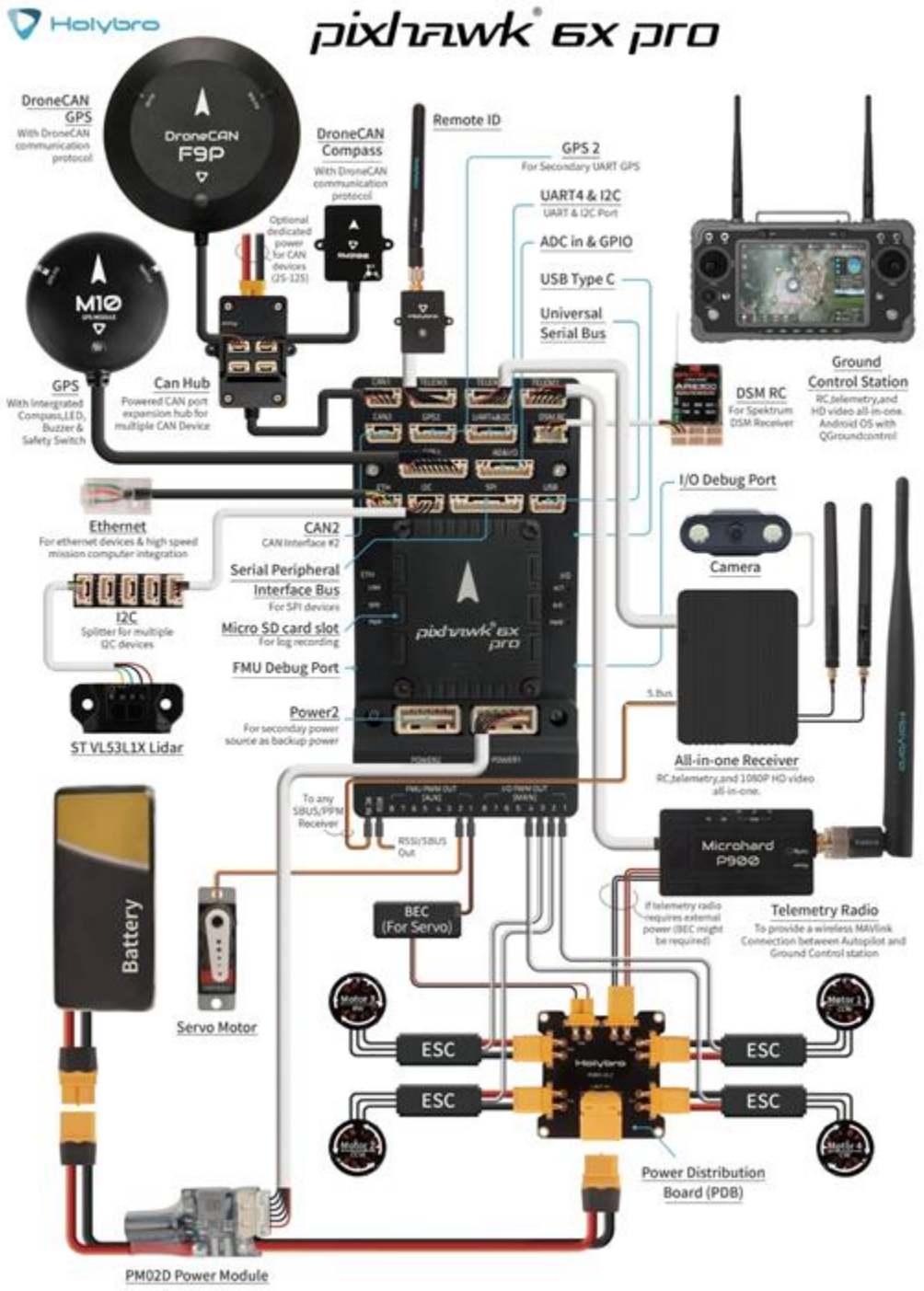
Protocol (Communication)

"...a set of standardized rules and procedures describing how to transmit or exchange data between (electronic) devices."

Data: …hard for the human to read…

Protocol: …defines how data is organized, structured, and represented during data transmission…





Images: https://bmsce.ac.in/Content/CS/ur

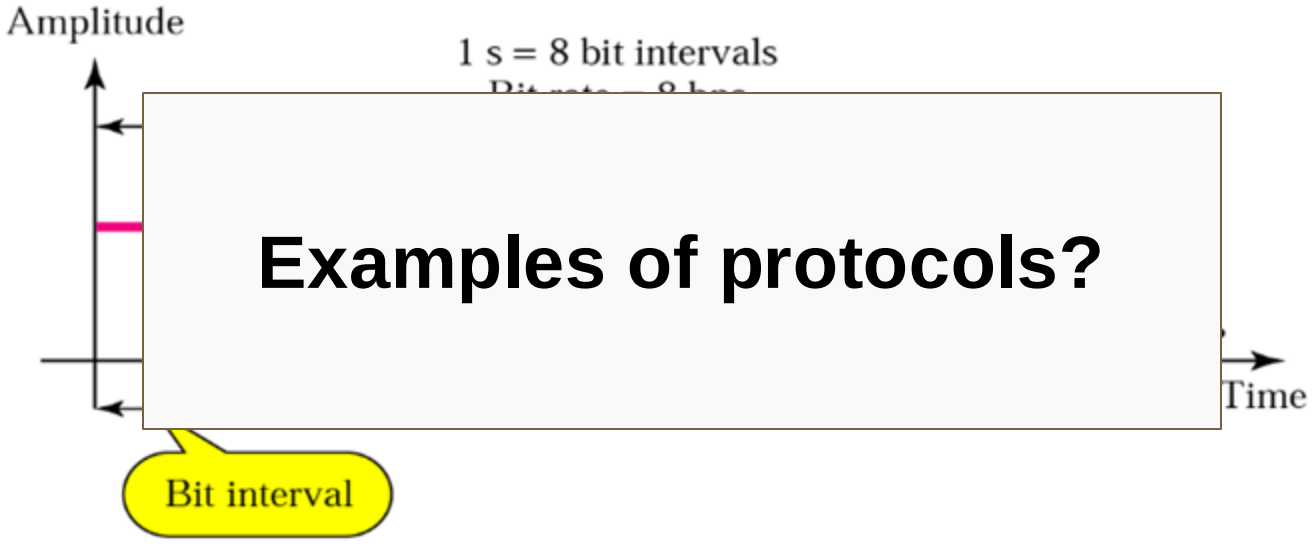
* bits per second(bps)

Protocol (Communication)

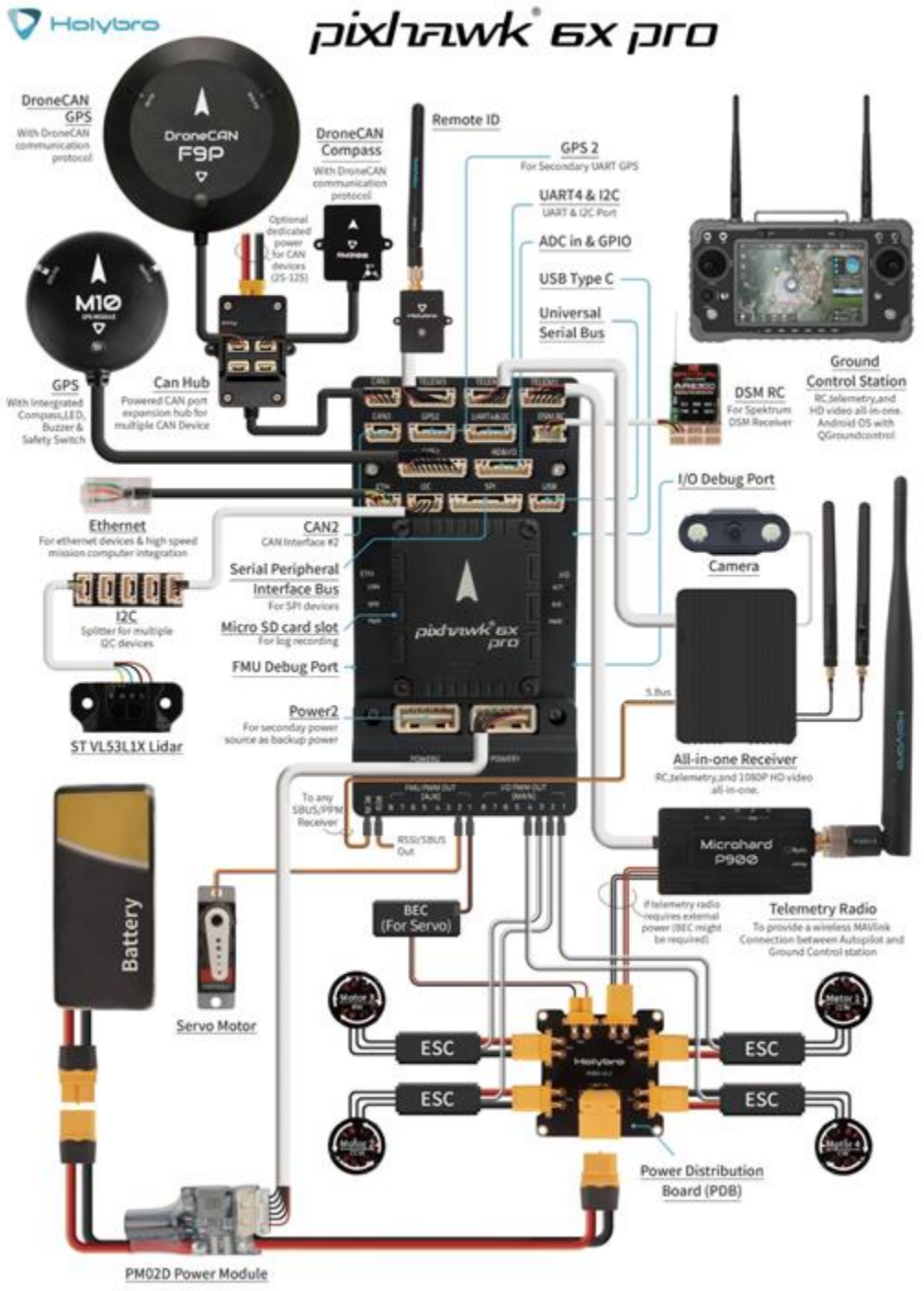
"...a set of standardized rules and procedures describing how to transmit or exchange data between (electronic) devices."

Data: …hard for the human to read…

Protocol: …defines how data is organized, structured, and represented during data transmission…



Examples of protocols?



Images: https://bmsce.ac.in/Content/CS/ur

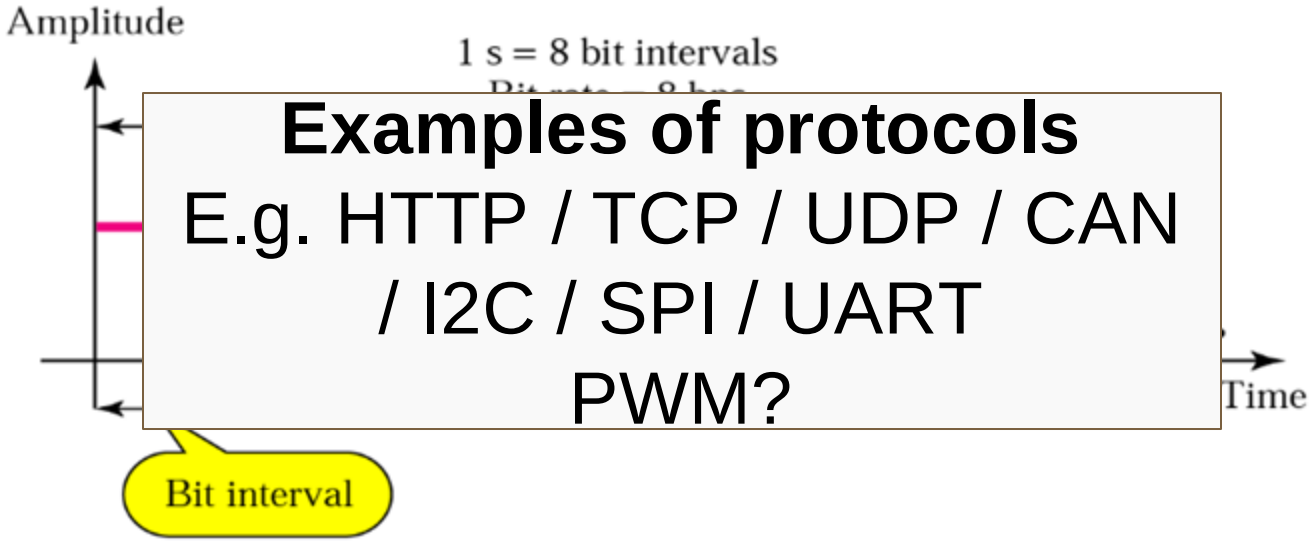
* bits per second(bps)

Protocol (Communication)

"...a set of standardized rules and procedures describing how to transmit or exchange data between (electronic) devices."

Data: …hard for the human to read…

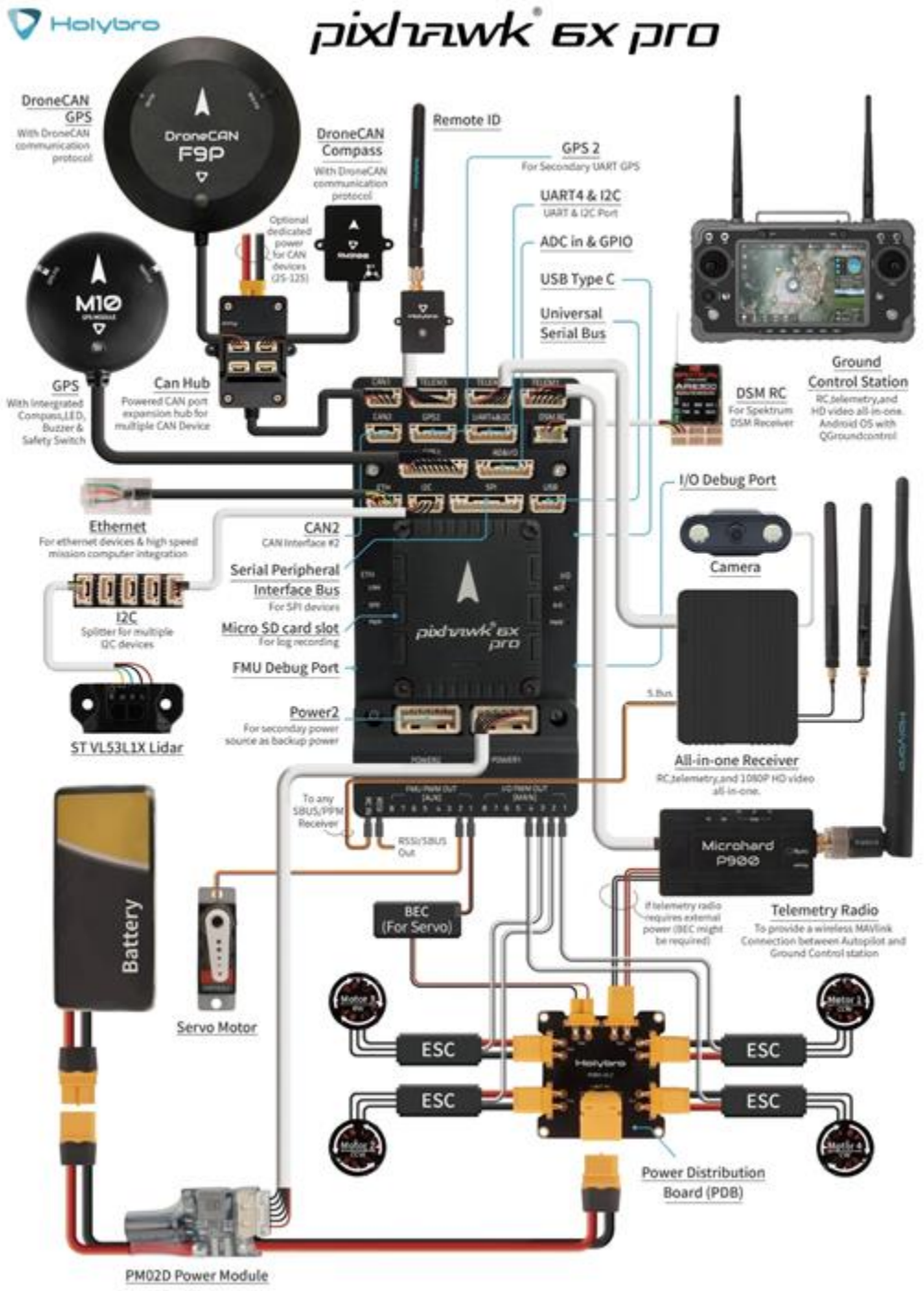
Protocol: …defines how data is organized, structured, and represented during data transmission…



Examples of protocols

E.g. HTTP / TCP / UDP / CAN / I2C / SPI / UART

PWM?



Images: https://bmsce.ac.in/Content/CS/ur

* bits per second(bps)

Digital sensors (Module 6)

"A type of sensor that directly outputs discrete digital data, typically in the form of binary signals (0s and 1s), representing the physical quantity it measures."

- …when data has been converted,
  - …it is then sent to a microcontroller over a digital communication protocol, such as

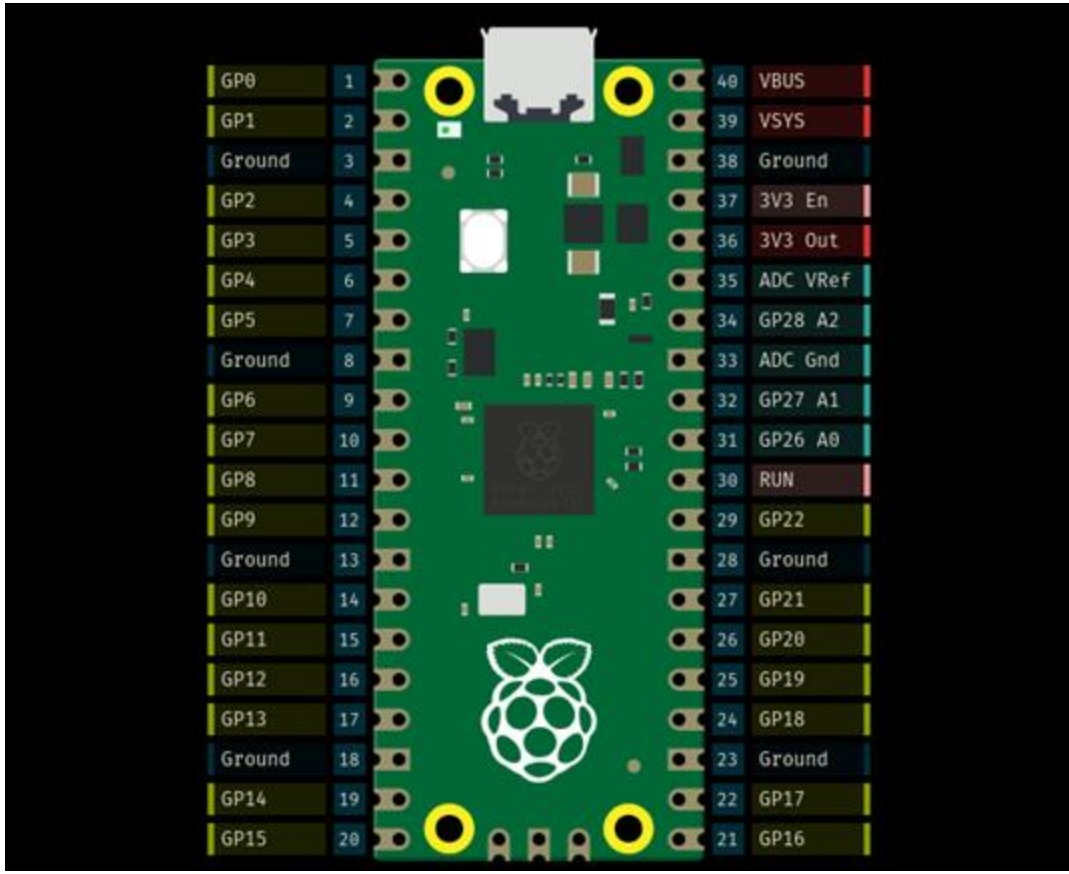


Illustration: https://pico.pinout.xyz/

Digital sensors (Module 6)

"A type of sensor that directly outputs discrete digital data, typically in the form of binary signals (0s and 1s), representing the physical quantity it measures."

- …when data has been converted,
  - …it is then sent to a microcontroller over a digital communication protocol, such as
    - I2C,

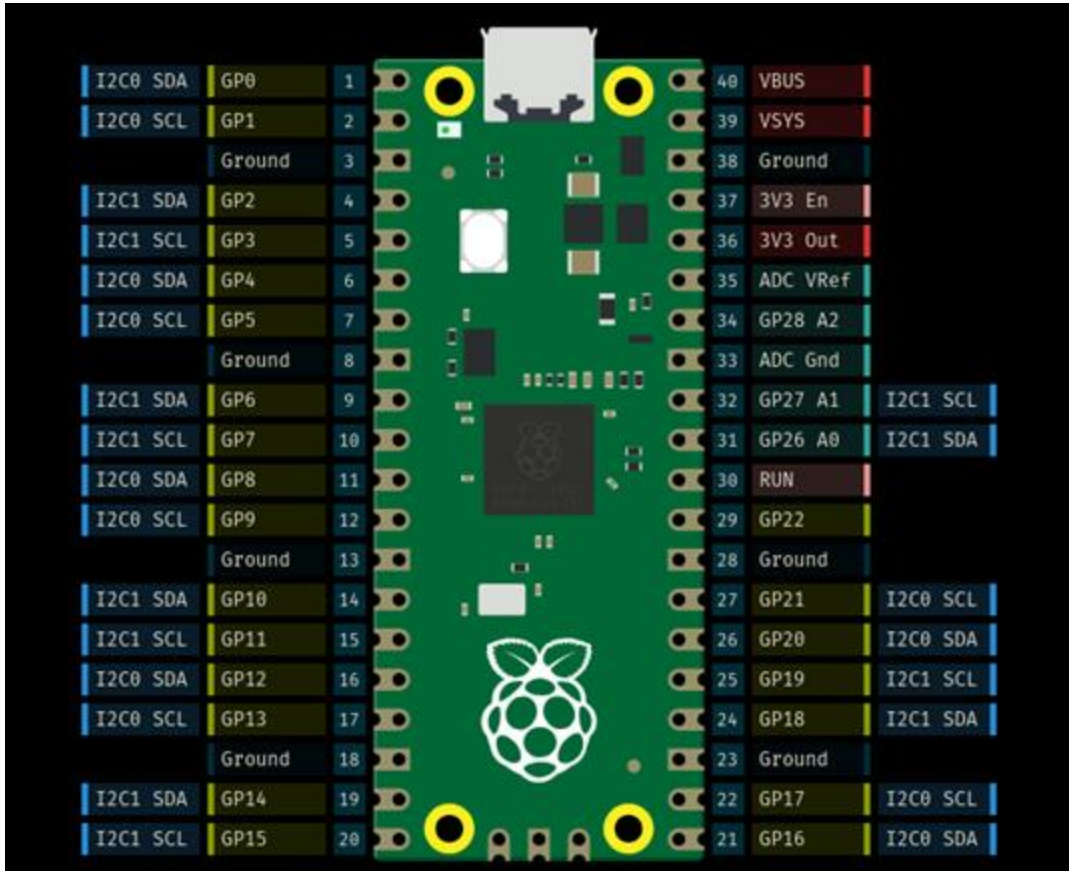


Illustration: https://pico.pinout.xyz/

Digital sensors (Module 6)

"A type of sensor that directly outputs discrete digital data, typically in the form of binary signals (0s and 1s), representing the physical quantity it measures."

- …when data has been converted,
  - …it is then sent to a microcontroller over a digital communication protocol, such as
    - I2C,
    - SPI, or

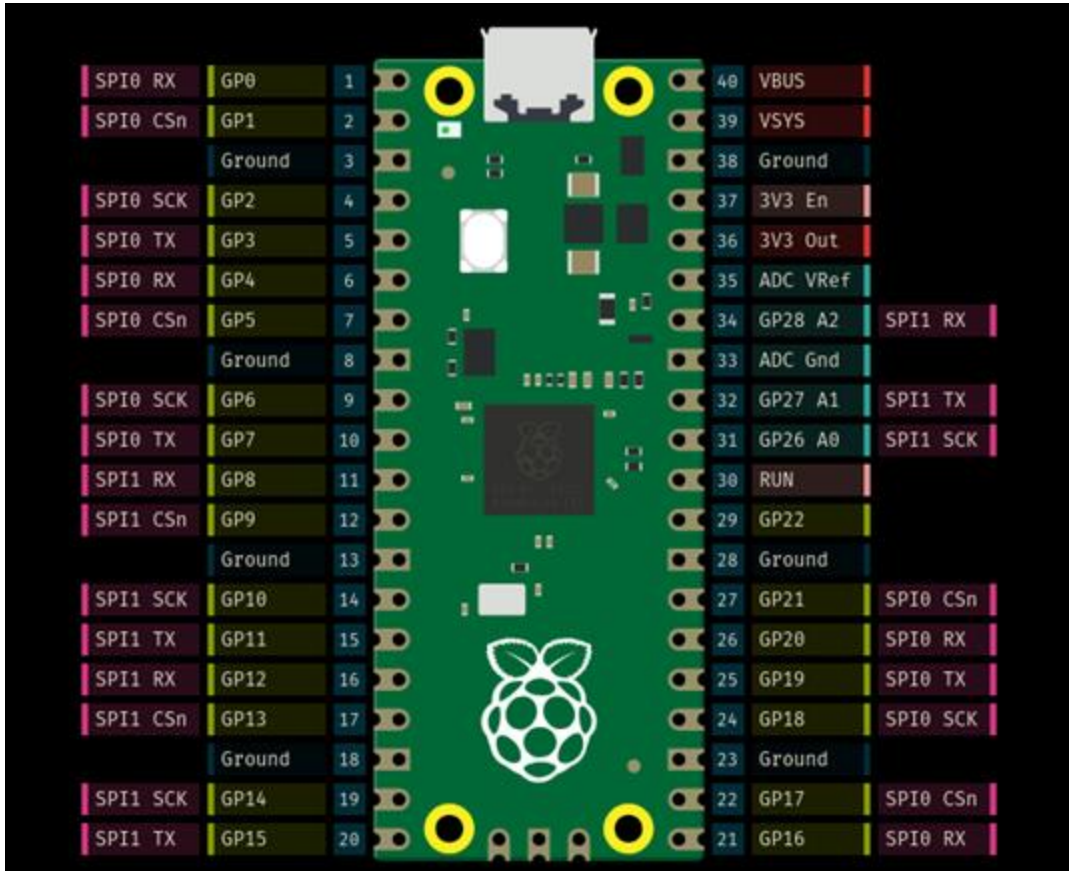


Illustration: https://pico.pinout.xyz/

Digital sensors (Module 6)

"A type of sensor that directly outputs discrete digital data, typically in the form of binary signals (0s and 1s), representing the physical quantity it measures."

- …when data has been converted,
  - …it is then sent to a microcontroller over a digital communication protocol, such as
    - I2C,
    - SPI, or
    - UART

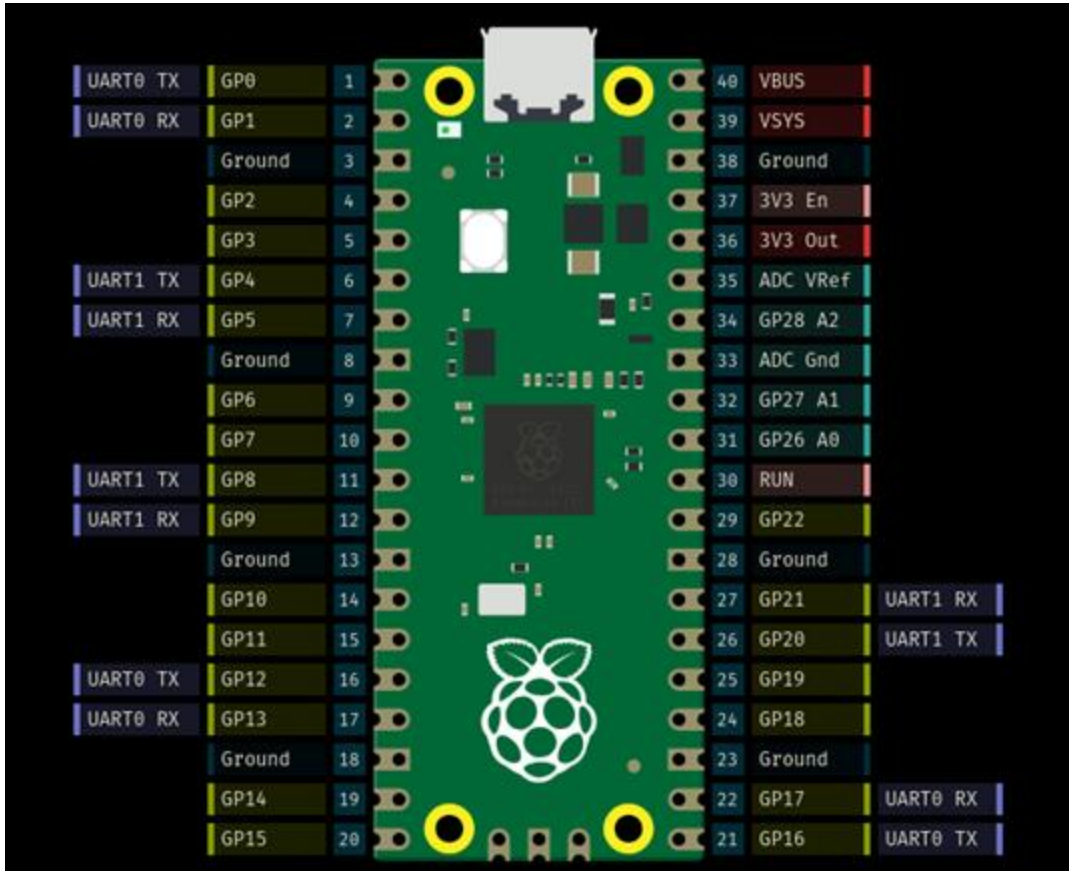


Illustration: https://pico.pinout.xyz/

Key modules, classes and functions (Module 3)

- **`machine` module:** The module for interfacing with the hardware of a microcontroller.

Classes (`machine` module)
• `class Pin` – control I/O pins • `class Signal` – control and sense external I/O devices • `class ADC` – analog to digital conversion • `class ADCBlock` – control ADC peripherals • `class PWM` – pulse width modulation • `class UART` – duplex serial communication bus • `class SPI` – a Serial Peripheral Interface bus protocol (controller side) • `class I2C` – a two-wire serial protocol • `class I2S` – Inter-IC Sound bus protocol • `class RTC` – real time clock • `class Timer` – control hardware timers • `class WDT` – watchdog timer • `class SD` – secure digital memory card (cc3200 port only) • `class SDCard` – secure digital memory card • `class USBDevice` – USB Device driver

Illustration: https://i.redd.it/76oyebb1i1a61.png

Key modules, classes and functions (Module 3)

- **`machine` module:** The module for interfacing with the hardware of a microcontroller.

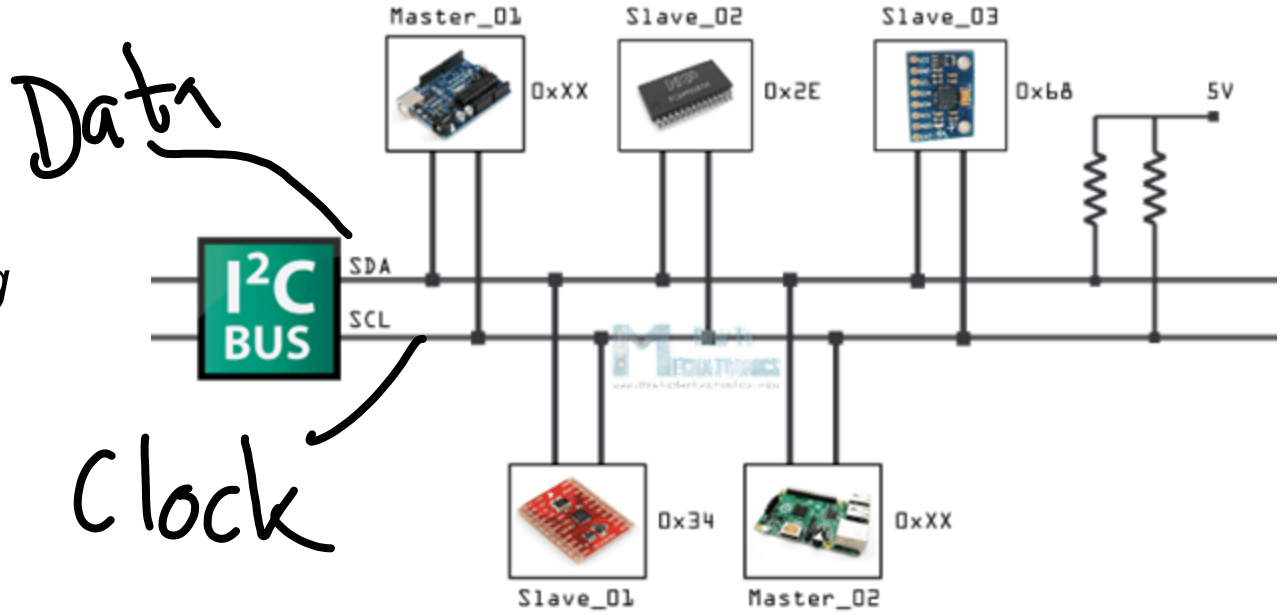
Classes (`machine` module)
`class Pin` – control I/O pins
`class Signal` – control and sense external I/O devices
`class ADC` – analog to digital conversion
`class ADCBlock` – control ADC peripherals
`class PWM` – pulse width modulation
`class UART` – duplex serial communication bus
`class SPI` – a Serial Peripheral Interface bus protocol (controller side)
`class I2C` – a two-wire serial protocol
`class I2S` – Inter-IC Sound bus protocol
`class RTC` – real time clock
`class Timer` – control hardware timers
`class WDT` – watchdog timer
`class SD` – secure digital memory card (cc3200 port only)
`class SDCard` – secure digital memory card
`class USBDevice` – USB Device driver

Illustration: https://i.redd.it/76oyebb1i1a61.png

I²C (Inter-Integrated Circuit)

I²C (Inter-Integrated Circuit)

"...a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL)"



Overview of I²C

- …a popular communication protocol for short distance communication between multiple low-speed devices.
  - called I 'two' C (typically) or I 'squared' C
  - **widely used in robotics** for interconnecting various sensors, actuators, and other components / ICs.
  - Developed in 1982 by Philips Semiconductor (now NXP Semiconductor)

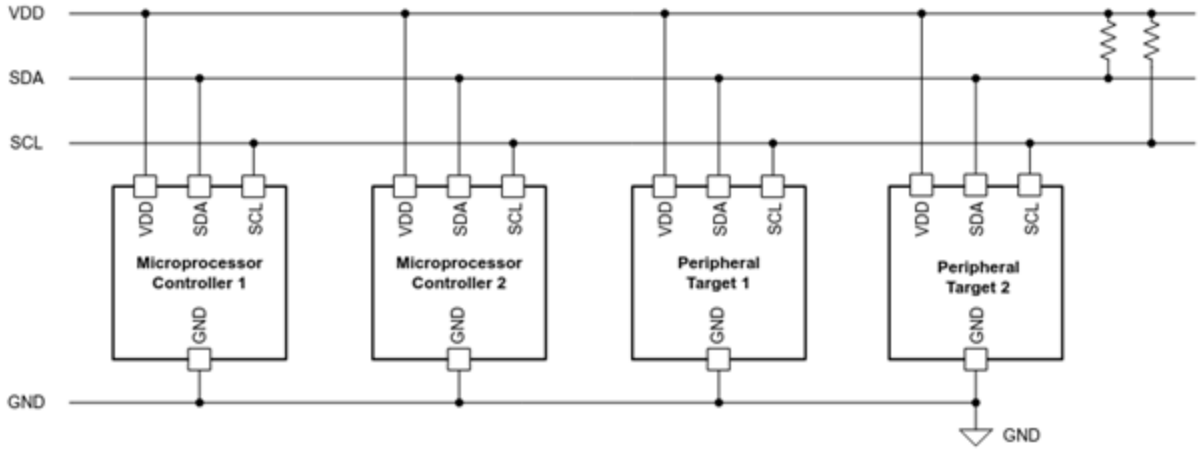


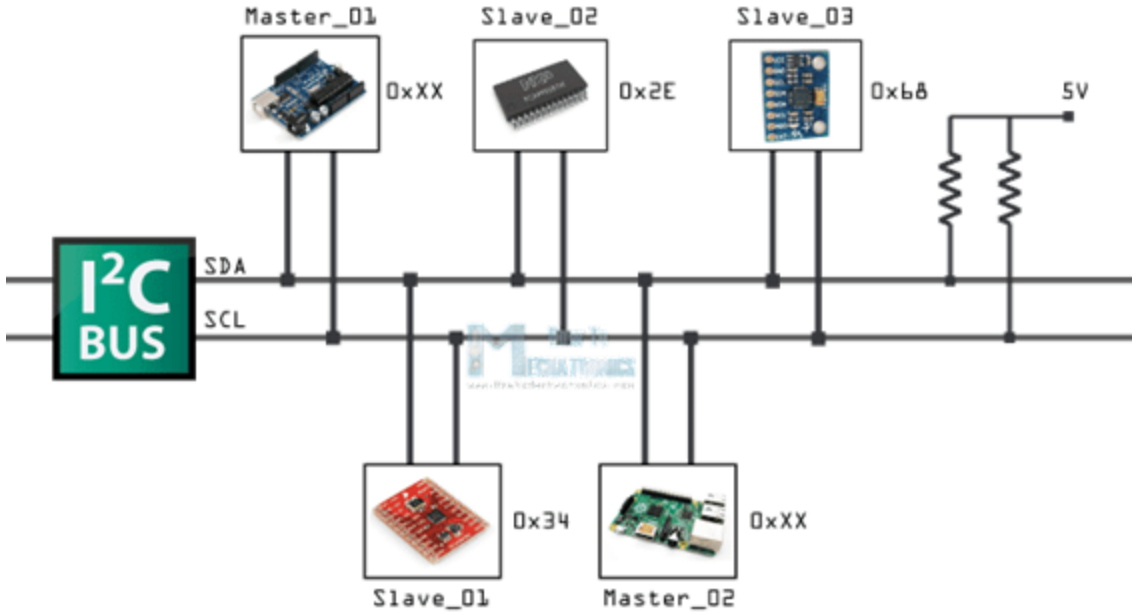
Illustration: https://ai.thestempedia.com/docs/evive/evive-technical-specifications/arduino-core-interface/i2c-communication/ and https://www.ti.com/lit/an/sbaa565/sbaa565.pdf?ts=1727931531869

I²C (Inter-Integrated Circuit)

"...a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL)"

Overview of I²C

- **Two-Wire Interface:**
  - **SDA (Serial Data) line**
    - …carries the data
  - **SCL (Serial Clock) line**
    - …carries the clock signal
    - Synchronizes data transmission.



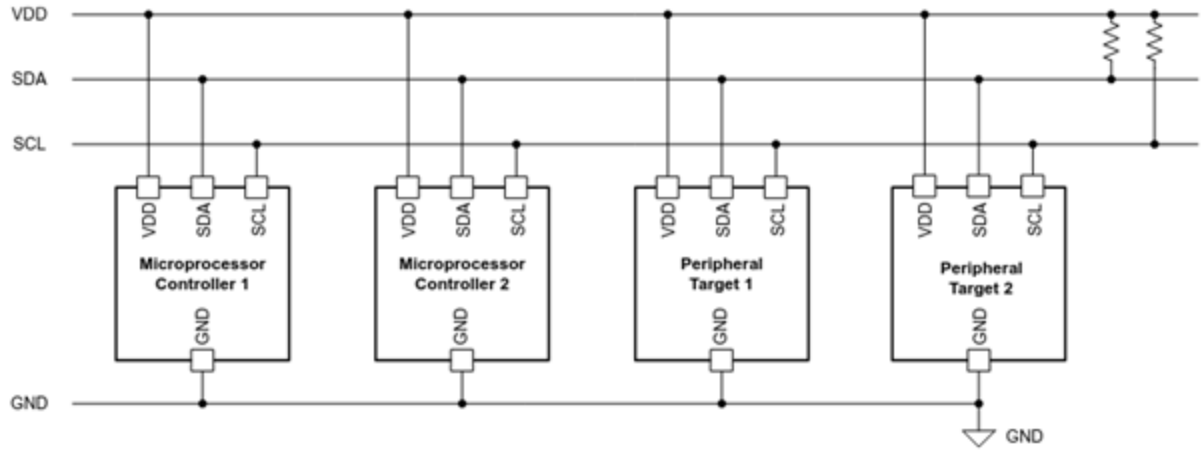


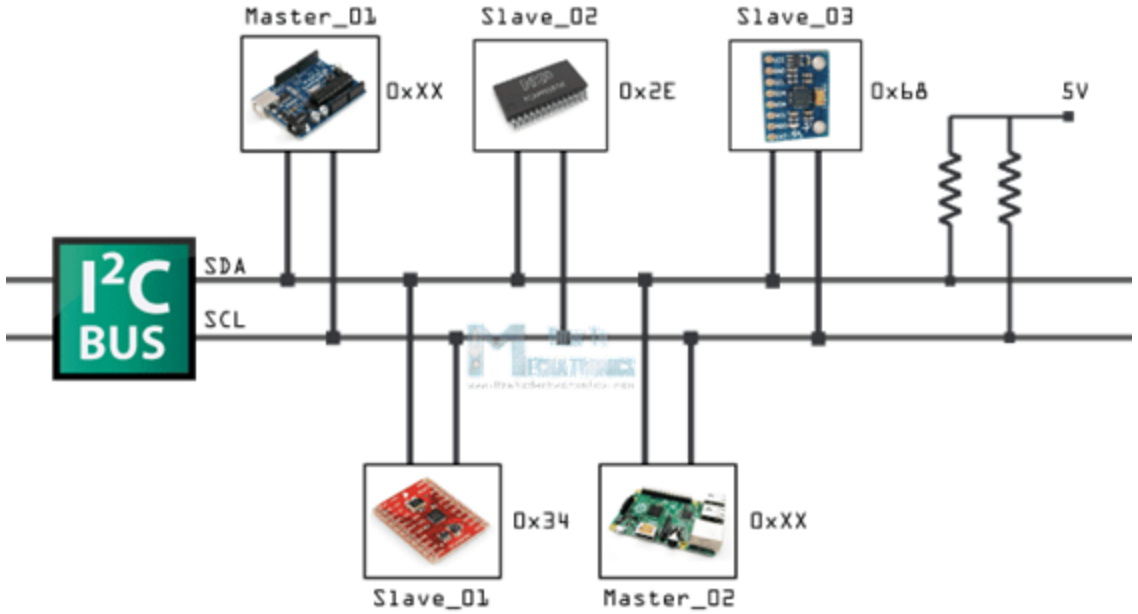
Illustration: https://ai.thestempedia.com/docs/evive/evive-technical-specifications/arduino-core-interface/i2c-communication/ and https://www.ti.com/lit/an/sbaa565/sbaa565.pdf?ts=1727931531869

I²C (Inter-Integrated Circuit)

"...a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL)"

Overview of I²C

- **Master / Slave Architecture**
  - The **master** controls communication…
    - …initiating data transfers with the **slaves**.
  - New Terminology?
    - Primary/Secondary?
    - Parent/Child?
- **Synchronous protocol** (shared clock line)



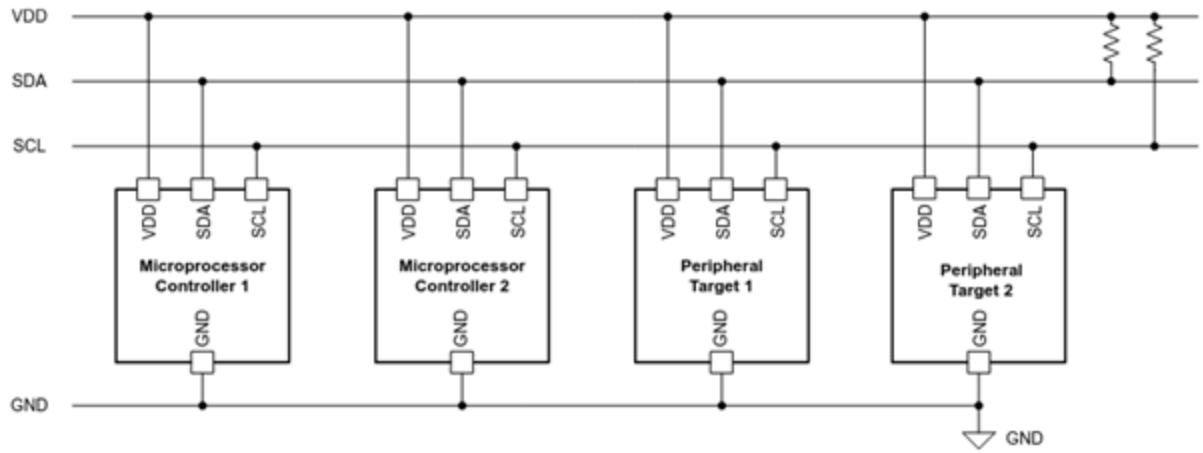
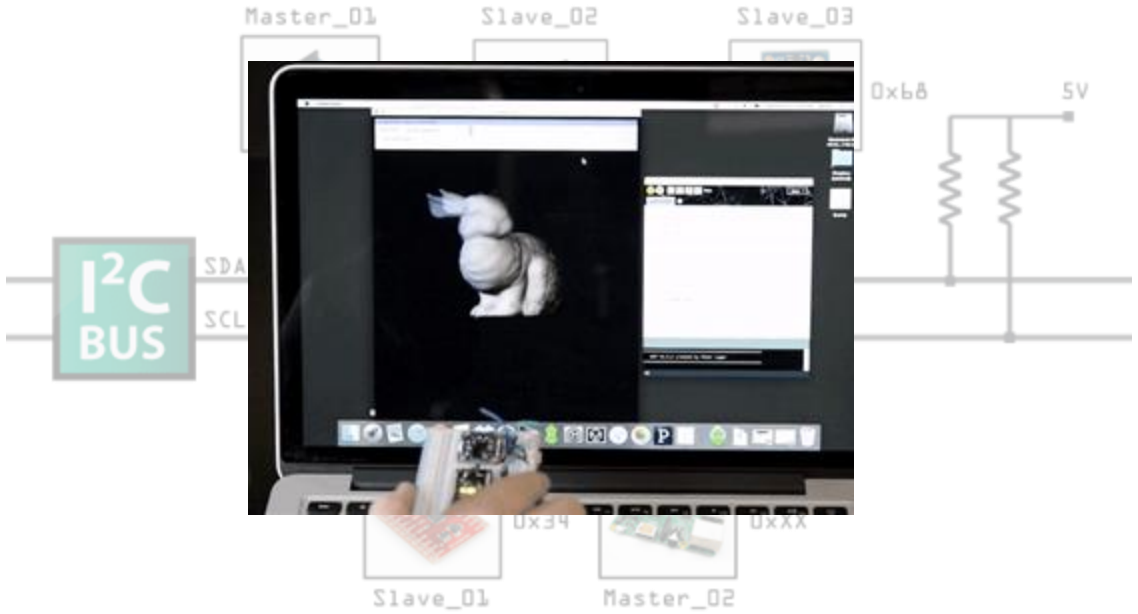


Illustration: https://ai.thestempedia.com/docs/evive/evive-technical-specifications/arduino-core-interface/i2c-communication/ and https://www.ti.com/lit/an/sbaa565/sbaa565.pdf?ts=1727931531869

I²C (Inter-Integrated Circuit)

"...a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL)"



Overview of I²C

- **Multi-Device Communication:**
  - Supports up to 128 devices on one bus
  - Ideal for integrating multiple sensors and components.
- **Addressing System:**
  - Each device on the I²C bus has a unique 7-bit address (typically). Examples:
    - ***OLED*** (SSD1306): 0x3c (60)
    - ***IMU*** (BNO085): 0x4A (74)
    - ***ToF*** (VL53L1X): 0x29 (41)

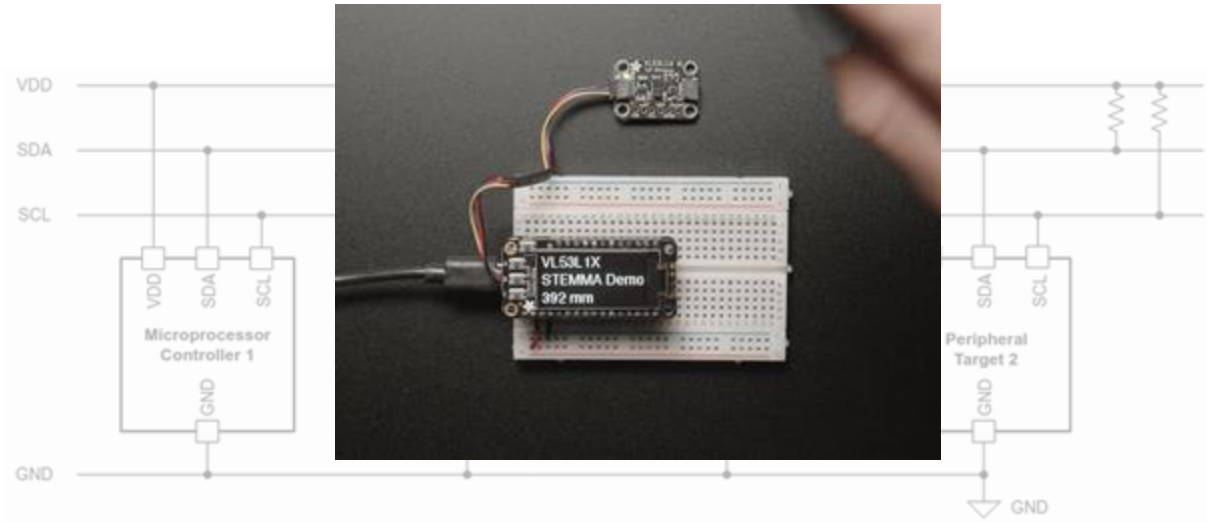
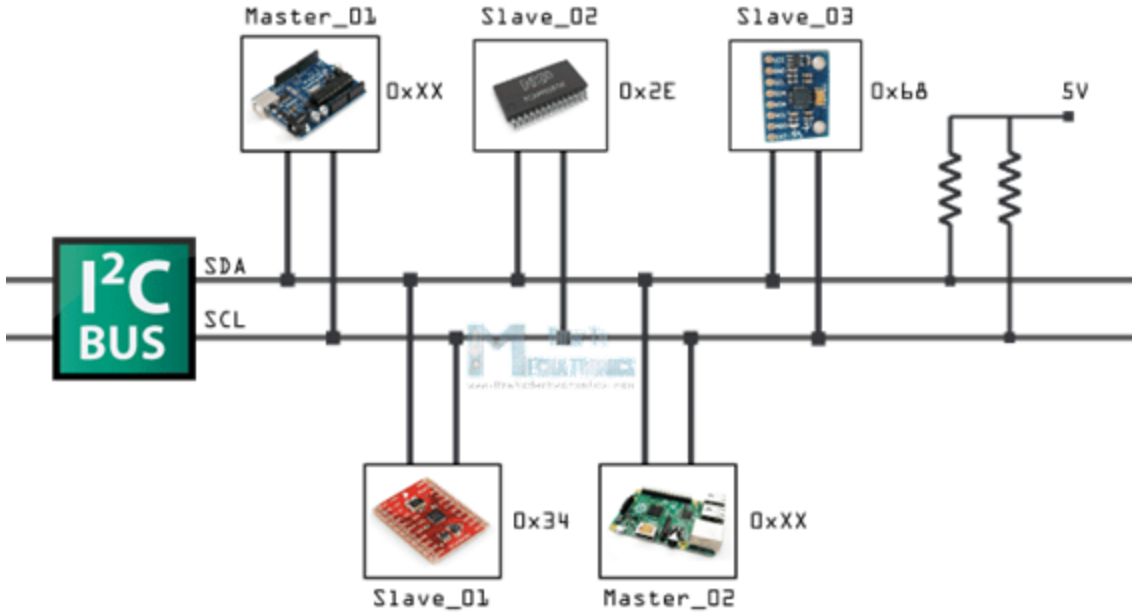


Illustration: https://ai.thestempedia.com/docs/evive/evive-technical-specifications/arduino-core-interface/i2c-communication/ and https://www.ti.com/lit/an/sbaa565/sbaa565.pdf?ts=1727931531869

I²C (Inter-Integrated Circuit)

"...a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL)"



Overview of I²C

- **Speed Modes:** Offers several speed modes
  - Starting with Standard-mode (100 kilobits per second (kbps))
  - …
  - Up to Ultra-Fast mode (5 Megabits per second (Mbps))

Table 1-1. Maximum Transmission Rates for Different I²C Modes

I²C Mode	Maximum Bit Rate
Standard-mode	100kbps
Fast-mode	400kbps
Fast-mode Plus	1Mbps
High-speed mode	3.4Mbps
Ultra-Fast mode	5Mbps

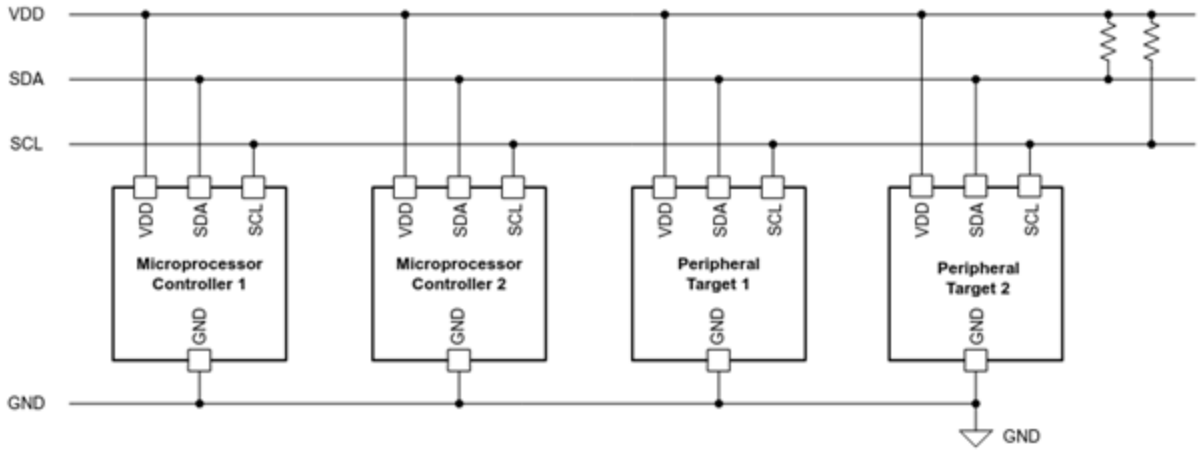


Illustration: https://ai.thestempedia.com/docs/evive/evive-technical-specifications/arduino-core-interface/i2c-communication/ and https://www.ti.com/lit/an/sbaa565/sbaa565.pdf?ts=1727931531869

I²C (Inter-Integrated Circuit)

"...a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL)"

I²C frames

- **Start condition:** The master claims the bus.
  - Pull first **SDA LOW**,
  - …then **SCL LOW**
  - **Starts sending the clock signal (SCL)**
  - (Any node in on the bus can claim it (being the master))

Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

I²C (Inter-Integrated Circuit)

"...a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL)"

I²C frames

- **Start condition:** The master claims the bus (Pull first SDA, then SCL LOW)
- **Slave address:** …who to communicate with? (7 bits)

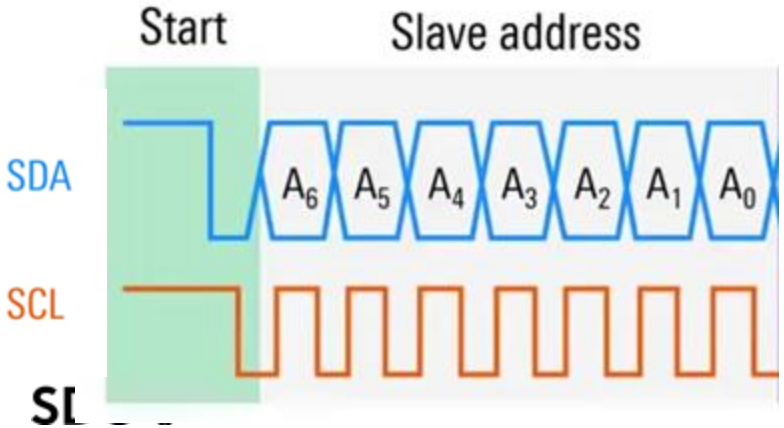


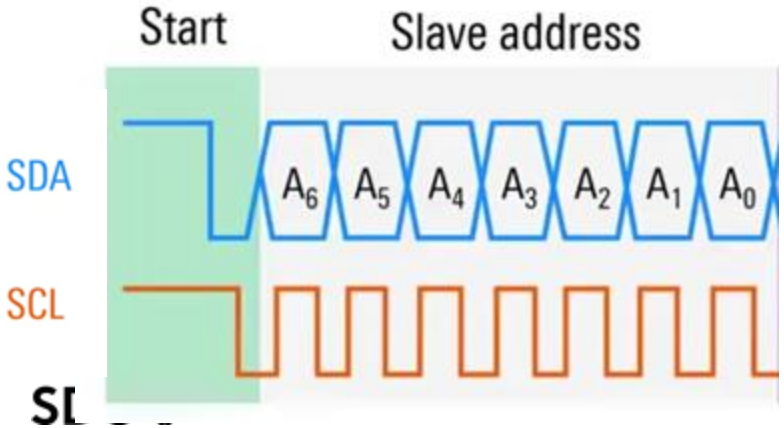
Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

I²C (Inter-Integrated Circuit)

"...a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL)"

I²C frames

- **Start condition:** The master claims the bus (Pull first SDA, then SCL LOW)
- **Slave address:** …who to communicate with? (7 bits)
  - **Timing** between data (SDA line) and the clock (SCL line)



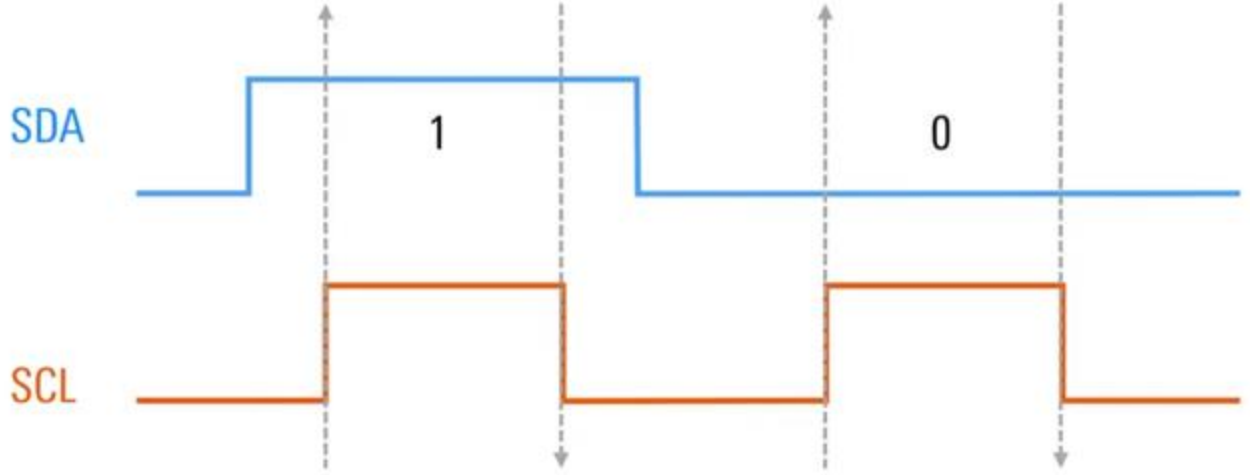


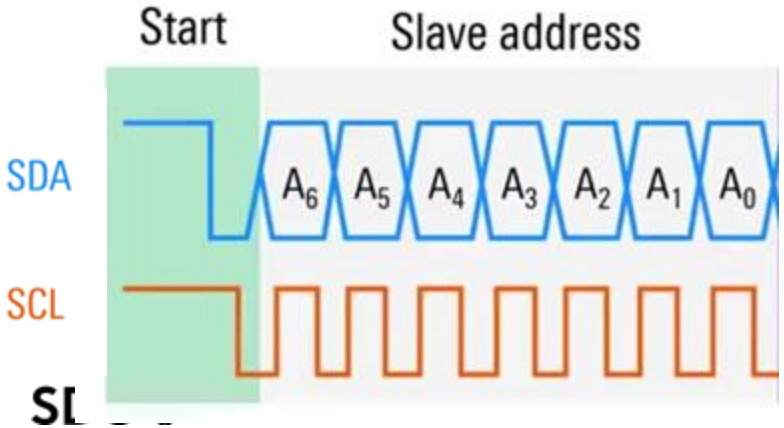
Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

I²C (Inter-Integrated Circuit)

"...a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL)"

I²C frames

- **Start condition:** The master claims the bus (Pull first SDA, then SCL LOW)
- **Slave address:** …who to communicate with? (7 bits)
  - **Timing** between data (SDA line) and the clock (SCL line)
    - SDA doesn't change between the RISING and FALLING edges of the SCL



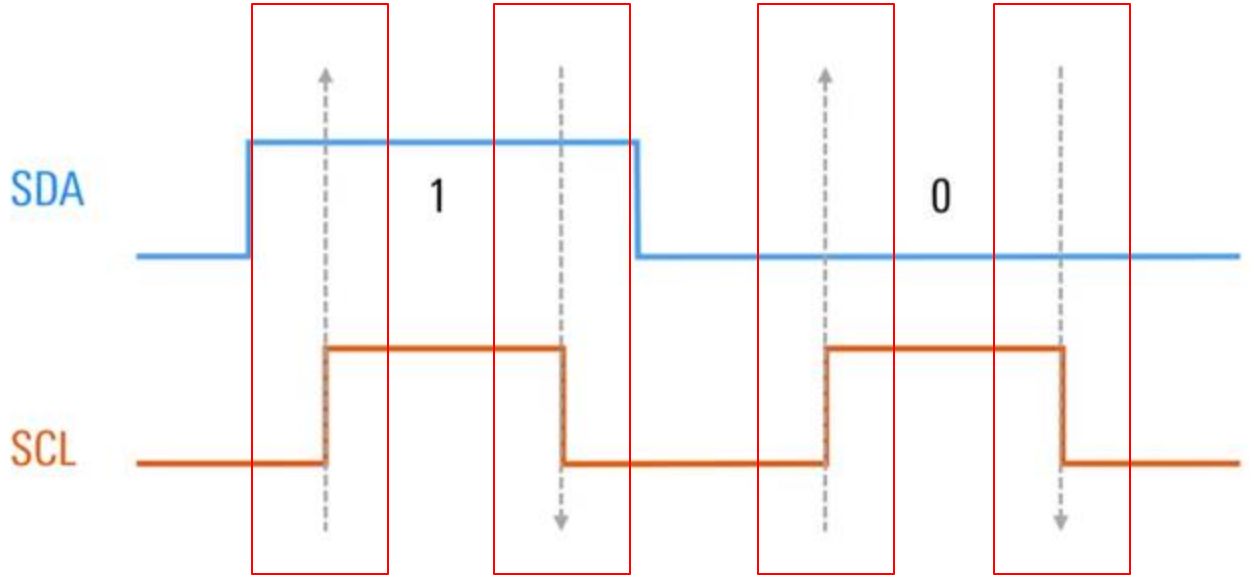


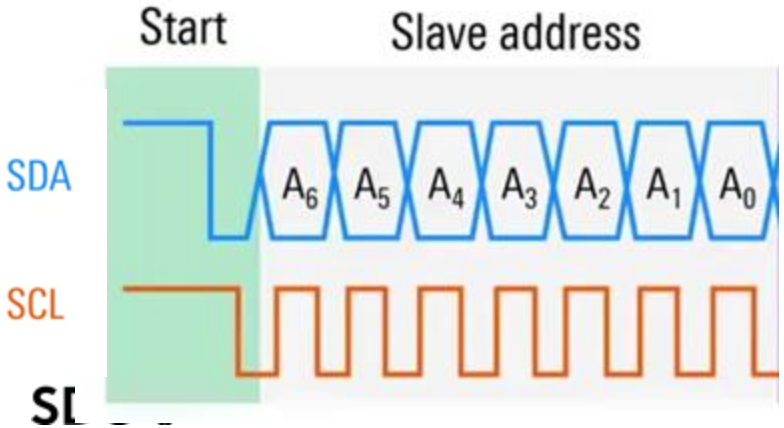
Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

I²C (Inter-Integrated Circuit)

"...a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL)"

I²C frames

- **Start condition:** The master claims the bus (Pull first SDA, then SCL LOW)
- **Slave address:** …who to communicate with? (7 bits)
  - **Timing** between data (SDA line) and the clock (SCL line)
    - SDA doesn't change between the RISING and FALLING edges of the SCL
    - **Data is only read when the SCL is HIGH**



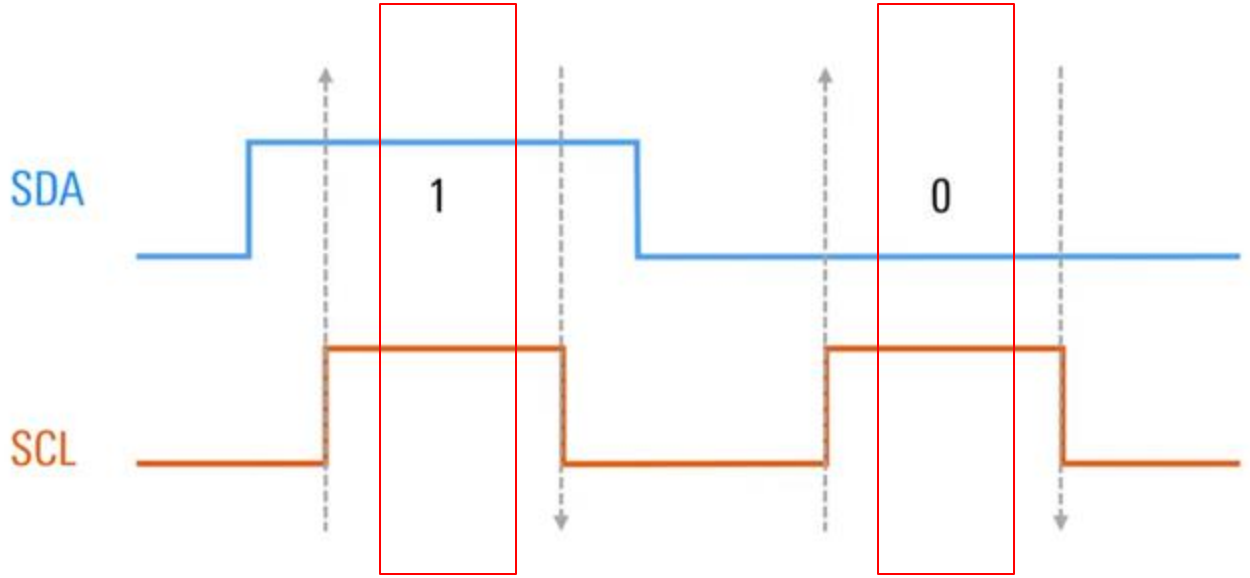


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

I²C (Inter-Integrated Circuit)

"...a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL)"

I²C frames

- **Start condition:** The master claims the bus (Pull first SDA, then SCL LOW)
- **Slave address:** …who to communicate with? (7-bits)
- **R/W:** Indication of if the master wish to read from the slave or write to it. (1-bit)
  - **LOW**: Master wants to write data to the slave
  - **HIGH**: Master wants to read data from the slave

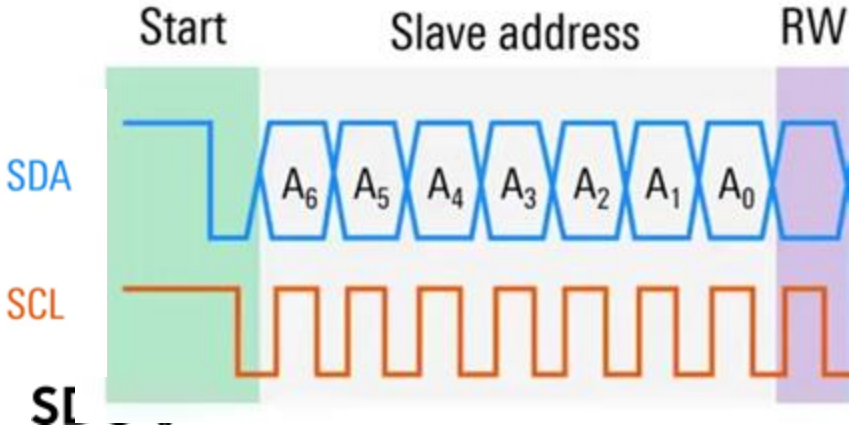


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

I²C (Inter-Integrated Circuit)

"...a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL)"

I²C frames

- **Start condition:** The master claims the bus (Pull first SDA, then SCL LOW)
- **Slave address:** …who to communicate with? (7-bits)
- **R/W:** Indication of if the master wish to read from the slave or write to it. (1-bit)
- **Ack:** The slave acknowledges its presence and that it is ready. (1-bit)
  - **LOW**: Acknowledgement (ACK)
  - **HIGH**: Negative Acknowledgement (NACK)
    - Since it actively HIGH, not pulling it LOW must assume the slave is not responding / Lack of response

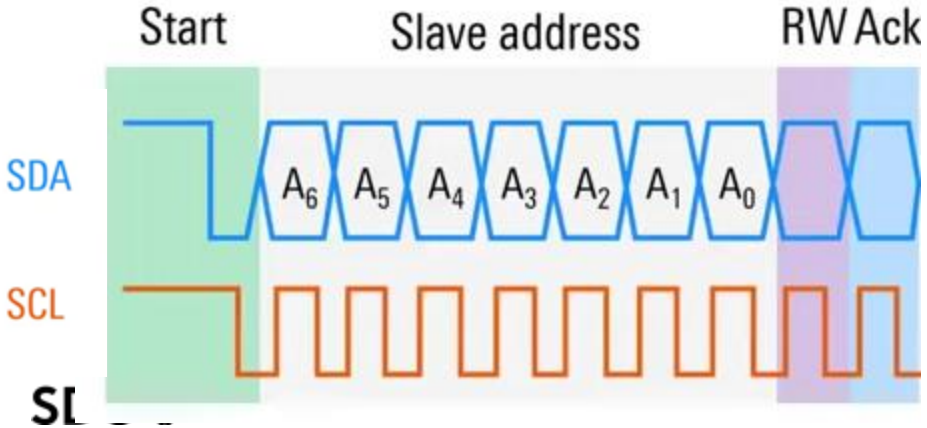


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

I²C (Inter-Integrated Circuit)

"...a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL)"

I²C frames

- **Start condition:** The master claims the bus (Pull first SDA, then SCL LOW)
- **Slave address:** …who to communicate with? (7 bits)
- **R/W:** Indication of if the master wish to read from the slave or write to it. (1 bit)
- **Ack:** The slave acknowledges its presence and that it is ready. (1 bit)
- **Data:** Actual data being transmitted. (8 bits / 1 byte)

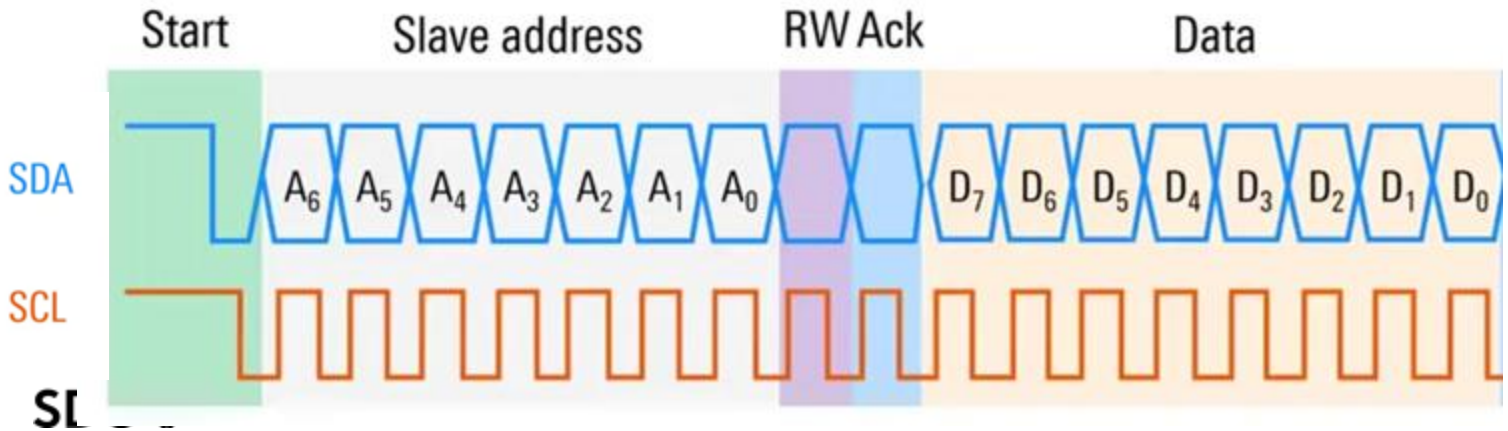


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

I²C (Inter-Integrated Circuit)

"...a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL)"

I²C frames

- **Start condition:** The master claims the bus (Pull first SDA, then SCL LOW)
- **Slave address:** …who to communicate with? (7-bits)
- **R/W:** Indication of if the master wish to read from the slave or write to it. (1-bit)
- **Ack:** The slave acknowledges its presence and that it is ready. (1-bit)
- **Data:** Actual data being transmitted. (8-bits)
- **Ack:** …and acknowledges when data has been transferred and received correctly. (1 bit)

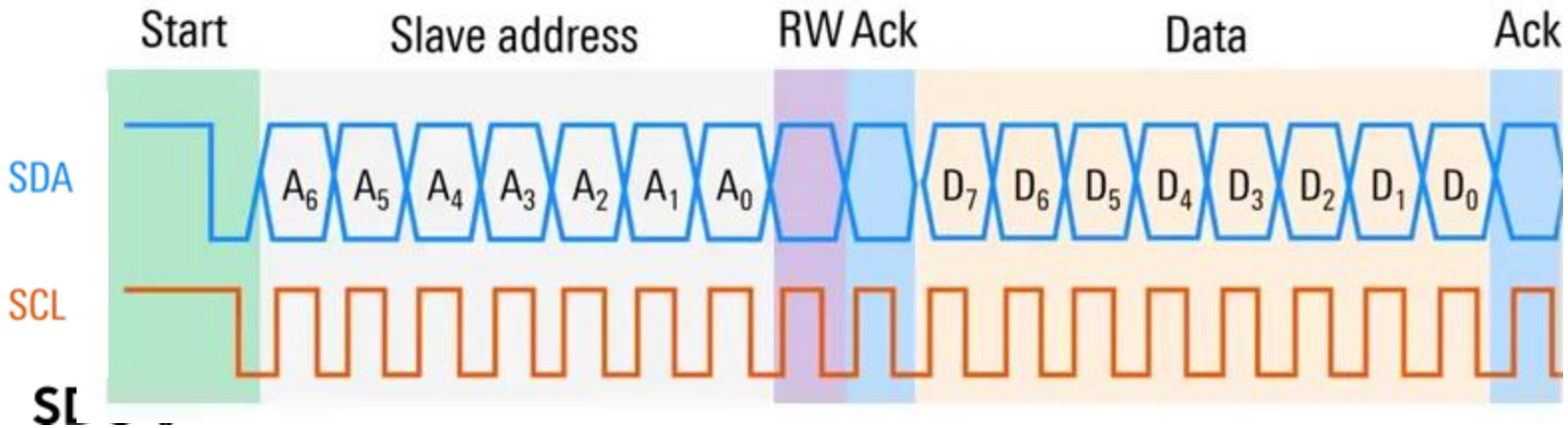


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

I²C (Inter-Integrated Circuit)

"...a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL)"

I²C frames

- **Start condition:** The master claims the bus (Pull first SDA, then SCL LOW)
- **Slave address:** …who to communicate with? (7-bits)
- **R/W:** Indication of if the master wish to read from the slave or write to it. (1-bit)
- **Ack:** The slave acknowledges its presence and that it is ready. (1-bit)
- **Data:** Actual data being transmitted. (8-bits)
- **Ack:** …and acknowledges when data has been transferred and received correctly. (1 bit)
  - …and multiple data bytes can be transmitted, but individually acknowledged…

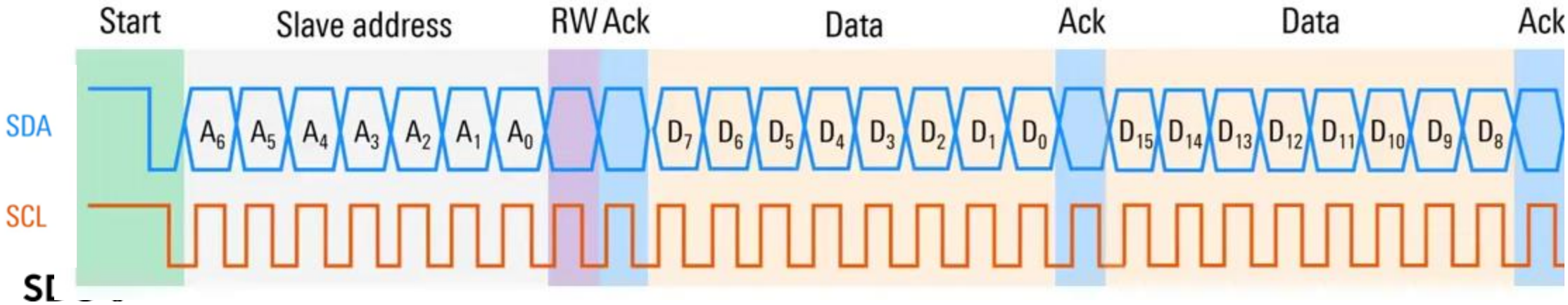


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

I²C (Inter-Integrated Circuit)

"...a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL)"

I²C frames

- **Start condition:** The master claims the bus.
- **Slave address:** …who to communicate with? (7-bits)
- **R/W:** Indication of if the master wish to read from the slave or write to it. (1-bit)
- **Ack:** The slave acknowledges its presence and that it is ready. (1-bit)
- **Data:** Actual data being transmitted. (8-bits)
- **Ack:** …and acknowledges when done. (1-bit)
- **Stop condition:** Communication is terminated.
  - Pull first **SCL HIGH** …then **SDA HIGH**

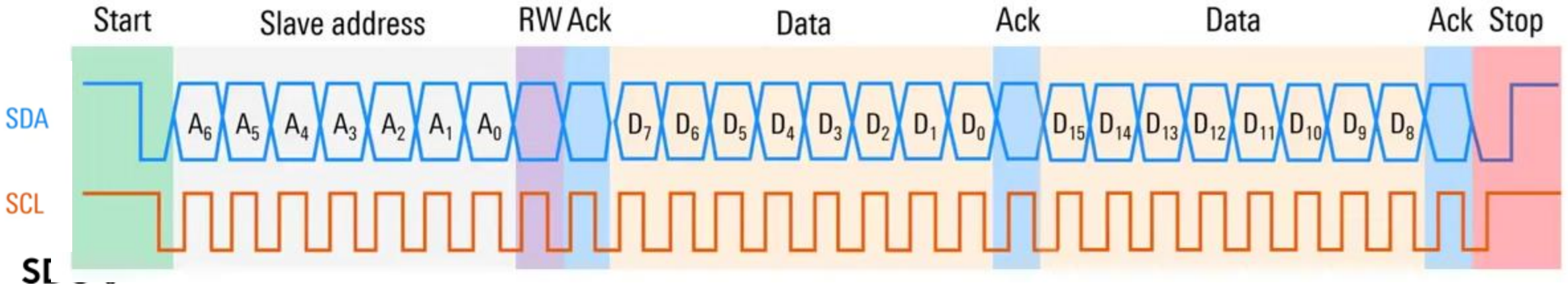


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

Python standard libraries and micro-libraries

- I2C **Class:** a two-wire serial protocol

- **Constructor**
  - `class machine.I2C(id, *, scl, sda, freq=400000, timeout=50000)`

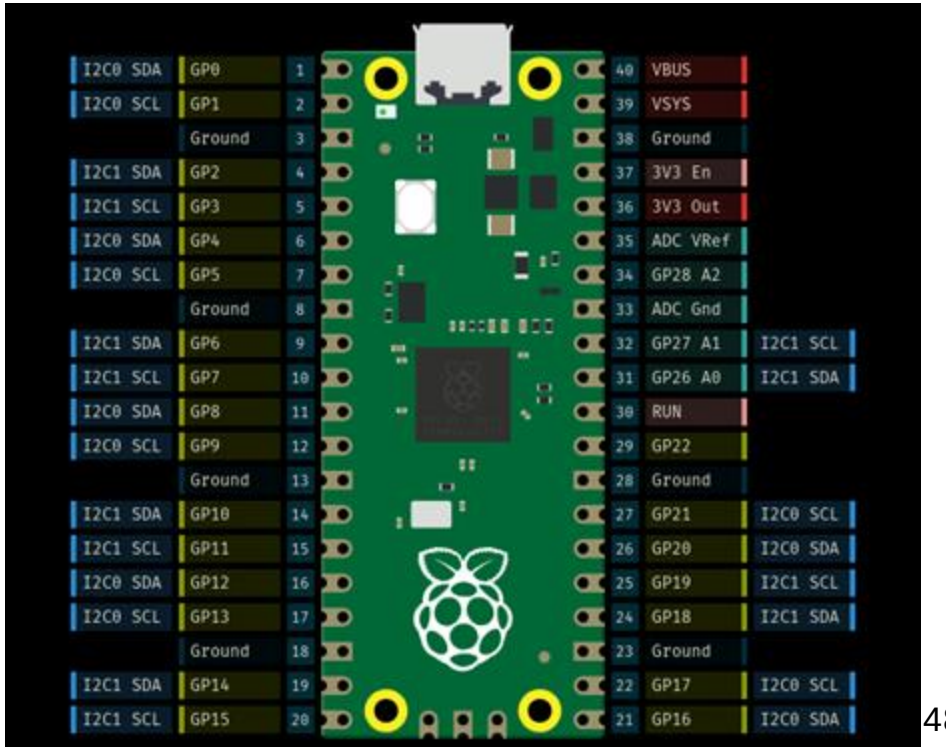
  - Parameters
    - `id`: Specifies the I2C bus to use. On some devices
      - Only `I2C(0)` and `I2C(1)` may be available.
    - `scl`: Pin object for the clock line (SCL).
    - `sda`: Pin object for the data line (SDA).
    - `freq`: (Optional) Frequency of the I2C bus. Default is 400kHz, which is a typical I2C frequency.

- **Methods**:
  - `init() / deinit()`: Initialize or re-initialize / de-initialize the I2C bus.
  - `scan()`: Returns a list of addresses of I2C devices connected to the bus.
  - `readfrom(addr, nbytes, stop=True)`: Reads a specified number of bytes from the device at the given address.
  - `writeto(addr, buf, stop=True)`: Writes bytes from a buffer to the device at the specified address.

  - …and many more methods:
    https://docs.micropython.org/en/latest/library/machine.I2C.html

Classes (`machine` module)

- `class Pin` – control I/O pins
- `...`
- `class PWM` – pulse width modulation
- `class UART` – duplex serial communication bus
- `class SPI` – a Serial Peripheral Interface bus protocol (controller side)
- `class I2C` – a two-wire serial protocol
- `class I2S` – Inter-IC Sound bus protocol
- `...`
- `class USBDevice` – USB Device driver



Python standard libraries and micro-libraries

- I2C **Class:** a two-wire serial protocol

- **Constructor**
  - `class machine.I2C(id, *, scl, sda, freq=400000, timeout=50000)`

  - Parameters
    - `id`: Specifies the I2C bus to use. On some devices
      - Only `I2C(0)` and `I2C(1)` may be available.
    - `scl`: Pin object for the clock line (SCL).
    - `sda`: Pin object for the data line (SDA).
    - `freq`: (Optional) Frequency of the I2C bus. Default is 400kHz, which is a typical I2C frequency.

- **Methods**:
  - `init() / deinit()`: Initialize or re-initialize / de-initialize the I2C bus.
  - `scan()`: Returns a list of addresses of I2C devices connected to the bus.
  - `readfrom(addr, nbytes, stop=True)`: Reads a specified number of bytes from the device at the given address.
  - `writeto(addr, buf, stop=True)`: Writes bytes from a buffer to the device at the specified address.

  - …and many more methods: **https://docs.micropython.org/en/latest/library/machine.I2C.html**

Example: Transmit and Receive data via I2C

```
from machine import I2C, Pin
import time

# Initialize I2C on the Pico with SDA on GP8 and SCL on GP9
i2c = I2C(0, scl=Pin(9), sda=Pin(8))

# Scan for I2C devices on the bus
devices = i2c.scan()

if devices:
    print("I2C devices found:", devices)
else:
    print("No I2C devices found")

# Define the address of the device you want to communicate with
device_address = 0x68

# Check if the device is found before attempting to communicate
if device_address in devices:
    print(f"Communicating with device at address: {hex(device_address)}")

    # Example: Write data to the device
    i2c.writeto(device_address, b'\x01')  # Sending byte 0x01 to the device

    # Pause before reading data
    time.sleep(0.1)

    # Example: Read 6 bytes from the device
    data = i2c.readfrom(device_address, 6)
    print("Received Data:", data)

else:
    print(f"Device at address {hex(device_address)} not found.")
```

Python standard libraries and micro-libraries

- I2C **Class:** a two-wire serial protocol

- **Constructor**
  - `class machine.I2C(id, *, scl, sda, freq=400000, timeout=50000)`

  - Parameters
    - `id`: Specifies the I2C bus to use. On some devices
      - Only `I2C(0)` and `I2C(1)` may be available.
    - `scl`: Pin object for the clock line (SCL).
    - `sda`: Pin object for the data line (SDA).
    - `freq`: (Optional) Frequency of the I2C bus. Default is 400kHz, which is a typical I2C frequency.

- **Methods**:
  - `init() / deinit()`: Initialize or re-initialize / de-initialize the I2C bus.
  - `scan()`: Returns a list of addresses of I2C devices connected to the bus.
  - `readfrom(addr, nbytes, stop=True)`: Reads a specified number of bytes from the device at the given address.
  - `writeto(addr, buf, stop=True)`: Writes bytes from a buffer to the device at the specified address.

  - …and many more methods: **https://docs.micropython.org/en/latest/library/machine.I2C.html**

Example: Transmit and Receive data via I2C

```
from machine import I2C, Pin
import time

# Initialize I2C on the Pico with SDA on GP8 and SCL on GP9
i2c = I2C(0, scl=Pin(9), sda=Pin(8))

# Scan for I2C devices on the bus
devices = i2c.scan()

if devices:
    print("I2C devices found:", devices)
else:
    print("No I2C devices found")

# Define the address of the device you want to communicate with
device_address = 0x68

# Check if the device is found before attempting to communicate
if device_address in devices:
    print(f"Communicating with device at address: {hex(device_address)}")

    # Example: Write data to the device
    i2c.writeto(device_address, b'\x01')  # Sending byte 0x01 to the device

    # Pause before reading data
    time.sleep(0.1)

    # Example: Read 6 bytes from the device
    data = i2c.readfrom(device_address, 6)
    print("Received Data:", data)

else:
    print(f"Device at address {hex(device_address)} not found.")
```

Python standard libraries and micro-libraries

- I2C **Class:** a two-wire serial protocol

- **Constructor**
  - `class machine.I2C(id, *, scl, sda, freq=400000, timeout=50000)`

  - Parameters
    - `id`: Specifies the I2C bus to use. On some devices
      - Only `I2C(0)` and `I2C(1)` may be available.
    - `scl`: Pin object for the clock line (SCL).
    - `sda`: Pin object for the data line (SDA).
    - `freq`: (Optional) Frequency of the I2C bus. Default is 400kHz, which is a typical I2C frequency.

- **Methods**:
  - `init() / deinit()`: Initialize or re-initialize / de-initialize the I2C bus.
  - `scan()`: Returns a list of addresses of I2C devices connected to the bus.
  - `readfrom(addr, nbytes, stop=True)`: Reads a specified number of bytes from the device at the given address.
  - `writeto(addr, buf, stop=True)`: Writes bytes from a buffer to the device at the specified address.

  - …and many more methods: **https://docs.micropython.org/en/latest/library/machine.I2C.html**

Example: Transmit and Receive data via I2C

```
from machine import I2C, Pin
import time

# Initialise I2C on the Pico with SDA on GP8 and SCL on GP9
i2c = I2C(0, scl=Pin(9), sda=Pin(8))

# Scan for I2C devices on the bus
devices = i2c.scan()

if devices:
    print("I2C devices found:", devices)
else:
    print("No I2C devices found")

# Define the address of the device you want to communicate with
device_address = 0x68

# Check if the device is found before attempting to communicate
if device_address in devices:
    print(f"Communicating with device at address: {hex(device_address)}")

    # Example: Write data to the device
    i2c.writeto(device_address, b'\x01')  # Sending byte 0x01 to the device

    # Pause before reading data
    time.sleep(0.1)

    # Example: Read 6 bytes from the device
    data = i2c.readfrom(device_address, 6)
    print("Received Data:", data)

else:
    print(f"Device at address {hex(device_address)} not found.")
```

Python standard libraries and micro-libraries

- I2C **Class:** a two-wire serial protocol

- **Constructor**
  - `class machine.I2C(id, *, scl, sda, freq=400000, timeout=50000)`

  - Parameters
    - `id`: Specifies the I2C bus to use. On some devices
      - Only `I2C(0)` and `I2C(1)` may be available.
    - `scl`: Pin object for the clock line (SCL).
    - `sda`: Pin object for the data line (SDA).
    - `freq`: (Optional) Frequency of the I2C bus. Default is 400kHz, which is a typical I2C frequency.

- **Methods**:
  - `init() / deinit()`: Initialize or re-initialize / de-initialize the I2C bus.
  - `scan()`: Returns a list of addresses of I2C devices connected to the bus.
  - `readfrom(addr, nbytes, stop=True)`: Reads a specified number of bytes from the device at the given address.
  - `writeto(addr, buf, stop=True)`: Writes bytes from a buffer to the device at the specified address.

  - …and many more methods: **https://docs.micropython.org/en/latest/library/machine.I2C.html**

Example: Transmit and Receive data via I2C

```
from machine import I2C, Pin
import time

# Initialize I2C on the Pico with SDA on GP8 and SCL on GP9
i2c = I2C(0, scl=Pin(9), sda=Pin(8))

# Scan for I2C devices on the bus
devices = i2c.scan()

if devices:
    print("I2C devices found:", devices)
else:
    print("No I2C devices found")

# Define the address of the device you want to communicate with
device_address = 0x68

# Check if the device is found before attempting to communicate
if device_address in devices:
    print(f"Communicating with device at address: {hex(device_address)}")

    # Example: Write data to the device
    i2c.writeto(device_address, b'\x01')  # Sending byte 0x01 to the device

    # Pause before reading data
    time.sleep(0.1)

    # Example: Read 6 bytes from the device
    data = i2c.readfrom(device_address, 6)
    print("Received Data:", data)

else:
    print(f"Device at address {hex(device_address)} not found.")
```

Python standard libraries and micro-libraries

- I2C **Class:** a two-wire serial protocol

- **Constructor**
  - `class machine.I2C(id, *, scl, sda, freq=400000, timeout=50000)`

  - Parameters
    - `id`: Specifies the I2C bus to use. On some devices
      - Only `I2C(0)` and `I2C(1)` may be available.
    - `scl`: Pin object for the clock line (SCL).
    - `sda`: Pin object for the data line (SDA).
    - `freq`: (Optional) Frequency of the I2C bus. Default is 400kHz, which is a typical I2C frequency.

- **Methods**:
  - `init() / deinit()`: Initialize or re-initialize / de-initialize the I2C bus.
  - `scan()`: Returns a list of addresses of I2C devices connected to the bus.
  - `readfrom(addr, nbytes, stop=True)`: Reads a specified number of bytes from the device at the given address.
  - `writeto(addr, buf, stop=True)`: Writes bytes from a buffer to the device at the specified address.

  - …and many more methods: **https://docs.micropython.org/en/latest/library/machine.I2C.html**

Example: Transmit and Receive data via I2C

```
from machine import I2C, Pin
import time

# Initialize I2C on the Pico with SDA on GP8 and SCL on GP9
i2c = I2C(0, scl=Pin(9), sda=Pin(8))

# Scan for I2C devices on the bus
devices = i2c.scan()

if devices:
    print("I2C devices found:", devices)
else:
    print("No I2C devices found")

# Define the address of the device you want to communicate with
device_address = 0x68

# Check if the device is found before attempting to communicate
if device_address in devices:
    print(f"Communicating with device at address: {hex(device_address)}")

    # Example: Write data to the device
    i2c.writeto(device_address, b'\x01')  # Sending byte 0x01 to the device

    # Pause before reading data
    time.sleep(0.1)

    # Example: Read 6 bytes from the device
    data = i2c.readfrom(device_address, 6)
    print("Received Data:", data)

else:
    print(f"Device at address {hex(device_address)} not found.")
```

SPI (Serial Peripheral Interface)

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

Overview of SPI

- …a popular communication protocol for high-speed communication between one master and several slaves
  - called S P I (typically) or 'spy'
  - Developed in 1982 by Philips Semiconductor (...now NXP Semiconductor)

Illustration: https://ai.thestempedia.com/docs/evive/evive-technical-specifications/arduino-core-interface/i2c-communication/ and https://www.ti.com/lit/an/sbaa565/sbaa565.pdf?ts=1727931531869

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

Overview of SPI

- **Master-Slave Architecture**
  (previously mentioned…)

- **Multi-Device Communication**
  - …typically one master, multiple slaves…

Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

Overview of SPI

- **Four-Wire Interface:**

Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

Overview of SPI

- **Four-Wire Interface:**
  - **SS/CS (Slave Select/Chip Select) line**
    - …enables communication with specific slaves.
    - …no addressing…

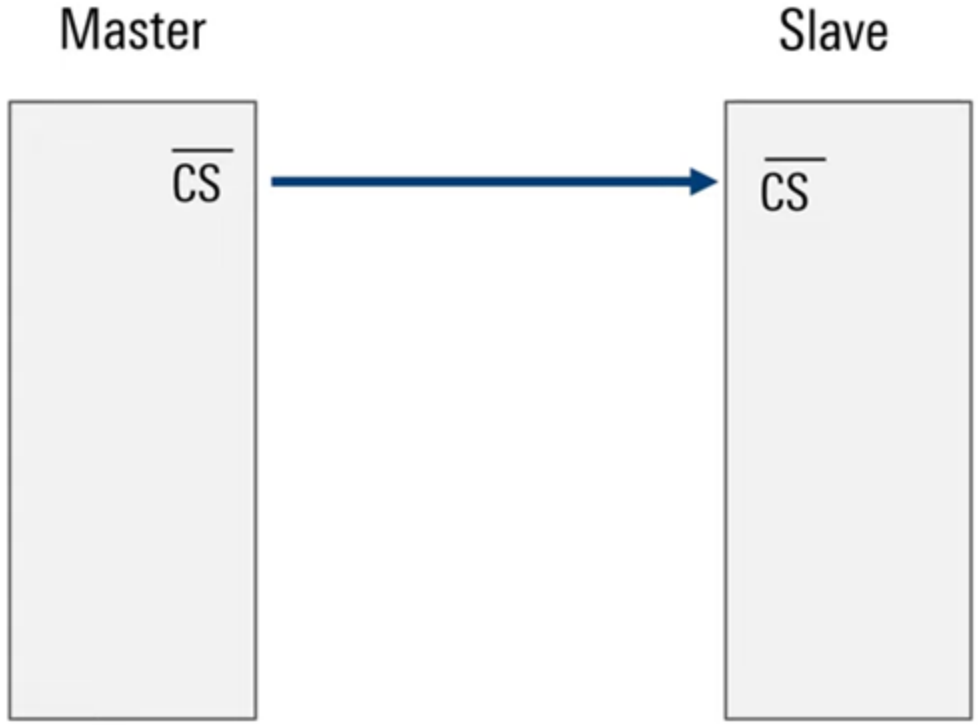


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

Overview of SPI

- **Four-Wire Interface:**
  - SS/CS (Slave Select/Chip Select) line
    - …enables communication with specific slaves.
    - …no addressing…
  - **SCLK (Serial Clock) line**
    - …carries the clock signal
    - Synchronizes data transmission.

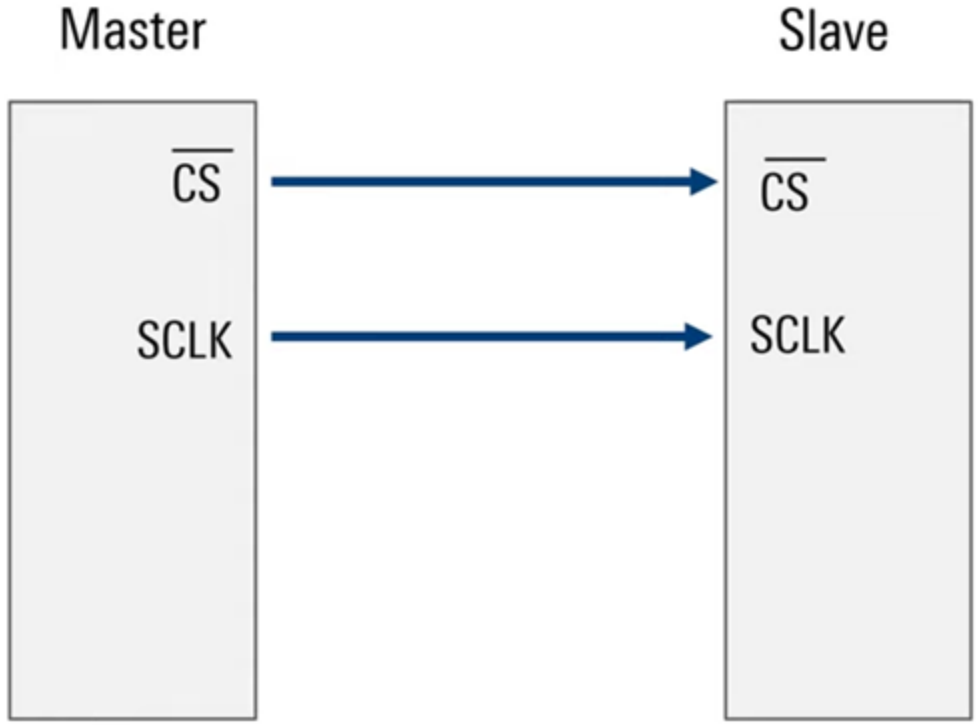


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

Overview of SPI

- **Four-Wire Interface:**
  - SS/CS (Slave Select/Chip Select) line
    - …enables communication with specific slaves.
    - …no addressing…
  - SCLK (Serial Clock) line
    - …carries the clock signal
    - Synchronizes data transmission.
  - **MOSI (Master Out Slave In) line** *(transmitting)*
    - …data from master to slave(s).

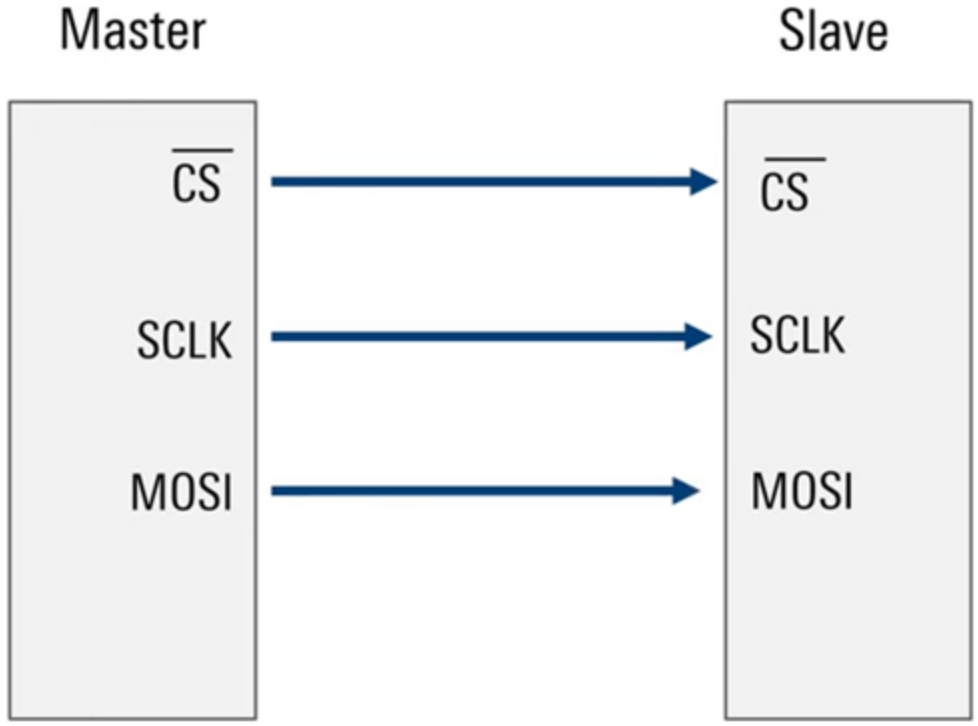


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

Overview of SPI

- **Four-Wire Interface:**
  - SS/CS (Slave Select/Chip Select) line
    - …enables communication with specific slaves.
    - …no addressing…
  - SCLK (Serial Clock) line
    - …carries the clock signal
    - Synchronizes data transmission.
  - MOSI (Master Out Slave In) line
    - …data from master to slave(s).
  - **MISO (Master In Slave Out) line** *(receiving)*
    - …data from slave(s) to master

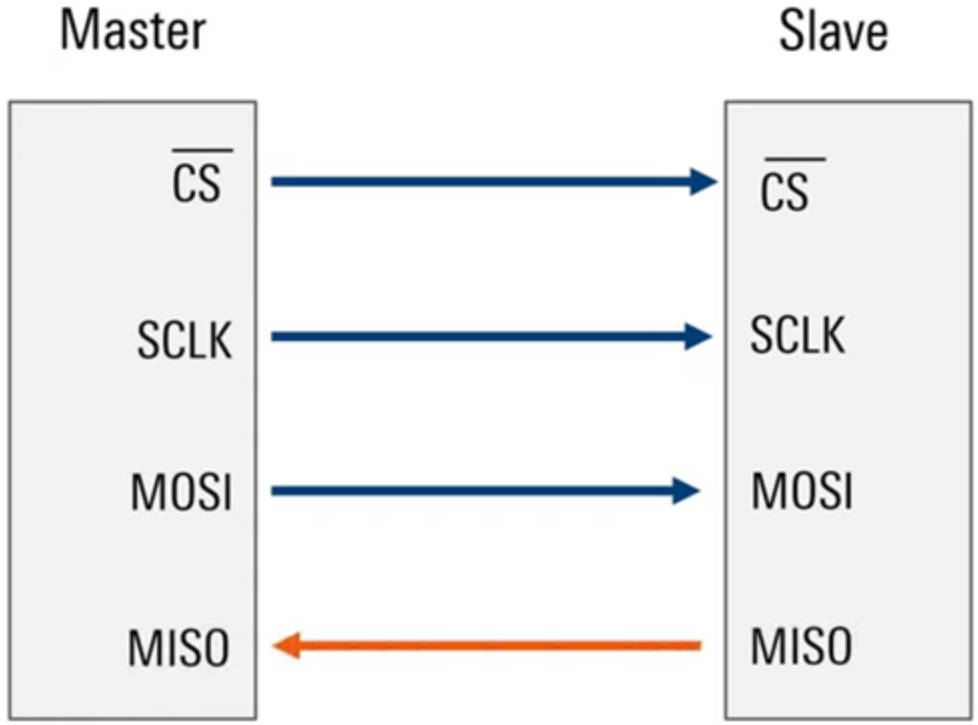


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

SPI frames

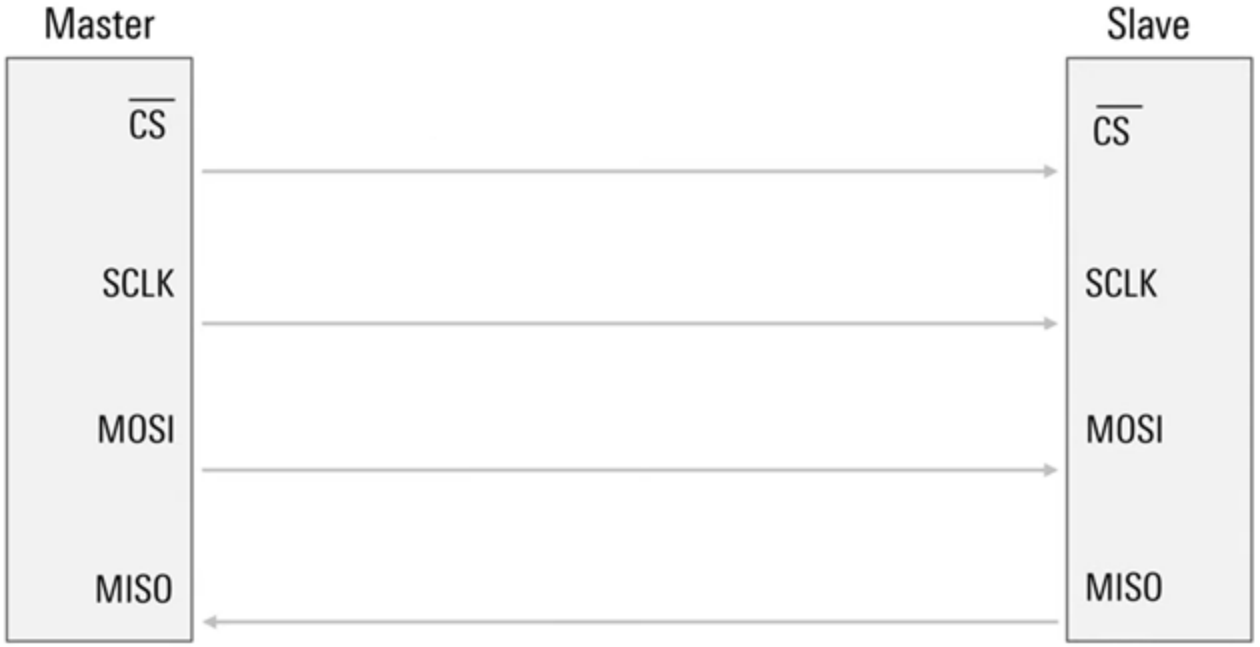


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

SPI frames

1. **Starting the communication (CS pulled LOW)**
   - Master indicates to the slave(s) that data will be transmitted.
   - Why the overbar? Typically active LOW…
   - A simple way to address a target slave
     - Or multiple target slaves, either using the same CS line or multiple CS lines

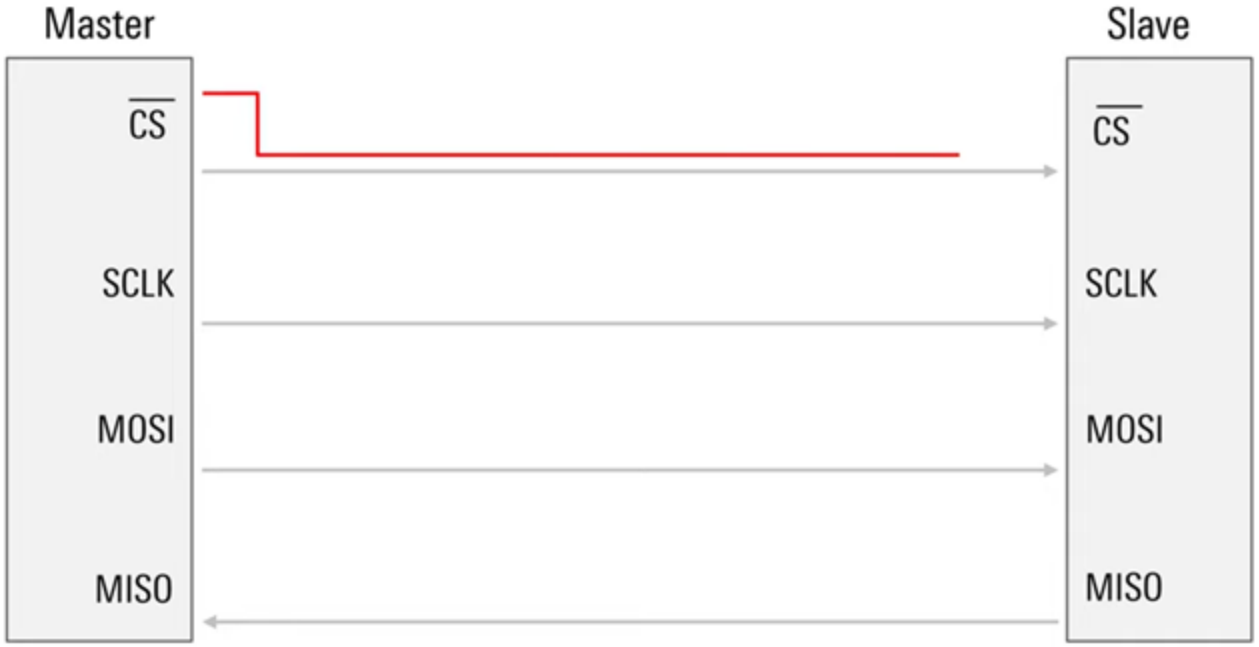


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

SPI frames

1. **Starting the communication ($\overline{CS}$ pulled LOW)**
2. **Starts sending the clock signal (SCLK)**
    a. Synchronizes data transmission
        i. Slaves doesn't need their own clock

Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

SPI frames

1. **Starting the communication ($\overline{CS}$ pulled LOW)**
2. **Starts sending the clock signal (SCLK)**
   a. Synchronizes data transmission
      i. Slaves doesn't need their own clock
   b. No standard clock speed
      i. typically in the range of MHz
      ii. …and usually faster than I²C and UART

Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

SPI frames

1. **Starting the communication ($\overline{CS}$ pulled LOW)**
2. **Starts sending the clock signal (SCLK)**
   a. Synchronizes data transmission
      i. Slaves doesn't need their own clock
   b. No standard clock speed
      i. typically in the range of MHz
      ii. …and usually faster than I²C and UART

   c. Two important configurations

Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

SPI frames

1. **Starting the communication (CS pulled LOW)**
2. **Starts sending the clock signal (SCLK)**
   a. Synchronizes data transmission
      i. Slaves doesn't need their own clock
   b. No standard clock speed
      i. typically in the range of MHz
      ii. …and usually faster than I²C and UART

   c. Two important configurations
      i. CPOL (Clock polarity)
         1. CPOL = 0 (Leading edge: RISING)
         2. CPOL = 1 (Leading edge: FALLING)

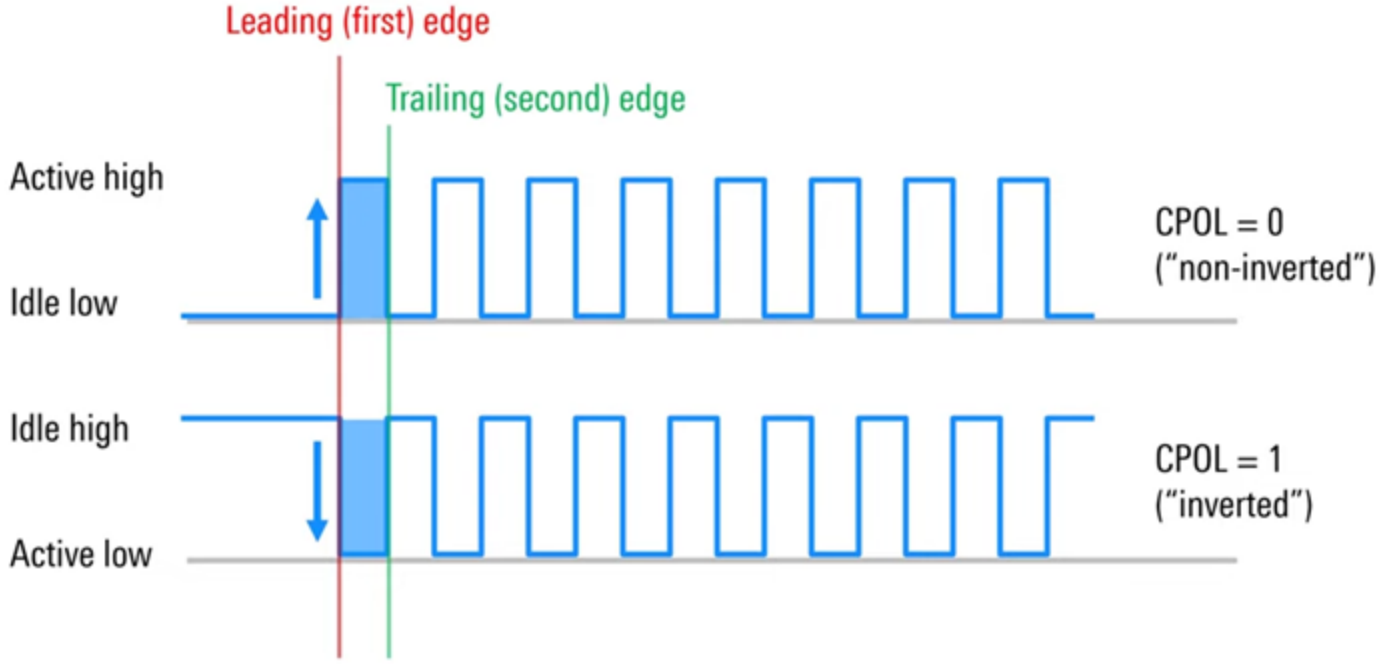


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

SPI frames

1. **Starting the communication (CS pulled LOW)**
2. **Starts sending the clock signal (SCLK)**
   a. Synchronizes data transmission
      i. Slaves doesn't need their own clock
   b. No standard clock speed
      i. typically in the range of MHz
      ii. …and usually faster than I²C and UART

   c. Two important configurations
      i. CPOL (**C**lock **pol**arity)

      ii. CPHA (**C**lock **pha**se)
         1. CPHA = 0: Data sample on leading edge
         2. CPHA = 1: Data sample on trailing edge

Here CPOL = 0, because the leading edge is rising.

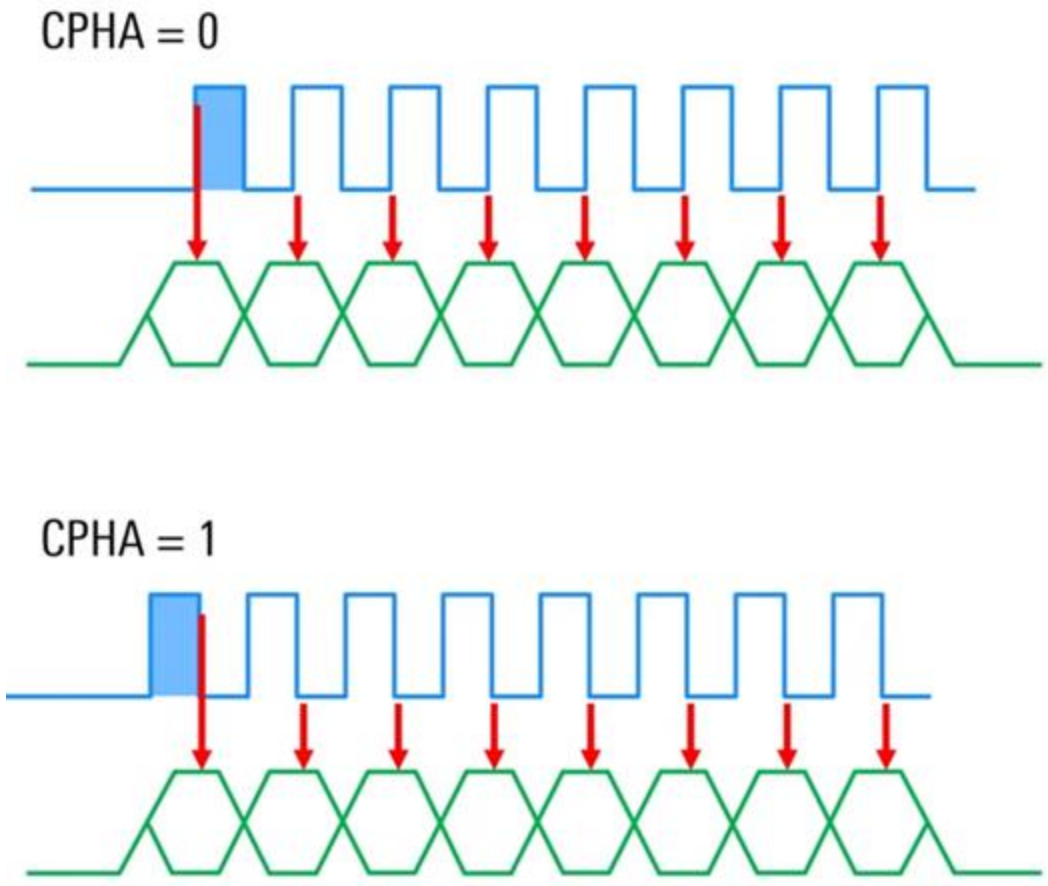
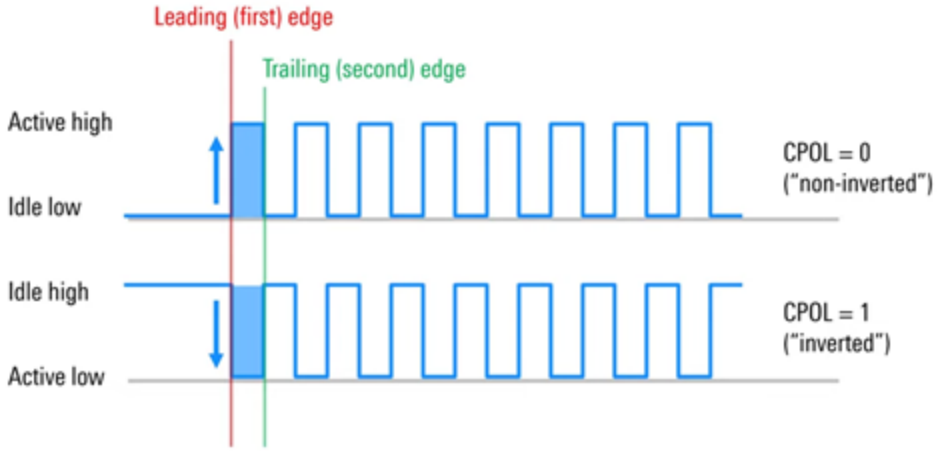
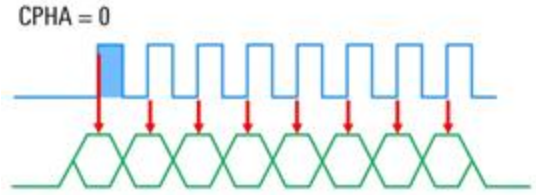


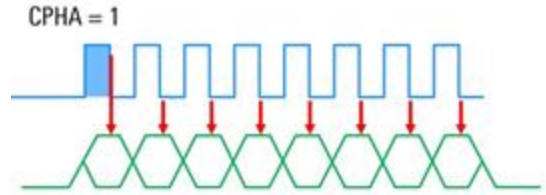
Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, sy
communication between a mas
more slaves."







SPI frames

1. **Starting the communication (CS pulled LOW)**
2. **Starts sending the clock signal (SCLK)**
   a. Synchronizes data transmission
      i. Slaves doesn't need their own clock
   b. No standard clock speed
      i. typically in the range of MHz
      ii. …and usually faster than I²C and UART

   c. Two important configurations: CPOL and CPHA

   d. SPI mode: Four possible combinations…
      i. **Mode 0 - 3**, depending on the configuration of CPOL and CPHA

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

SPI frames

1. **Starting the communication ($\overline{CS}$ pulled LOW)**
2. **Starts sending the clock signal (SCLK)**
3. **Data exchange (MOSI/MISO)**
    a. Master starts sending bits to the slave(s)
        i. …using the MOSI line

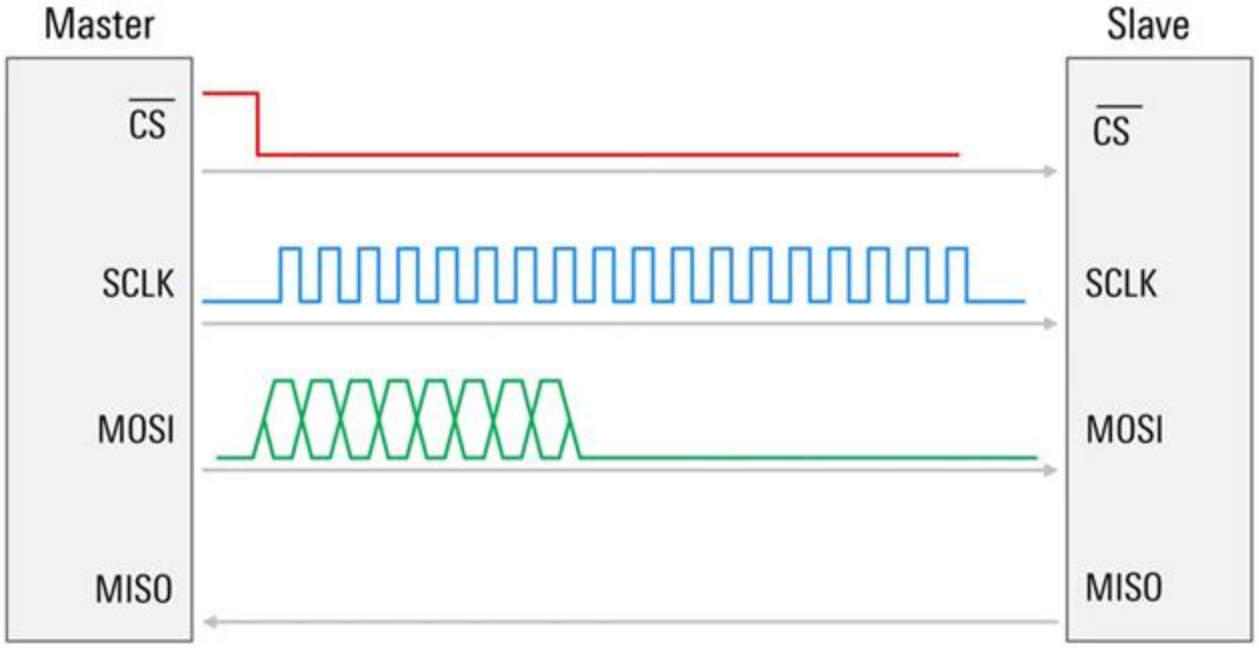


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

SPI frames

1. **Starting the communication ($\overline{CS}$ pulled LOW)**
2. **Starts sending the clock signal (SCLK)**
3. **Data exchange (MOSI/MISO)**
    a. Master starts sending bits to the slave(s)
        i. …using the MOSI line

    b. …and the slave(s) can send data to the master
        i. …using the MISO line

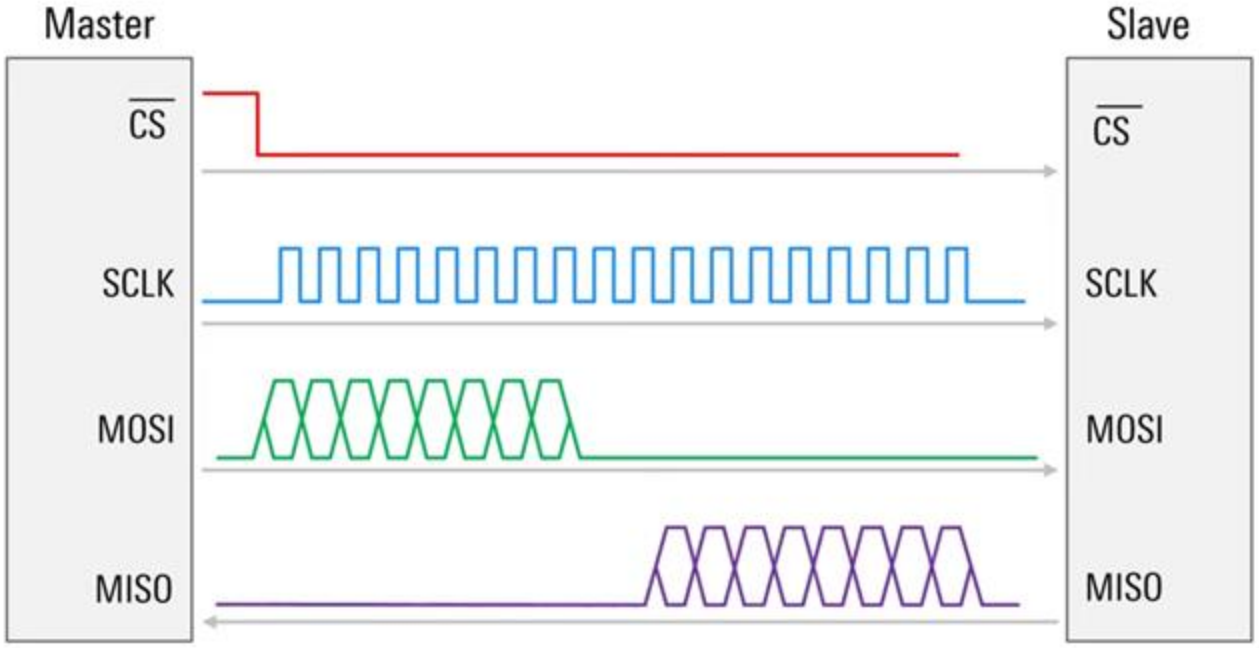


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

SPI frames

1. **Starting the communication ($\overline{CS}$ pulled LOW)**
2. **Starts sending the clock signal (SCLK)**
3. **Data exchange (MOSI/MISO)**
   a. Master starts sending bits to the slave(s)
      i. …using the MOSI line

   b. …and the slave(s) can send data to the master
      i. …using the MISO line

   c. Usually sent in bytes (8 bits)
      i. …and multiples bytes can be transferred

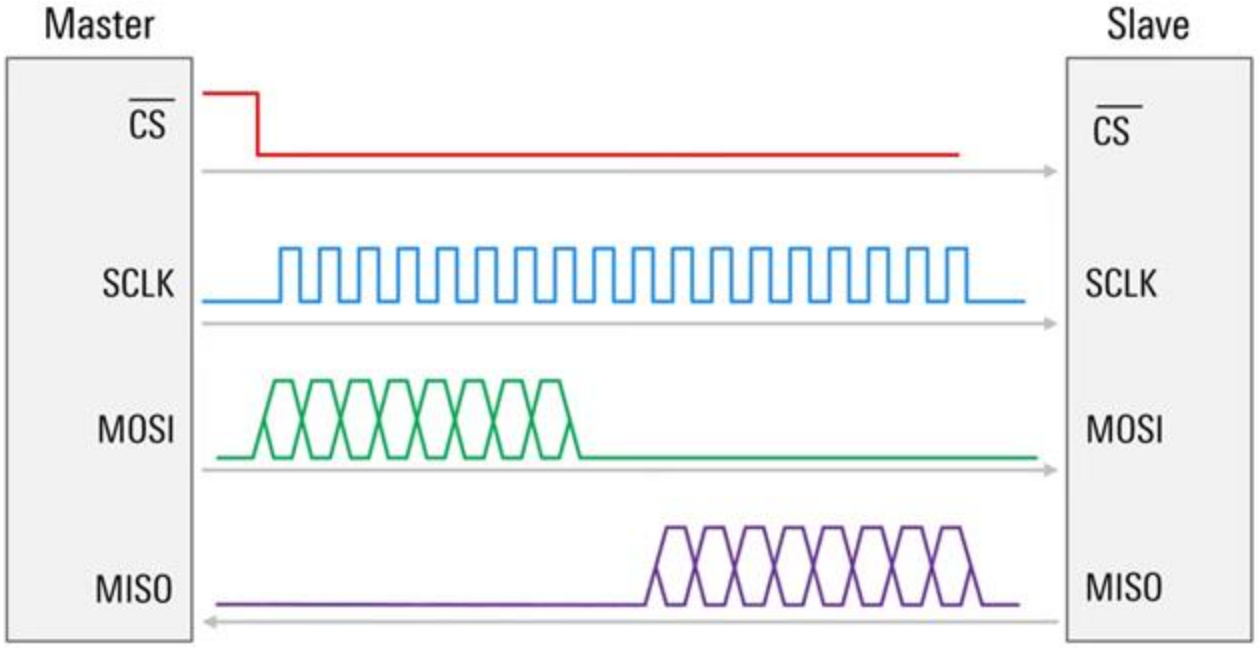


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

SPI (Serial Peripheral Interface)

"...a four-wire, full-duplex, synchronous communication between a master and one or more slaves."

SPI frames

1. **Starting the communication ($\overline{CS}$ pulled LOW)**
2. **Starts sending the clock signal (SCLK)**
3. **Data exchange (MOSI/MISO)**
4. **Ending the communication ($\overline{CS}$ pulled HIGH)**

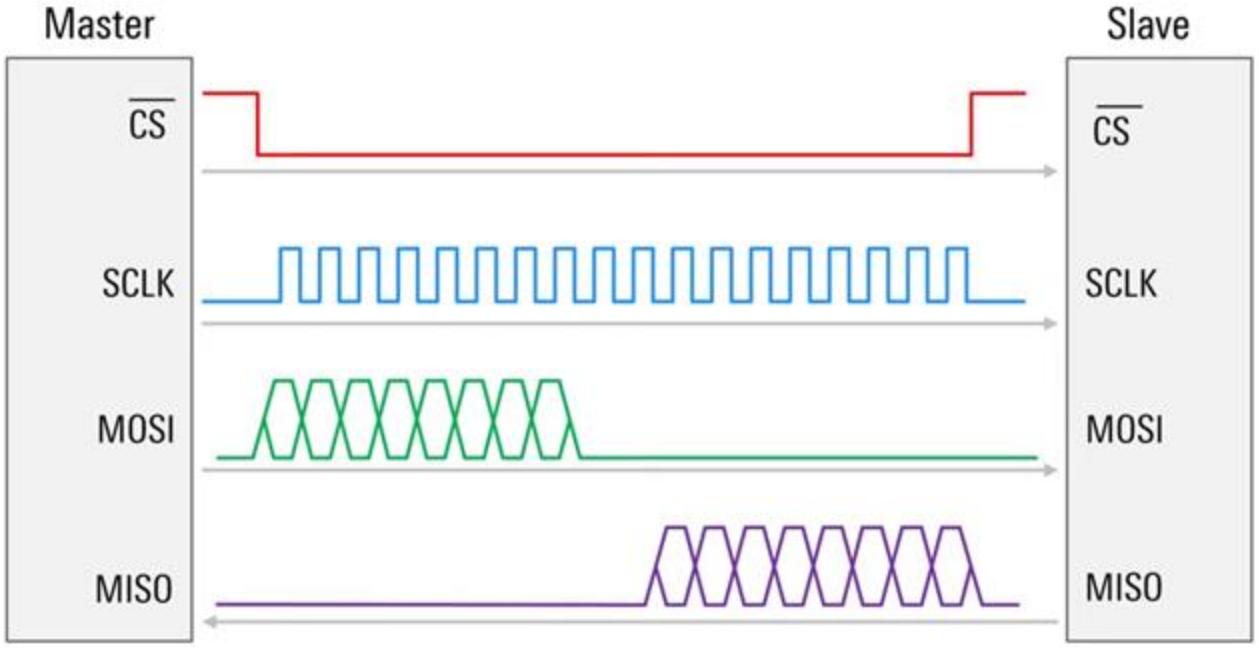


Illustration: https://www.youtube.com/watch?v=CAvawEcxoPU

Python standard libraries and micro-libraries

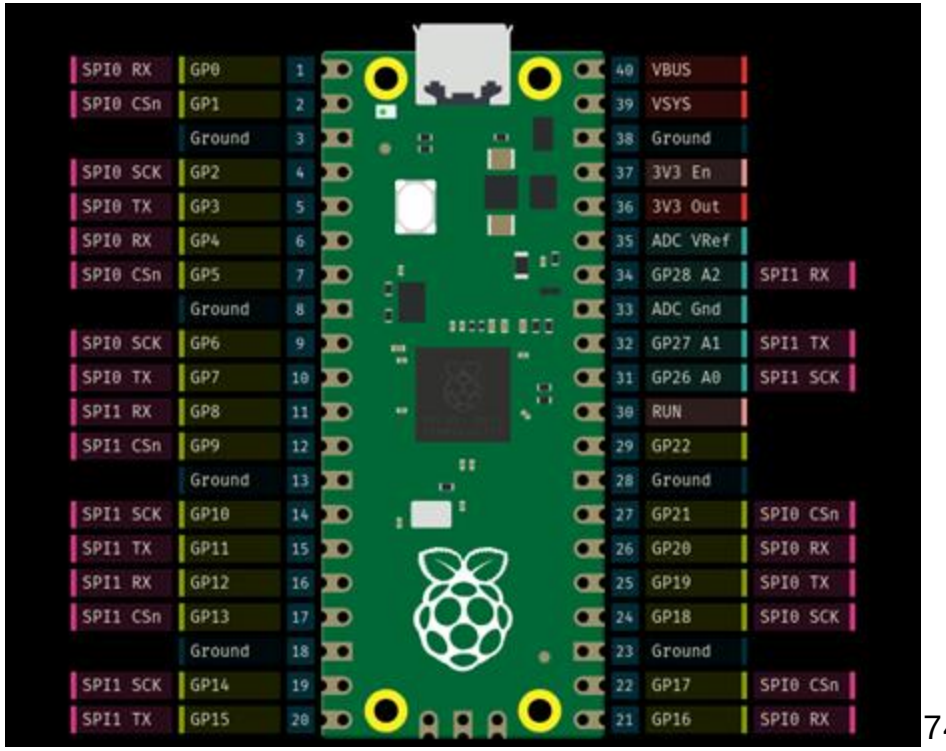
- SPI **Class:** a Serial Peripheral Interface bus protocol (controller side)

- **Constructor**
  - `class machine.SPI(id, baudrate=1000000, *, polarity=0, phase=0, bits=8, firstbit=SPI.MSB, sck=None, mosi=None, miso=None, pins=(SCK, MOSI, MISO))`

  - Parameters
    - `id`: Specifies the SPI bus to use.
      - On some devices, only `I2C(0)` and `I2C(1)` may be available.
    - `baudrate`: The clock speed, with a default of 1 MHz.
    - `polarity`: Determines the idle state of the clock (0 = low, 1 = high).
    - `phase`: Specifies when data is sampled relative to the clock edge (0 or 1).
    - `bits`: Number of bits per transfer (commonly 8).
    - `firstbit`: Defines the bit order (SPI.MSB or SPI.LSB).
    - `sck, mosi, miso`: Specify the pins for
      - the clock (SCK),
      - Master Out Slave In (MOSI), and
      - Master In Slave Out (MISO) lines.
    - `pins`: A tuple specifying SCK, MOSI, and MISO pins, as an alternative to individually specifying each pin.

Classes (`machine` module)

- `class Pin` – control I/O pins
- `...`
- `class PWM` – pulse width modulation
- `class UART` – duplex serial communication bus
- `class SPI` – a Serial Peripheral Interface bus protocol (controller side)
- `class I2C` – a two-wire serial protocol
- `class I2S` – Inter-IC Sound bus protocol
- `...`
- `class USBDevice` – USB Device driver



Python standard libraries and micro-libraries

- SPI **Class:** a Serial Peripheral Interface bus protocol (controller side)

- **Constructor**
  - `class machine.SPI(id, baudrate=1000000, *, polarity=0, phase=0, bits=8, firstbit=SPI.MSB, sck=None, mosi=None, miso=None, pins=(SCK, MOSI, MISO))`

- **Methods**:
  - `init() / deinit()`: Initialize or re-initialize / de-initialize the SPI bus.
  - `read(nbytes, write=0x00)`: Reads a specified number of bytes from the device at the given address, while continuously writing the single byte given by write.
  - `write(buf)`: Writes bytes from a buffer to the device at the specified address.
  - `readinto/write_readinto:` Reads/writes data into a provided buffer.
  - …and many more methods: https://docs.micropython.org/en/latest/library/machine.SPI.html

Example: Transmit and Receive data via SPI

```
from machine import Pin, SPI
import time

# Initialize SPI on the master Pico
spi = SPI(0, baudrate=1_000_000, polarity=0, phase=0, sck=Pin(18), mosi=Pin(19), miso=Pin(16))

cs = Pin(17, Pin.OUT)

# Function to send data to the slave and receive a response
def send_receive(data):
    cs.value(0)  # Set CS low to select the slave

    spi.write(bytes([data]))  # Send a single byte to the slave
    response = spi.read(1)  # Read a single byte from the slave

    cs.value(1)  # Set CS high to deselect the slave
    print(f"Sent: {data}, Received: {response[0]}")

while True:
    send_receive(42)  # Send 42 to the slave
    time.sleep(1)  # Wait 1 second before sending the next byte
```

Python standard libraries and micro-libraries

- SPI **Class:** a Serial Peripheral Interface bus protocol (controller side)

- **Constructor**
  - `class machine.SPI(id, baudrate=1000000, *, polarity=0, phase=0, bits=8, firstbit=SPI.MSB, sck=None, mosi=None, miso=None, pins=(SCK, MOSI, MISO))`

- **Methods**:
  - `init() / deinit()`: Initialize or re-initialize / de-initialize the SPI bus.
  - `read(nbytes, write=0x00)`: Reads a specified number of bytes from the device at the given address, while continuously writing the single byte given by write.
  - `write(buf)`: Writes bytes from a buffer to the device at the specified address.
  - `readinto/write_readinto:` Reads/writes data into a provided buffer.
  - …and many more methods: https://docs.micropython.org/en/latest/library/machine.SPI.html

Example: Transmit and Receive data via SPI

```
from machine import Pin, SPI
import time

# Initialize SPI on the master Pico
spi = SPI(0, baudrate=1_000_000, polarity=0, phase=0, sck=Pin(18), mosi=Pin(19), miso=Pin(16))

cs = Pin(17, Pin.OUT)

# Function to send data to the slave and receive a response
def send_receive(data):
    cs.value(0)  # Set CS low to select the slave

    spi.write(bytes([data]))  # Send a single byte to the slave
    response = spi.read(1)  # Read a single byte from the slave

    cs.value(1)  # Set CS high to deselect the slave
    print(f"Sent: {data}, Received: {response[0]}")

while True:
    send_receive(42)  # Send 42 to the slave
    time.sleep(1)  # Wait 1 second before sending the next byte
```

Python standard libraries and micro-libraries

- SPI **Class:** a Serial Peripheral Interface bus protocol (controller side)

- **Constructor**
  - `class machine.SPI(id, baudrate=1000000, *, polarity=0, phase=0, bits=8, firstbit=SPI.MSB, sck=None, mosi=None, miso=None, pins=(SCK, MOSI, MISO))`

- **Methods**:
  - `init() / deinit()`: Initialize or re-initialize / de-initialize the SPI bus.
  - `read(nbytes, write=0x00)`: Reads a specified number of bytes from the device at the given address, while continuously writing the single byte given by write.
  - `write(buf)`: Writes bytes from a buffer to the device at the specified address.
  - `readinto/write_readinto:` Reads/writes data into a provided buffer.
  - …and many more methods: https://docs.micropython.org/en/latest/library/machine.SPI.html

Example: Transmit and Receive data via SPI

```
from machine import Pin, SPI
import time

# Initialize SPI on the master Pico
spi = SPI(0, baudrate=1_000_000, polarity=0, phase=0, sck=Pin(18), mosi=Pin(19),
miso=Pin(16))

cs = Pin(17, Pin.OUT)

# Function to send data to the slave and receive a response
def send_receive(data):
    cs.value(0)  # Set CS low to select the slave

    spi.write(bytes([data]))  # Send a single byte to the slave
    response = spi.read(1)  # Read a single byte from the slave

    cs.value(1)  # Set CS high to deselect the slave
    print(f"Sent: {data}, Received: {response[0]}")

while True:
    send_receive(42)  # Send 42 to the slave
    time.sleep(1)  # Wait 1 second before sending the next byte
```

Python standard libraries and micro-libraries

- SPI **Class:** a Serial Peripheral Interface bus protocol (controller side)

- **Constructor**
  - `class machine.SPI(id, baudrate=1000000, *, polarity=0, phase=0, bits=8, firstbit=SPI.MSB, sck=None, mosi=None, miso=None, pins=(SCK, MOSI, MISO))`

- **Methods**:
  - `init() / deinit()`: Initialize or re-initialize / de-initialize the SPI bus.
  - `read(nbytes, write=0x00)`: Reads a specified number of bytes from the device at the given address, while continuously writing the single byte given by write.
  - `write(buf)`: Writes bytes from a buffer to the device at the specified address.
  - `readinto/write_readinto:` Reads/writes data into a provided buffer.
  - …and many more methods: https://docs.micropython.org/en/latest/library/machine.SPI.html

Example: Transmit and Receive data via SPI

```
from machine import Pin, SPI
import time

# Initialize SPI on the master Pico
spi = SPI(0, baudrate=1_000_000, polarity=0, phase=0, sck=Pin(18), mosi=Pin(19),
miso=Pin(16))

cs = Pin(17, Pin.OUT)

# Function to send data to the slave and receive a response
def send_receive(data):
    cs.value(0)  # Set CS low to select the slave

    spi.write(bytes([data]))  # Send a single byte to the slave
    response = spi.read(1)  # Read a single byte from the slave

    cs.value(1)  # Set CS high to deselect the slave
    print(f"Sent: {data}, Received: {response[0]}")

while True:
    send_receive(42)  # Send 42 to the slave
    time.sleep(1)  # Wait 1 second before sending the next byte
```

Universal Asynchronous Receiver-Transmitter (UART)

UART (Universal Asynchronous Receiver-Transmitter)

"...a two-wire, full-duplex, asynchronous serial communication protocol for direct, point-to-point communication between two devices."

Overview of UART

- …a popular communication protocol for simple, point-to-point communication, particularly over longer distances.
- Asynchronous Transmission
- Full-Duplex (typically)
- Point-to-Point

Illustration: https://www.youtube.com/watch?v=sTHckUyxwp8

UART (Universal Asynchronous Receiver-Transmitter)

"...a two-wire, full-duplex, asynchronous serial communication protocol for direct, point-to-point communication between two devices."

Overview of UART

- **Two-Wire Interface:**

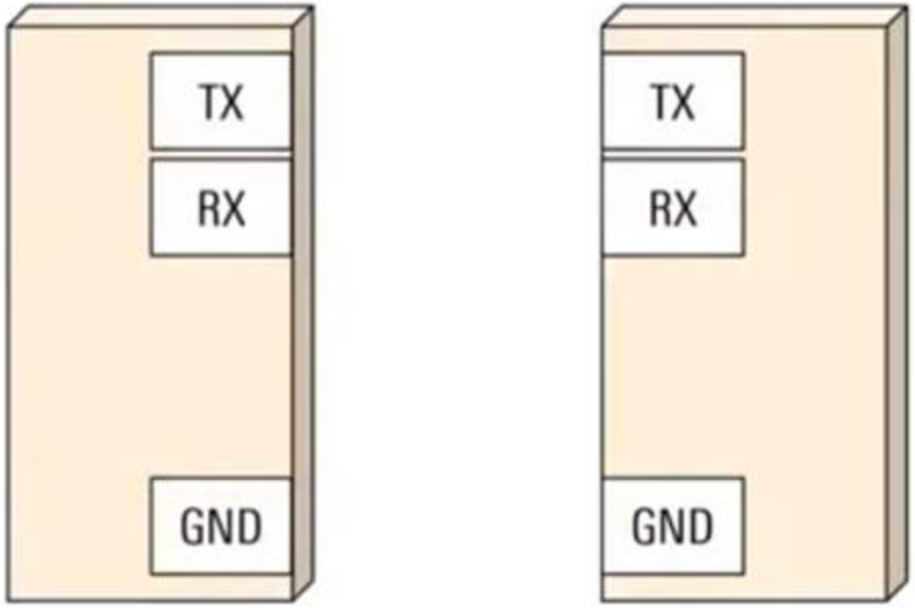


Illustration: https://www.youtube.com/watch?v=sTHckUyxwp8

UART (Universal Asynchronous Receiver-Transmitter)

"...a two-wire, full-duplex, asynchronous serial communication protocol for direct, point-to-point communication between two devices."

Overview of UART

- **Two-Wire Interface:**
  - TX (Transmit) line
    - …carries data from the transmitter to the other connected device (RX)

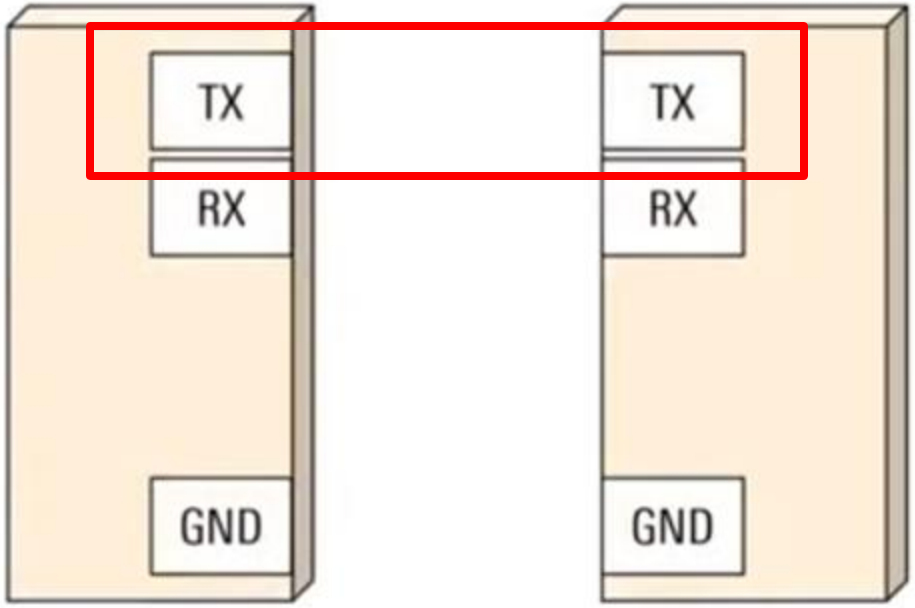


Illustration: https://www.youtube.com/watch?v=sTHckUyxwp8

UART (Universal Asynchronous Receiver-Transmitter)

"...a two-wire, full-duplex, asynchronous serial communication protocol for direct, point-to-point communication between two devices."

Overview of UART

- **Two-Wire Interface:**
  - TX (Transmit) line
    - …carries data from the transmitter to the other connected device (RX)
  - RX (Receive) line
    - …carries data transmitted from the other connected device (TX)

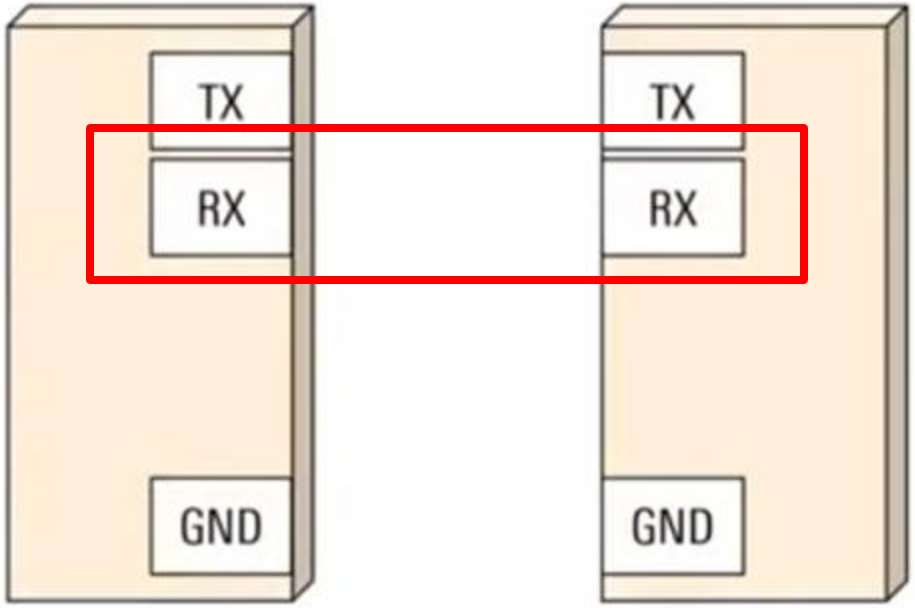
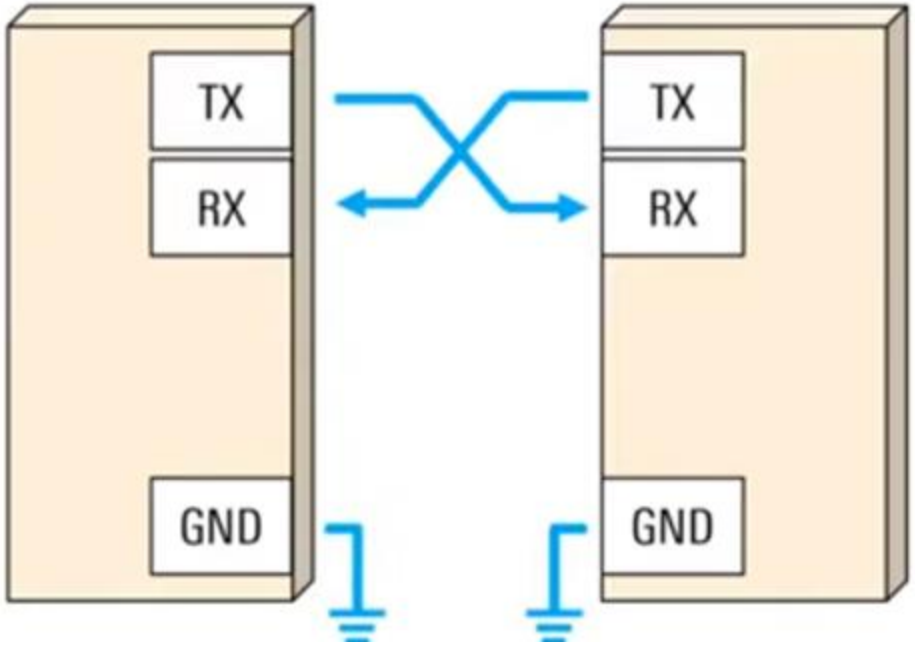


Illustration: https://www.youtube.com/watch?v=sTHckUyxwp8

UART (Universal Asynchronous Receiver-Transmitter)

"...a two-wire, full-duplex, asynchronous serial communication protocol for direct, point-to-point communication between two devices."



Overview of UART

- **Two-Wire Interface:**
  - TX (Transmit) line
    - …carries data from the transmitter to the other connected device (RX)
  - RX (Receive) line
    - …carries data transmitted from the other connected device (TX)
  - (in each direction)

Illustration: https://www.youtube.com/watch?v=sTHckUyxwp8

UART (Universal Asynchronous Receiver-Transmitter)

"...a two-wire, full-duplex, asynchronous serial communication protocol for direct, point-to-point communication between two devices."

Overview of UART

- **…plex-ing:** Can support
  - **Simplex**: Only one devices sends data

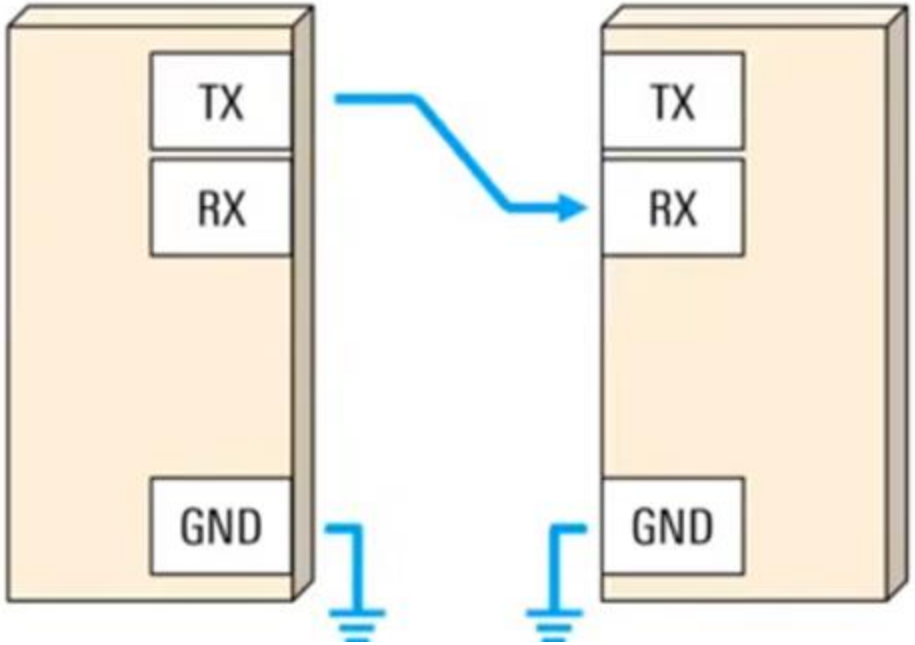


Illustration: https://www.youtube.com/watch?v=sTHckUyxwp8

UART (Universal Asynchronous Receiver-Transmitter)

"...a two-wire, full-duplex, asynchronous serial communication protocol for direct, point-to-point communication between two devices."

Overview of UART

- **…plex-ing:** Can support
  - Simplex: Only one devices sends data
  - **Half-duplex**: Both devices sends data, but only one at the time

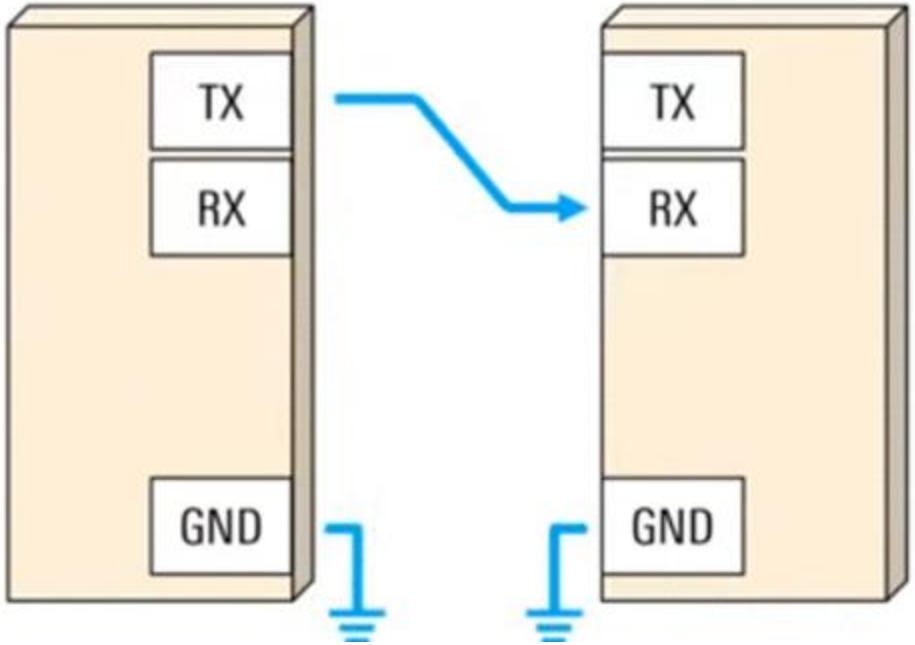


Illustration: https://www.youtube.com/watch?v=sTHckUyxwp8

UART (Universal Asynchronous Receiver-Transmitter)

"...a two-wire, full-duplex, asynchronous serial communication protocol for direct, point-to-point communication between two devices."

Overview of UART

- **…plex-ing:** Can support
  - Simplex: Only one devices sends data
  - Half-duplex: Both devices sends data, but only one at the time
  - **Full-duplex**: Both devices sends data simultaneously

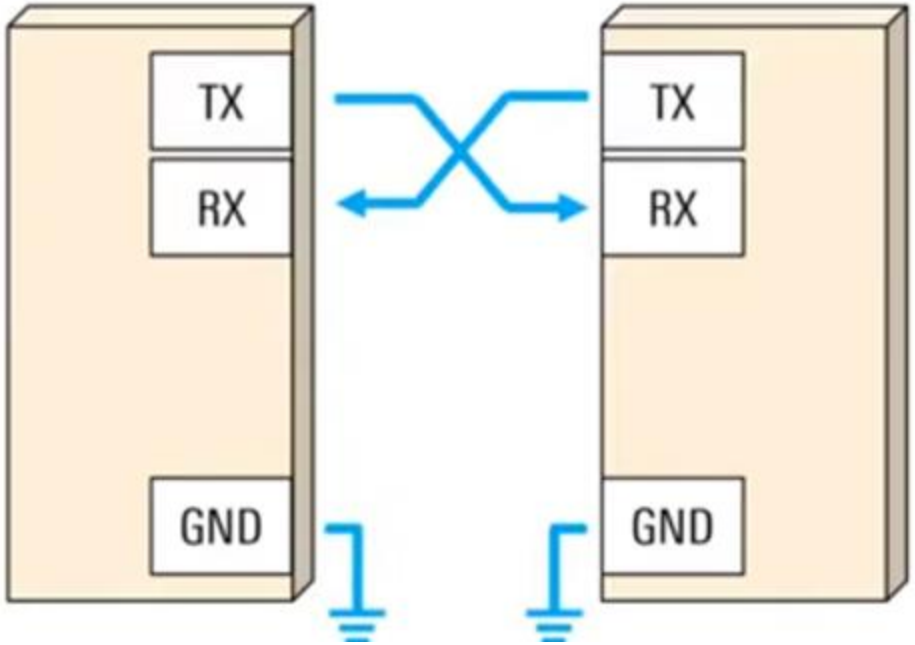


Illustration: https://www.youtube.com/watch?v=sTHckUyxwp8

UART (Universal Asynchronous Receiver-Transmitter)

"...a two-wire, full-duplex, asynchronous serial communication protocol for direct, point-to-point communication between two devices."

UART parameters and frames

- **Baud-rate**
  - …since the transmission is asynchronous, both devices must therefore transmit/receive at the same **known** speed (*baud rate*)
    - measured in **bits per second (bps)**
    - Common baud rates include
      - 9600 bps,
      - 19200 bps,
      - 38400 bps,
      - 57600 bps, and
      - 115200 bps.

Illustration: https://www.youtube.com/watch?v=sTHckUyxwp8

UART (Universal Asynchronous Receiver-Transmitter)

"...a two-wire, full-duplex, asynchronous serial communication protocol for direct, point-to-point communication between two devices."

UART parameters and frames

- **Frame structure**

Illustration: https://www.youtube.com/watch?v=sTHckUyxwp8

UART (Universal Asynchronous Receiver-Transmitter)

"...a two-wire, full-duplex, asynchronous serial communication protocol for direct, point-to-point communication between two devices."

UART parameters and frames

- **Frame structure**
  - **Idle:** The TX line is held HIGH

Illustration: https://www.youtube.com/watch?v=sTHckUyxwp8

UART (Universal Asynchronous Receiver-Transmitter)

"...a two-wire, full-duplex, asynchronous serial communication protocol for direct, point-to-point communication between two devices."

UART parameters and frames

- **Frame structure**
  - **Idle:** The TX line is held HIGH
  - **Start Bit**: At the beginning of each data packet, a start bit is sent.
    - It indicates the start of a data transmission
    - Pulls the line from its idle state (HIGH) to LOW.

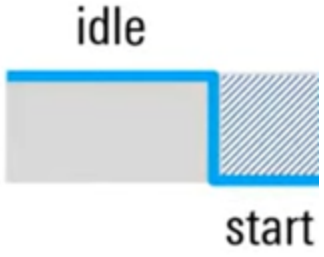


Illustration: https://www.youtube.com/watch?v=sTHckUyxwp8

UART (Universal Asynchronous Receiver-Transmitter)

"...a two-wire, full-duplex, asynchronous serial communication protocol for direct, point-to-point communication between two devices."

UART parameters and frames

- **Frame structure**
  - **Idle:** The TX line is held HIGH
  - **Start Bit**: At the beginning of each data packet, a start bit is sent.
  - **Data Bits**: After the start bit, the actual data is transmitted,
    - Typically in chunks of 7, 8, or 9 bits.
    - Example: 7-bit ASCII 'S'
      - 0x52 = 0b1010011
      - LSB order: 0b1100101 → Send it out

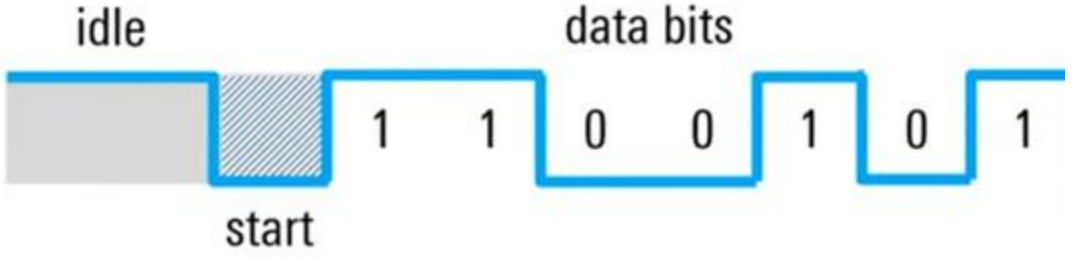


Illustration: https://www.youtube.com/watch?v=sTHckUyxwp8

UART (Universal Asynchronous Receiver-Transmitter)

"...a two-wire, full-duplex, asynchronous serial communication protocol for direct, point-to-point communication between two devices."

UART parameters and frames

- **Frame structure**
  - **Idle:** The TX line is held HIGH
  - **Start Bit**: At the beginning of each data packet, a start bit is sent.
  - **Data Bits**: After the start bit, the actual data is transmitted,
  - **Parity Bit** (Optional): Used for simple error-checking.
    - The parity bit can be
      - Even (the total number of 1's is even) or
      - Odd (the total number of 1's is odd).
    - Can only detect if a single bit has failed / flipped…

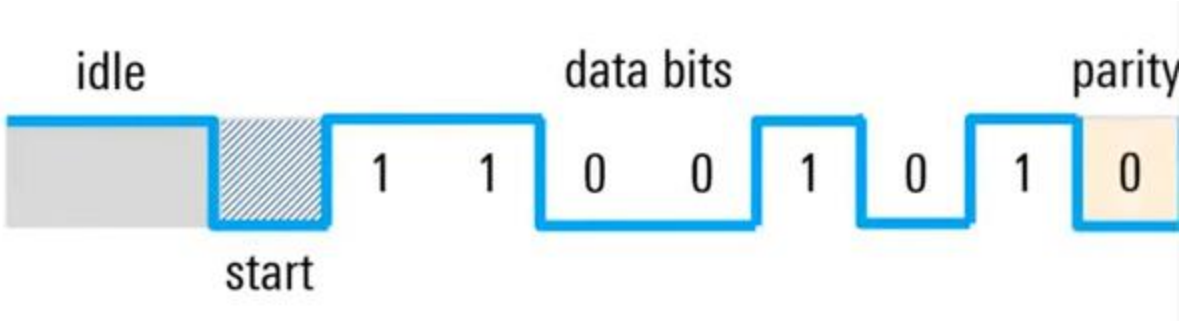


Illustration: https://www.youtube.com/watch?v=sTHckUyxwp8

Example: ASCII 'S'

- 0b1010011 = Four 1's

Using
- Even parity = 0
- Odd parity = 1
  - Five (including the parity bit)

UART (Universal Asynchronous Receiver-Transmitter)

"...a two-wire, full-duplex, asynchronous serial communication protocol for direct, point-to-point communication between two devices."

UART parameters and frames

- **Frame structure**
  - **Idle:** The TX line is held HIGH
  - **Start Bit**: At the beginning of each data packet, a start bit is sent.
  - **Data Bits**: After the start bit, the actual data is transmitted,
  - **Parity Bit** (Optional): Used for simple error-checking.
  - **Stop Bit**: At the end of the data frame, one or more stop bits are sent to signal the end of the transmission.
    - Return to Idle state / Stay HIGH for x numbers of bits
    - (Optional) specific bit pattern, but not common

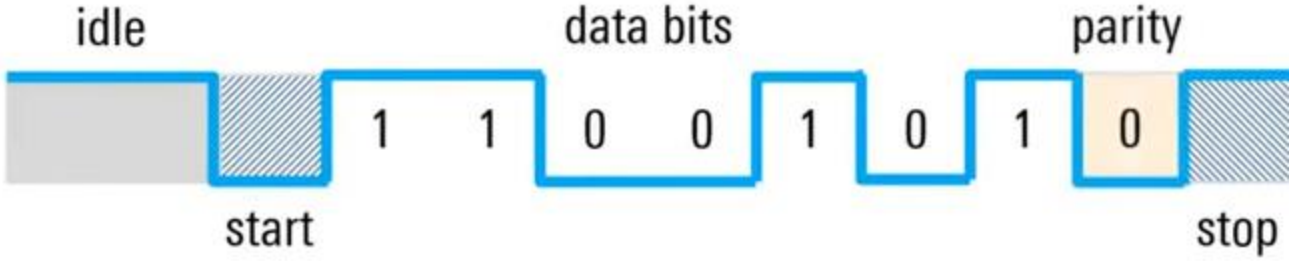


Illustration: https://www.youtube.com/watch?v=sTHckUyxwp8

UART (Universal Asynchronous Receiver-Transmitter)

"...a two-wire, full-duplex, asynchronous serial communication protocol for direct, point-to-point communication between two devices."

UART parameters and frames

- **Frame structure**
  - **Idle:** The TX line is held HIGH
  - **Start Bit**: At the beginning of each data packet, a start bit is sent.
  - **Data Bits**: After the start bit, the actual data is transmitted,
  - **Parity Bit** (Optional): Used for simple error-checking.
  - **Stop Bit**: At the end of the data frame, one or more stop bits are sent to signal the end of the transmission.
    - Back to **Idle**:This returns the line to its idle state and provides a break between successive frames.

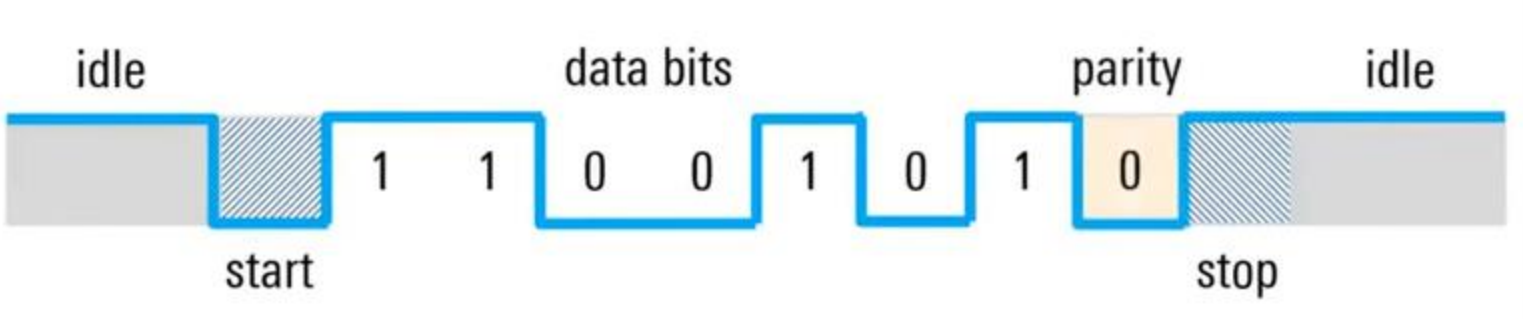


Illustration: https://www.youtube.com/watch?v=sTHckUyxwp8

Python standard libraries and micro-libraries

- UART **Class:** a two-wire serial protocol

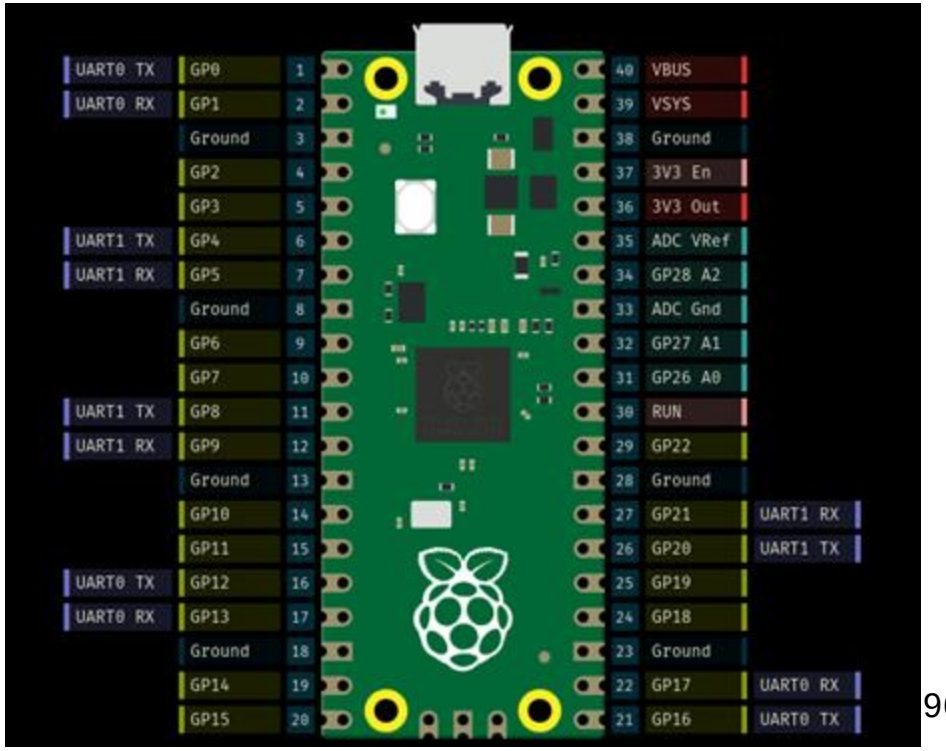
- **Constructor**
  - `class machine.UART(id, baudrate=9600, tx, rx, bits=8, parity=None, stop=1, *, ...)`

  - Parameters
    - `id`: The UART peripheral identifier (Typically 0 or 1)
    - `baudrate`: Speed of data transfer (in bits per second).
    - `tx` and `rx`: Pins for transmitting and receiving data.
    - `timeout`: Timeout in milliseconds for receiving data.
    - `bits`

- **Methods**:
  - `init() / deinit()`: Initialize or re-initialize / de-initialize the I2C bus.
  - `read(n)`: Reads up to n bytes.
  - `write(data)`: Writes the `data` string or bytes to the UART.
  - `any()`: Returns the number of bytes available in the input buffer.
  - `flush()`: Waits until all outgoing data has been sent..
  - …and many more methods:
    https://docs.micropython.org/en/latest/library/machine.UART.html

Classes (`machine` module)

- `class Pin` – control I/O pins
- `...`
- `class PWM` – pulse width modulation
- `class UART` – duplex serial communication bus
- `class SPI` – a Serial Peripheral Interface bus protocol (controller side)
- `class I2C` – a two-wire serial protocol
- `class I2S` – Inter-IC Sound bus protocol
- `...`
- `class USBDevice` – USB Device driver



Python standard libraries and micro-libraries

- UART **Class:** a two-wire serial protocol

- **Constructor**
  - `class machine.UART(id, baudrate=9600, tx, rx, bits=8, parity=None, stop=1, *, ...)`

  - Parameters
    - `id`: The UART peripheral identifier (Typically 0 or 1)
    - `baudrate`: Speed of data transfer (in bits per second).
    - `tx` and `rx`: Pins for transmitting and receiving data.
    - `timeout`: Timeout in milliseconds for receiving data.

- **Methods**:
  - `init() / deinit()`: Initialize or re-initialize / de-initialize the I2C bus.
  - `read(n)`: Reads up to n bytes.
  - `write(data)`: Writes the `data` string or bytes to the UART.
  - `any()`: Returns the number of bytes available in the input buffer.
  - `flush()`: Waits until all outgoing data has been sent..
  - …and many more methods: https://docs.micropython.org/en/latest/library/machine.UART.html

Example: Transmit and Receive data via UART

```
from machine import UART
import time

# Initialize UART on UART0 with baudrate 9600,
# and TX on GPIO 17, RX on GPIO 16
uart = UART(0, baudrate=9600, tx=17, rx=16)

# Send a message
message = "Hello, World!"
uart.write(message)
print("Sent:", message)

# Allow some time for the device to respond
time.sleep(1)

# Check if there is data to read
if uart.any():
    # Read the response (up to 20 bytes)
    response = uart.read(20)
    print("Received:", response.decode('utf-8'))

# Deinitialize UART after communication is complete
uart.deinit()
```

Python standard libraries and micro-libraries

- UART **Class:** a two-wire serial protocol

- **Constructor**
  - `class machine.UART(id, baudrate=9600, tx, rx, bits=8, parity=None, stop=1, *, ...)`

  - Parameters
    - `id`: The UART peripheral identifier (Typically 0 or 1)
    - `baudrate`: Speed of data transfer (in bits per second).
    - `tx` and `rx`: Pins for transmitting and receiving data.
    - `timeout`: Timeout in milliseconds for receiving data.

- **Methods**:
  - `init() / deinit()`: Initialize or re-initialize / de-initialize the I2C bus.
  - `read(n)`: Reads up to n bytes.
  - `write(data)`: Writes the `data` string or bytes to the UART.
  - `any()`: Returns the number of bytes available in the input buffer.
  - `flush()`: Waits until all outgoing data has been sent..
  - …and many more methods: https://docs.micropython.org/en/latest/library/machine.UART.html

Example: Transmit and Receive data via UART

```
from machine import UART
import time

# Initialize UART on UART0 with baudrate 9600,
# and TX on GPIO 17, RX on GPIO 16
uart = UART(0, baudrate=9600, tx=17, rx=16)

# Send a message
message = "Hello, World!"
uart.write(message)
print("Sent:", message)

# Allow some time for the device to respond
time.sleep(1)

# Check if there is data to read
if uart.any():
    # Read the response (up to 20 bytes)
    response = uart.read(20)
    print("Received:", response.decode('utf-8'))

# Deinitialize UART after communication is complete
uart.deinit()
```

Python standard libraries and micro-libraries

- UART **Class:** a two-wire serial protocol

- **Constructor**
  - `class machine.UART(id, baudrate=9600, tx, rx, bits=8, parity=None, stop=1, *, ...)`

  - Parameters
    - `id`: The UART peripheral identifier (Typically 0 or 1)
    - `baudrate`: Speed of data transfer (in bits per second).
    - `tx` and `rx`: Pins for transmitting and receiving data.
    - `timeout`: Timeout in milliseconds for receiving data.

- **Methods**:
  - `init() / deinit()`: Initialize or re-initialize / de-initialize the I2C bus.
  - `read(n)`: Reads up to n bytes.
  - `write(data)`: Writes the `data` string or bytes to the UART.
  - `any()`: Returns the number of bytes available in the input buffer.
  - `flush()`: Waits until all outgoing data has been sent..
  - …and many more methods: https://docs.micropython.org/en/latest/library/machine.UART.html

Example: Transmit and Receive data via UART

```
from machine import UART
import time

# Initialize UART on UART0 with baudrate 9600,
# and TX on GPIO 17, RX on GPIO 16
uart = UART(0, baudrate=9600, tx=17, rx=16)

# Send a message
message = "Hello, World!"
uart.write(message)
print("Sent:", message)

# Allow some time for the device to respond
time.sleep(1)

# Check if there is data to read
if uart.any():
    # Read the response (up to 20 bytes)
    response = uart.read(20)
    print("Received:", response.decode('utf-8'))

# Deinitialize UART after communication is complete
uart.deinit()
```

Python standard libraries and micro-libraries

- UART **Class:** a two-wire serial protocol

- **Constructor**
  - `class machine.UART(id, baudrate=9600, tx, rx, bits=8, parity=None, stop=1, *, ...)`

  - Parameters
    - `id`: The UART peripheral identifier (Typically 0 or 1)
    - `baudrate`: Speed of data transfer (in bits per second).
    - `tx` and `rx`: Pins for transmitting and receiving data.
    - `timeout`: Timeout in milliseconds for receiving data.

- **Methods**:
  - `init() / deinit()`: Initialize or re-initialize / de-initialize the I2C bus.
  - `read(n)`: Reads up to n bytes.
  - `write(data)`: Writes the `data` string or bytes to the UART.
  - `any()`: Returns the number of bytes available in the input buffer.
  - `flush()`: Waits until all outgoing data has been sent..
  - …and many more methods: https://docs.micropython.org/en/latest/library/machine.UART.html

Example: Transmit and Receive data via UART

```
from machine import UART
import time

# Initialize UART on UART0 with baudrate 9600,
# and TX on GPIO 17, RX on GPIO 16
uart = UART(0, baudrate=9600, tx=17, rx=16)

# Send a message
message = "Hello, World!"
uart.write(message)
print("Sent:", message)

# Allow some time for the device to respond
time.sleep(1)

# Check if there is data to read
if uart.any():
    # Read the response (up to 20 bytes)
    response = uart.read(20)
    print("Received:", response.decode('utf-8'))

# Deinitialize UART after communication is complete
uart.deinit()
```

Python standard libraries and micro-libraries

- UART **Class:** a two-wire serial protocol

- **Constructor**
  - `class machine.UART(id, baudrate=9600, tx, rx, bits=8, parity=None, stop=1, *, ...)`

  - Parameters
    - `id`: The UART peripheral identifier (Typically 0 or 1)
    - `baudrate`: Speed of data transfer (in bits per second).
    - `tx` and `rx`: Pins for transmitting and receiving data.
    - `timeout`: Timeout in milliseconds for receiving data.

- **Methods**:
  - `init() / deinit()`: Initialize or re-initialize / de-initialize the I2C bus.
  - `read(n)`: Reads up to n bytes.
  - `write(data)`: Writes the `data` string or bytes to the UART.
  - `any()`: Returns the number of bytes available in the input buffer.
  - `flush()`: Waits until all outgoing data has been sent..
  - …and many more methods: https://docs.micropython.org/en/latest/library/machine.UART.html

Example: Transmit and Receive data via UART

```
from machine import UART
import time

# Initialize UART on UART0 with baudrate 9600,
# and TX on GPIO 17, RX on GPIO 16
uart = UART(0, baudrate=9600, tx=17, rx=16)

# Send a message
message = "Hello, World!"
uart.write(message)
print("Sent:", message)

# Allow some time for the device to respond
time.sleep(1)

# Check if there is data to read
if uart.any():
    # Read the response (up to 20 bytes)
    response = uart.read(20)
    print("Received:", response.decode('utf-8'))

# Deinitialize UART after communication is complete
uart.deinit()
```

Python standard libraries and micro-libraries

- UART **Class:** a two-wire serial protocol

- **Constructor**
  - `class machine.UART(id, baudrate=9600, tx, rx, bits=8, parity=None, stop=1, *, ...)`

  - Parameters
    - `id`: The UART peripheral identifier (Typically 0 or 1)
    - `baudrate`: Speed of data transfer (in bits per second).
    - `tx` and `rx`: Pins for transmitting and receiving data.
    - `timeout`: Timeout in milliseconds for receiving data.

- **Methods**:
  - `init() / deinit()`: Initialize or re-initialize / de-initialize the I2C bus.
  - `read(n)`: Reads up to n bytes.
  - `write(data)`: Writes the `data` string or bytes to the UART.
  - `any()`: Returns the number of bytes available in the input buffer.
  - `flush()`: Waits until all outgoing data has been sent..
  - …and many more methods:
    https://docs.micropython.org/en/latest/library/machine.UART.html

Example: Transmit and Receive data via UART

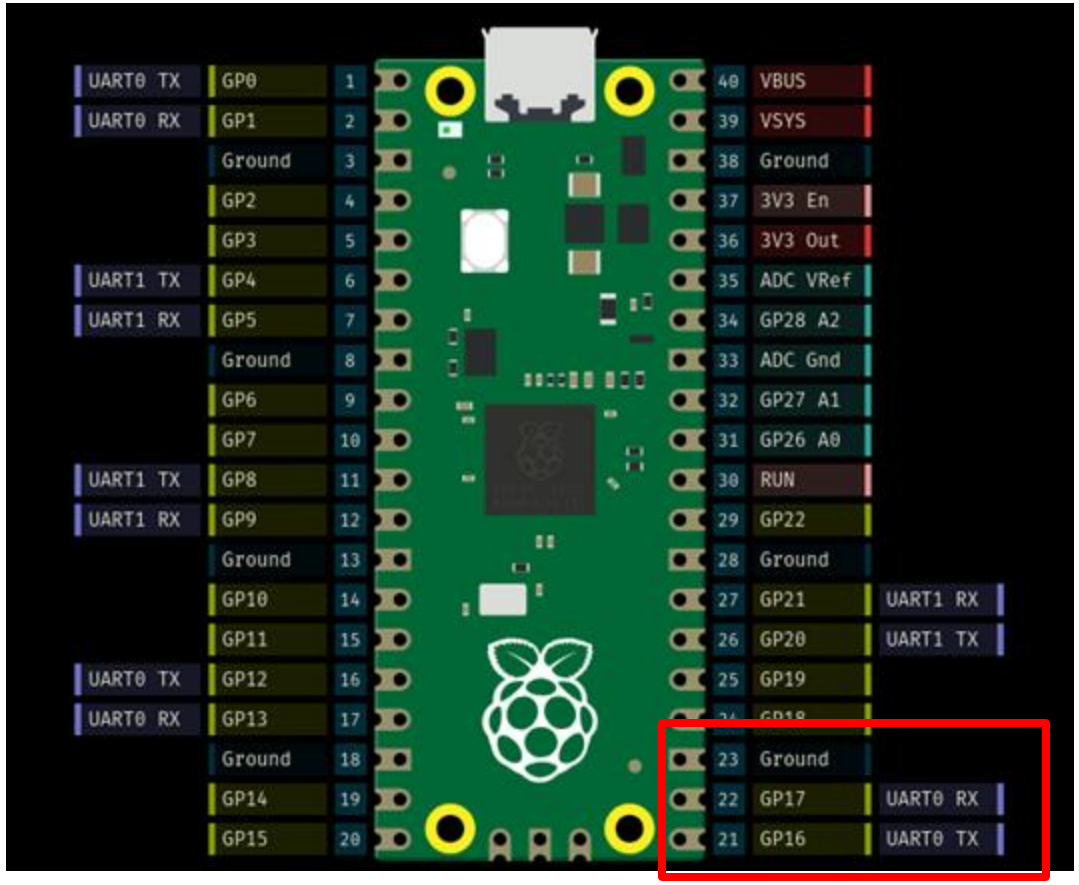
```
from machine import UART
import time

# Initialize
# and TX
uart =

# Send a me
message = "Hel
uart.write(messag
print("Sent:", messa

# Allow some time for
time.sleep(1)

# Check if th
if uart.an
    # Re
    respo
    print("R
```

```
# Deinitialize UART after communication is complete
uart.deinit()
```

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
ValueError: bad TX pin
```

TX≠17
TX=16

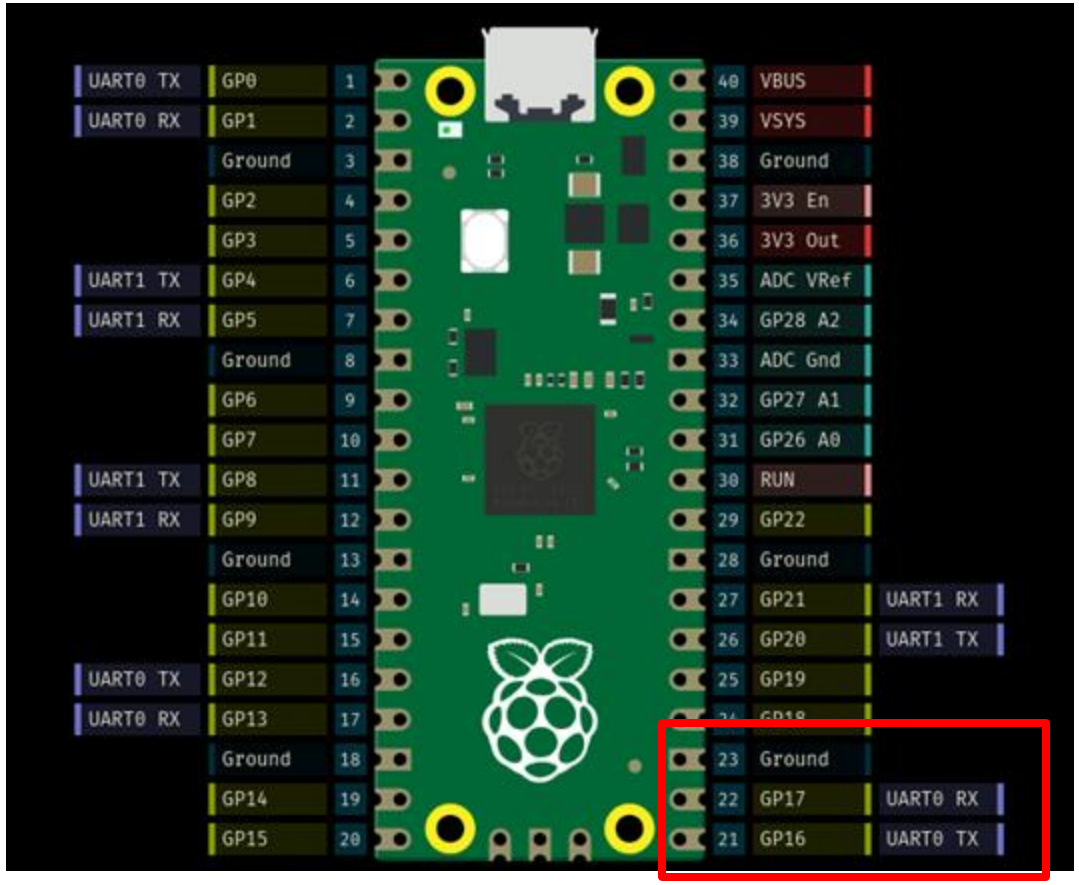
Debugging



Example: Transmit and Receive data via UART

```
from machine import UART
import time

# Initialize UART on UART0 with baudrate 9600,
# and TX on GPIO 16, RX on GPIO 17
uart = UART(0, baudrate=9600, tx=16, rx=17)

# Send a message
message = "Hello, World!"
uart.write(message)
print("Sent:", message)

# Allow some time for the device to respond
time.sleep(1)

# Check if there is data to read
if uart.any():
    # Read the response (up to 20 bytes)
    response = uart.read(20)
    print("Received:", response.decode('utf-8'))

# Deinitialize UART after communication is complete
uart.deinit()
```

Debugging

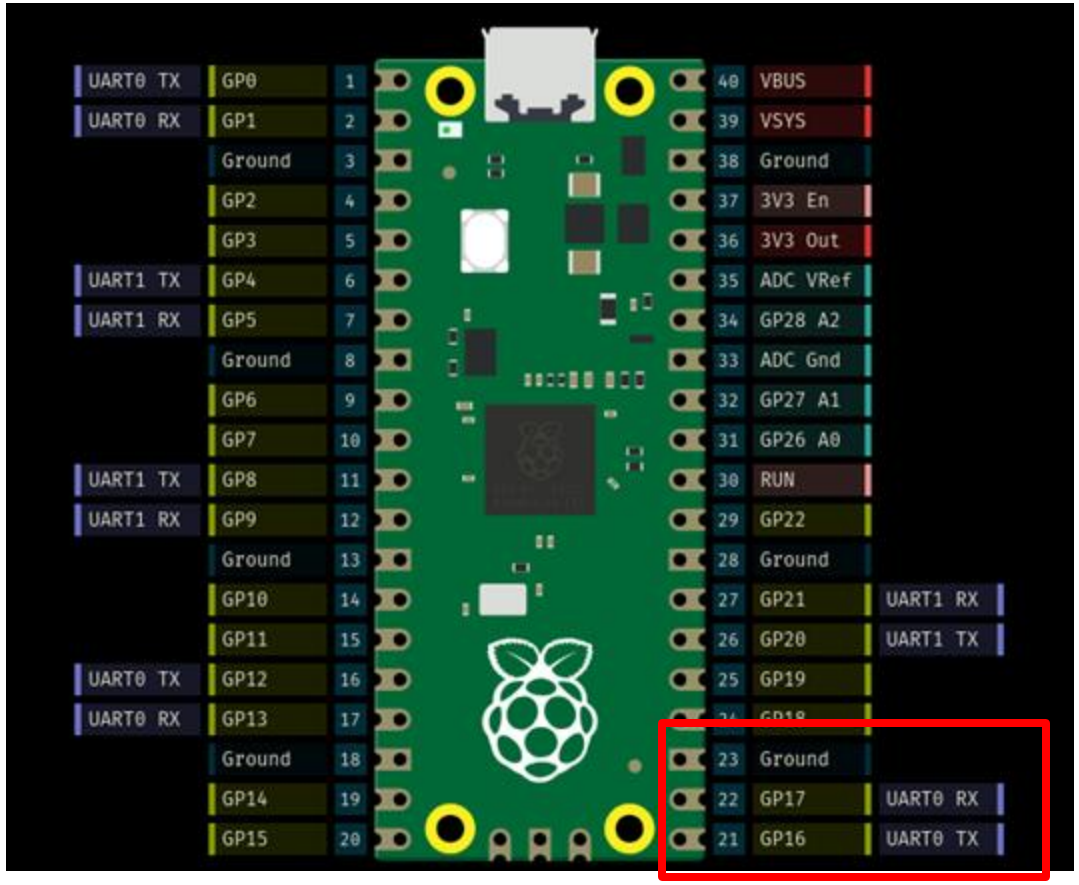


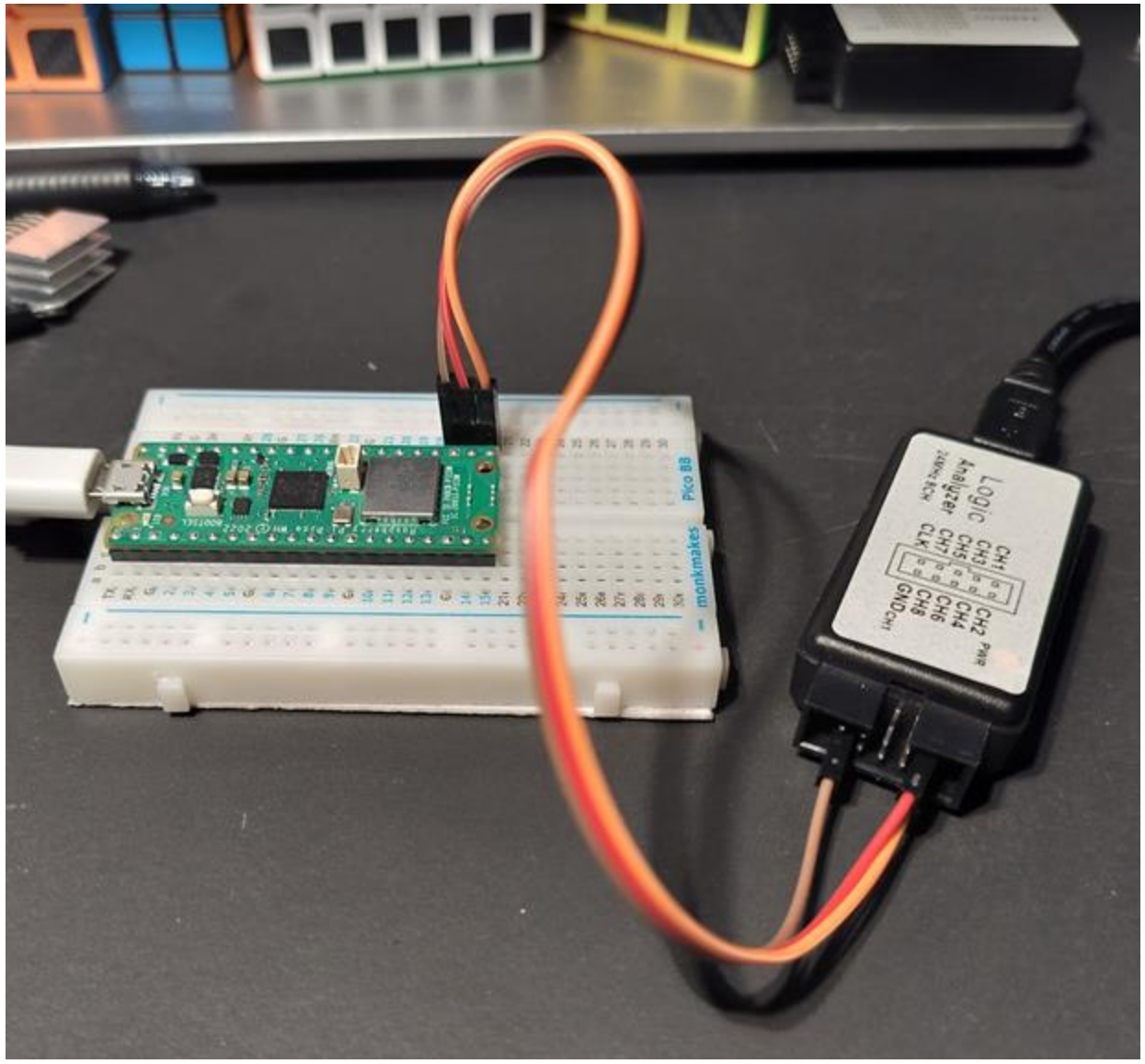
Example: Transmit and Receive data via UART

```
from machine import UART
import time

# Initialize UART on UART0 with baudrate 9
# and TX on GPIO 16, RX on GPIO 17
uart = UART(0, baudrate=9600, tx=16,

# Send a message
message = "Hello, World!"
uart.write(message)
p                    essage)

                                    id
ti

# Check
if uart.an
    # Read th                    ytes)
    response =
    print("Receiv                 decode('utf-8'))

# Deinitialize UART              munication is complete
uart.deinit()
```

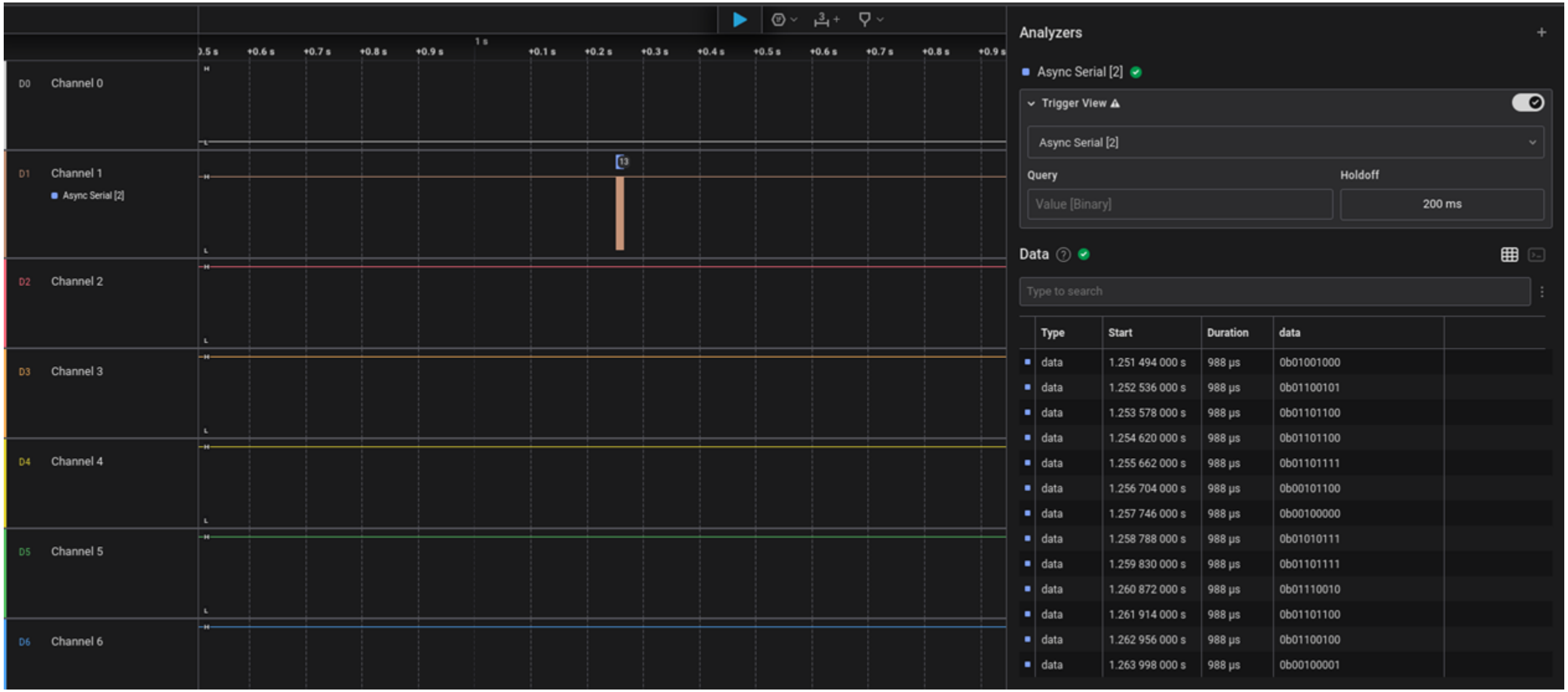
```
Sent: Hello, World!
```

Debugging

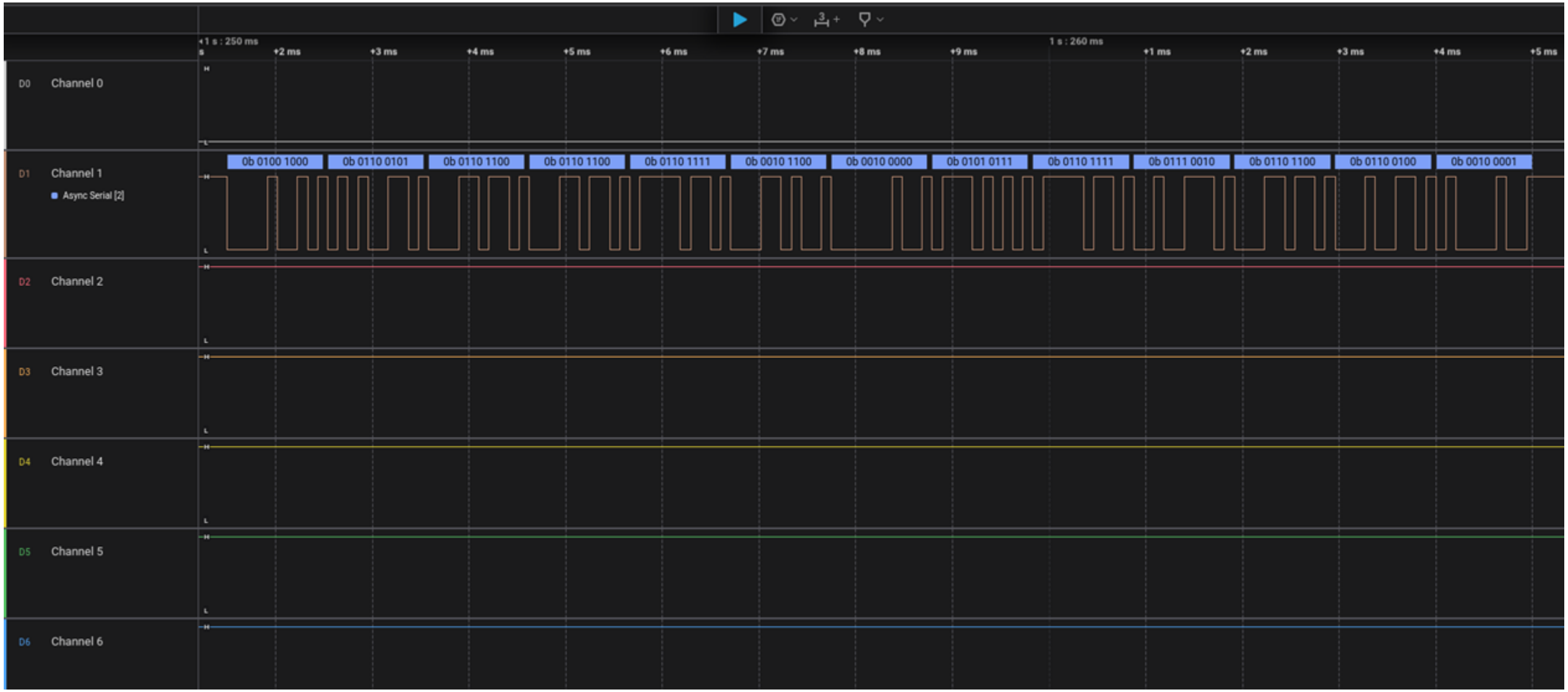

UART0 TX GP0 1
UART0 RX GP1 2
Ground 3
GP2 4
GP3 5
UART1 TX GP4 6
UART1 RX GP5 7
Ground 8
GP6 9
GP7 10
UART1 TX GP8 11
UART1 RX GP9 12
Ground 13
GP10 14
GP11 15
UART0 TX GP12 16
UART0 RX GP13 17
Ground 18
GP14 19
GP15 20
40 VBUS
39 VSYS
38 Ground
37 3V3 En
36 3V3 Out
35 ADC VRef
34 GP28 A2
33 ADC Gnd
32 GP27 A1
31 GP26 A0
30 RUN
29 GP22
28 Ground
27 GP21 UART1 RX
26 GP20 UART1 TX
25 GP19
23 Ground
22 GP17 UART0 RX
21 GP16 UART0 TX


Pico BB
monkmakes
LOGIC
Analyzer

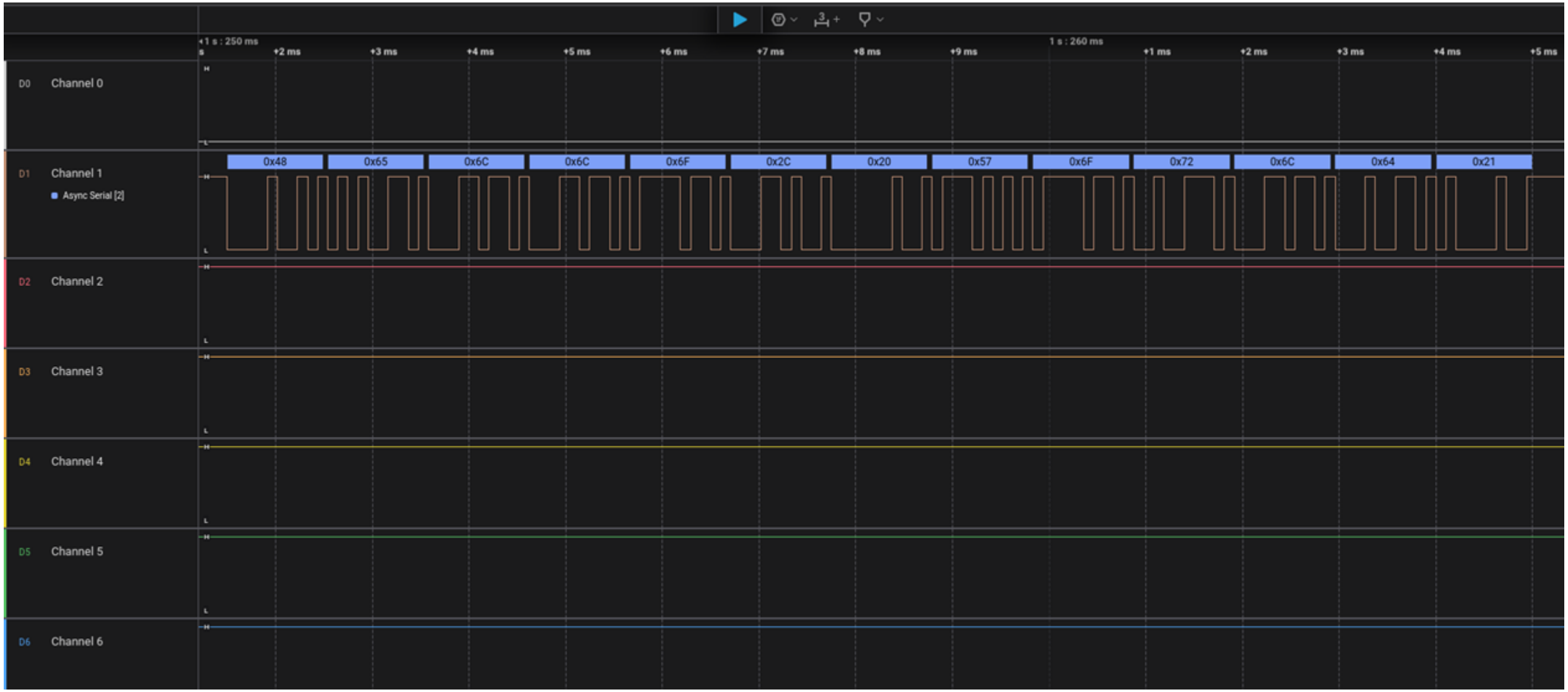
Debugging

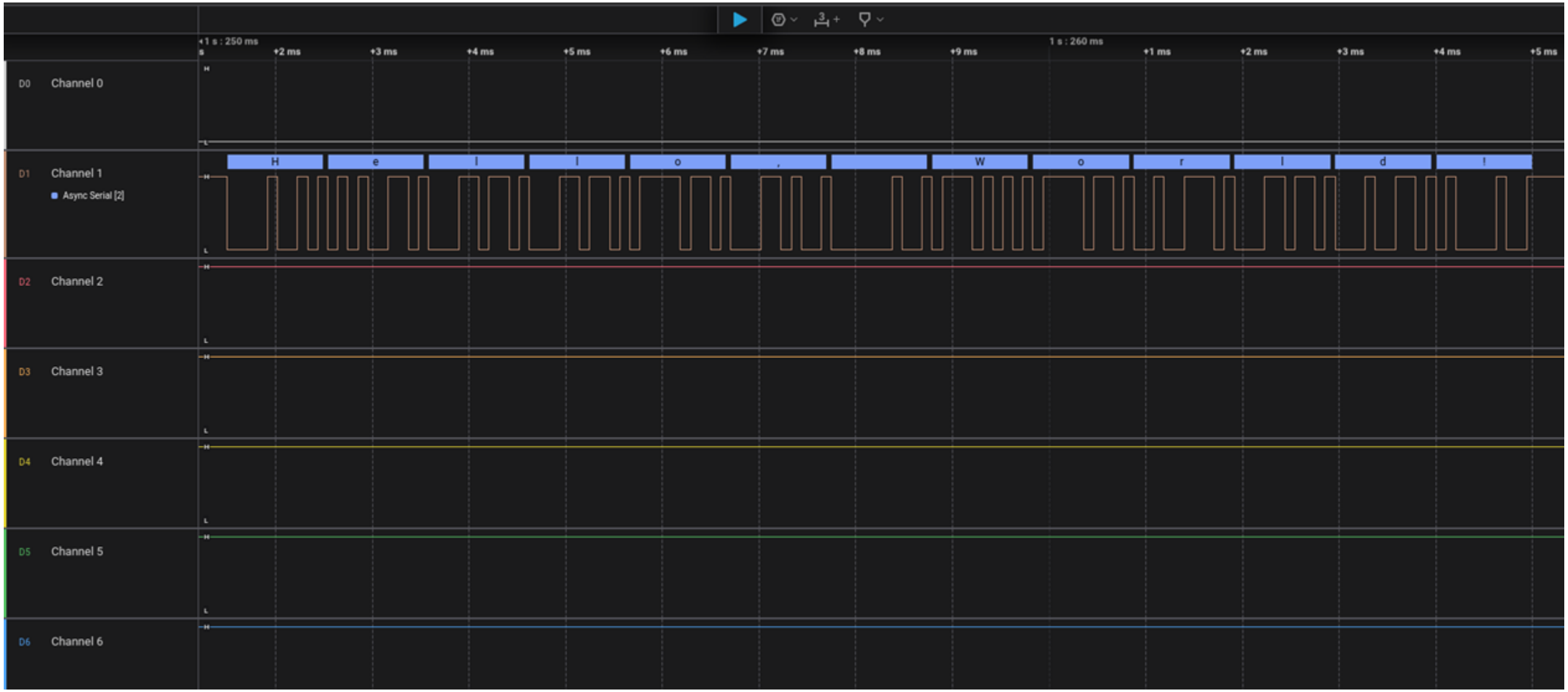

1 s
D0 Channel 0
D1 Channel 1
Async Serial [2]
D2 Channel 2
D3 Channel 3
D4 Channel 4
D5 Channel 5
D6 Channel 6
Analyzers
Async Serial [2]
Trigger View
Async Serial [2]
Query
Holdoff
Value [Binary]
200 ms
Data
Type to search
Type Start Duration data
data 1.251 494 000 s 988 µs 0b01001000
data 1.252 536 000 s 988 µs 0b01100101
data 1.253 578 000 s 988 µs 0b01101100
data 1.254 620 000 s 988 µs 0b01101100
data 1.255 662 000 s 988 µs 0b01101111
data 1.256 704 000 s 988 µs 0b00101100
data 1.257 746 000 s 988 µs 0b00100000
data 1.258 788 000 s 988 µs 0b01010111
data 1.259 830 000 s 988 µs 0b01101111
data 1.260 872 000 s 988 µs 0b01110010
data 1.261 914 000 s 988 µs 0b01101100
data 1.262 956 000 s 988 µs 0b01100100
data 1.263 998 000 s 988 µs 0b00100001

Debugging

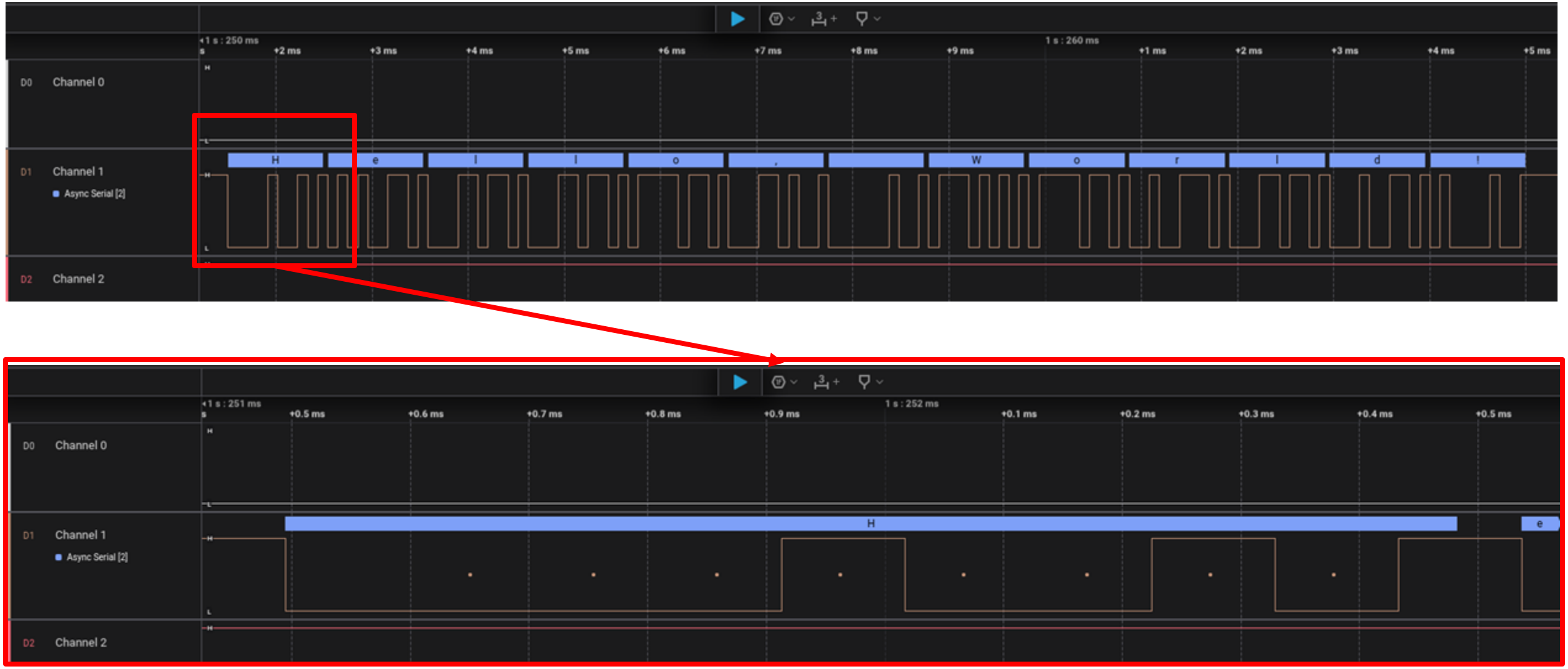

1 s : 250 ms
+2 ms
+3 ms
+4 ms
+5 ms
+6 ms
+7 ms
+8 ms
+9 ms
1 s : 260 ms
+1 ms
+2 ms
+3 ms
+4 ms
+5 ms
D0 Channel 0
D1 Channel 1
Async Serial [2]
0b 0100 1000
0b 0110 0101
0b 0110 1100
0b 0110 1100
0b 0110 1111
0b 0010 1100
0b 0010 0000
0b 0101 0111
0b 0110 1111
0b 0111 0010
0b 0110 1100
0b 0110 0100
0b 0010 0001
D2 Channel 2
D3 Channel 3
D4 Channel 4
D5 Channel 5
D6 Channel 6

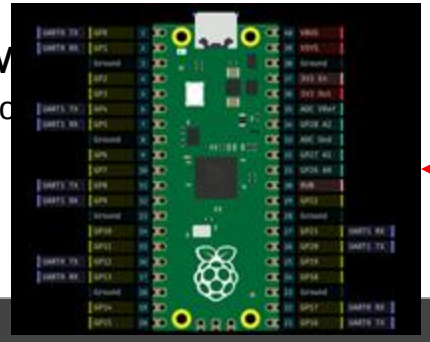
Debugging

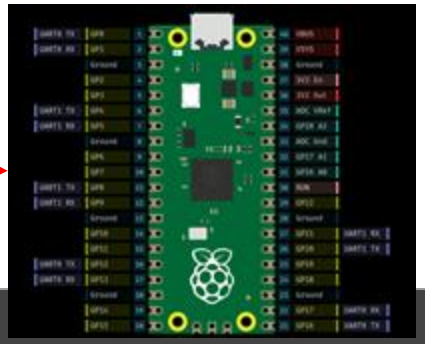

1 s : 250 ms
+2 ms
+3 ms
+4 ms
+5 ms
+6 ms
+7 ms
+8 ms
+9 ms
1 s : 260 ms
+1 ms
+2 ms
+3 ms
+4 ms
+5 ms
D0 Channel 0
D1 Channel 1
Async Serial [2]
0x48
0x65
0x6C
0x6C
0x6F
0x2C
0x20
0x57
0x6F
0x72
0x6C
0x64
0x21
D2 Channel 2
D3 Channel 3
D4 Channel 4
D5 Channel 5
D6 Channel 6


+1 s : 250 ms
+2 ms
+3 ms
+4 ms
+5 ms
+6 ms
+7 ms
+8 ms
+9 ms
1 s : 260 ms
+1 ms
+2 ms
+3 ms
+4 ms
+5 ms
D0 Channel 0
D1 Channel 1
Async Serial [2]
H e l l o , W o r l d !
D2 Channel 2
D3 Channel 3
D4 Channel 4
D5 Channel 5
D6 Channel 6

Debugging


1 s : 250 ms
+2 ms
+3 ms
+4 ms
+5 ms
+6 ms
+7 ms
+8 ms
+9 ms
1 s : 260 ms
+1 ms
+2 ms
+3 ms
+4 ms
+5 ms
D0 Channel 0
D1 Channel 1
Async Serial [2]
H e l l o , W o r l d !
D2 Channel 2
1 s : 251 ms
+0.5 ms
+0.6 ms
+0.7 ms
+0.8 ms
+0.9 ms
1 s : 252 ms
+0.1 ms
+0.2 ms
+0.3 ms
+0.4 ms
+0.5 ms
D0 Channel 0
D1 Channel 1
Async Serial [2]
H e
D2 Channel 2



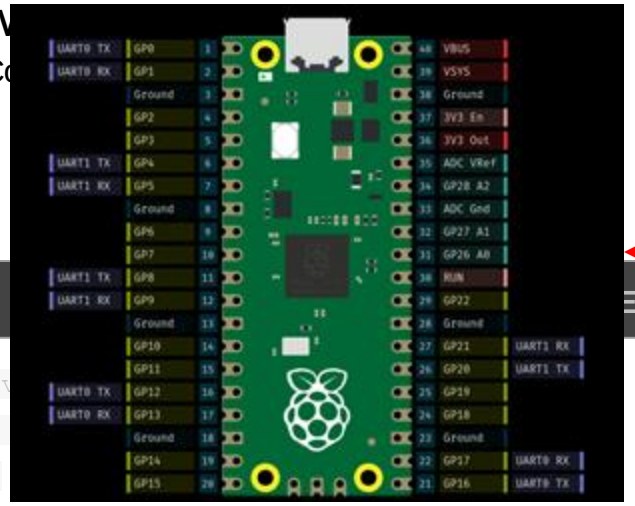


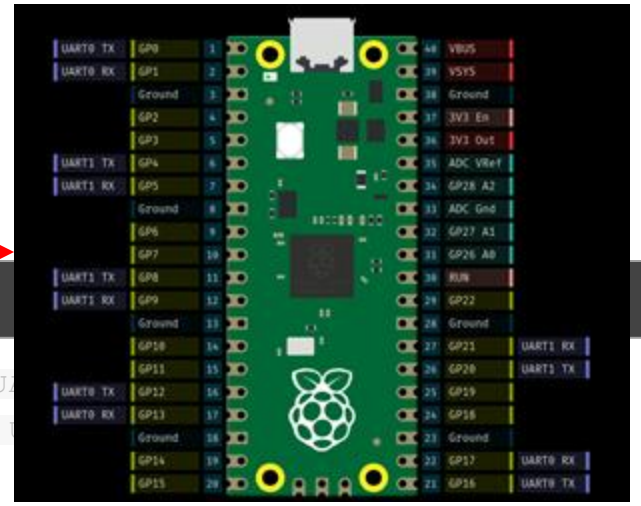
Example: Transmit and Receive data via UART

```
# Send data via UART
from machine import UART, Pin
import utime

led = Pin

# Initiali
uart = UAR                                                    on GPIO 1,
RX on GPIO

while True
    message
    uart.wr
    led.tog

    print(

    utime.s                                                   ode and
```

```
# Receive data via UART
from machine import UART, Pin
import utime

            led.toggle()

    utime.sleep(0.1)  # Small delay to reduce CPU load
```

Demo (using Thonny)





Example: Transmit and Receive data via UART

Task: Make a simple chat program between two Pi Picos via UART

(30 min)

Portfolio 4: Controller for a Mobile Robot
(Open-ended Mini Project)

and

Extra Credit Activities 4: Present your Mini Project

Portfolio 4: Controller for a Mobile Robot (Open-ended Mini Project)

- **Deadline:** 1/12, 23:59
- **Hand-in:**
  - 1-minute and 30-second video showcasing your working prototype
  - Source code (your own git)
- **Requirement (for the mine project)**
  - Movement commands (forward/backward/left/right)
  - Speed adjustment
  - User Interface (control input) with visual speed indicator
  - Safety and Emergency Stop Functionality

Extra Credit Activities 4: Present your Mini Project

- **Deadline:** 2/12, in-class
- **Hand-in:**
  - Take an active part in the presentation of your Mini Project for the rest of the class
  - Point(s): 2 points