ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν
Πανεπιστήμιον Αθηνών
ΙΔΡΥΘΕΝ ΤΟ 1837

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ**

# ΔΠΜΣ Space Technologies, Applications and seRvices (STAR)

## M806 Space Data Systems

## ΣΕΙΡΙΑΚΑ ΠΡΩΤΟΚΟΛΛΑ

*Ακαδημαϊκό Έτος 2023-2024*

*Νεκτάριος Κρανίτης*

# Serial Communication

- Native word size is multi-bit (8, 16, 32, 64, etc.)
- Often, it's not feasible to support sending all the word's bits at the same time
- Serial communication
  - Transmits data **one bit at a time,** in a sequential fashion
  - In contrast to parallel comm. where multiple bits are sent as a whole
- Commonly used for long-haul communication, modems and non-networked communication between devices
- Examples: UART, I2C, CAN, USB, Ethernet, PCI Express etc.


Serial communication

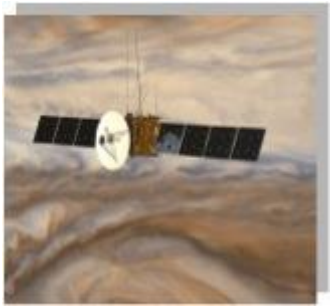
Parallel communication

# Serial vs. Parallel Communication

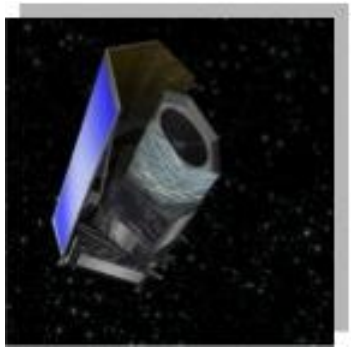
- Cost and weight
  - Less cost and weight, as less wires and smaller connectors are needed compared with parallel communication
- Better reliability
  - Parallel communications may introduce more clock skews, as well as crosstalk between different wires
- Higher clock rate
  - Due to the higher reliability, serial communication can be clocked in a higher frequency, hence increase the throughput
- On the other hand, the conversion between serial and parallel data may consume extra overheads

# Example: SpaceWire Interface for Space Systems

- SpaceWire (SpW) protocol is a standard for "high"-speed links and networks for use onboard spacecraft, easing interconnection of:
  - sensors
  - mass-memories
  - processing units, and
  - downlink telemetry sub-systems
- SpW is being widely used on many space missions by: ESA, NASA, JAXA, CNSA
- SpW equipment is connected together using SpW links which are:
  - serial
  - high-speed (2 Mbits/sec to 200 Mbits/sec)
  - bi-directional
  - full-duplex



**Juice**: Launch 2022



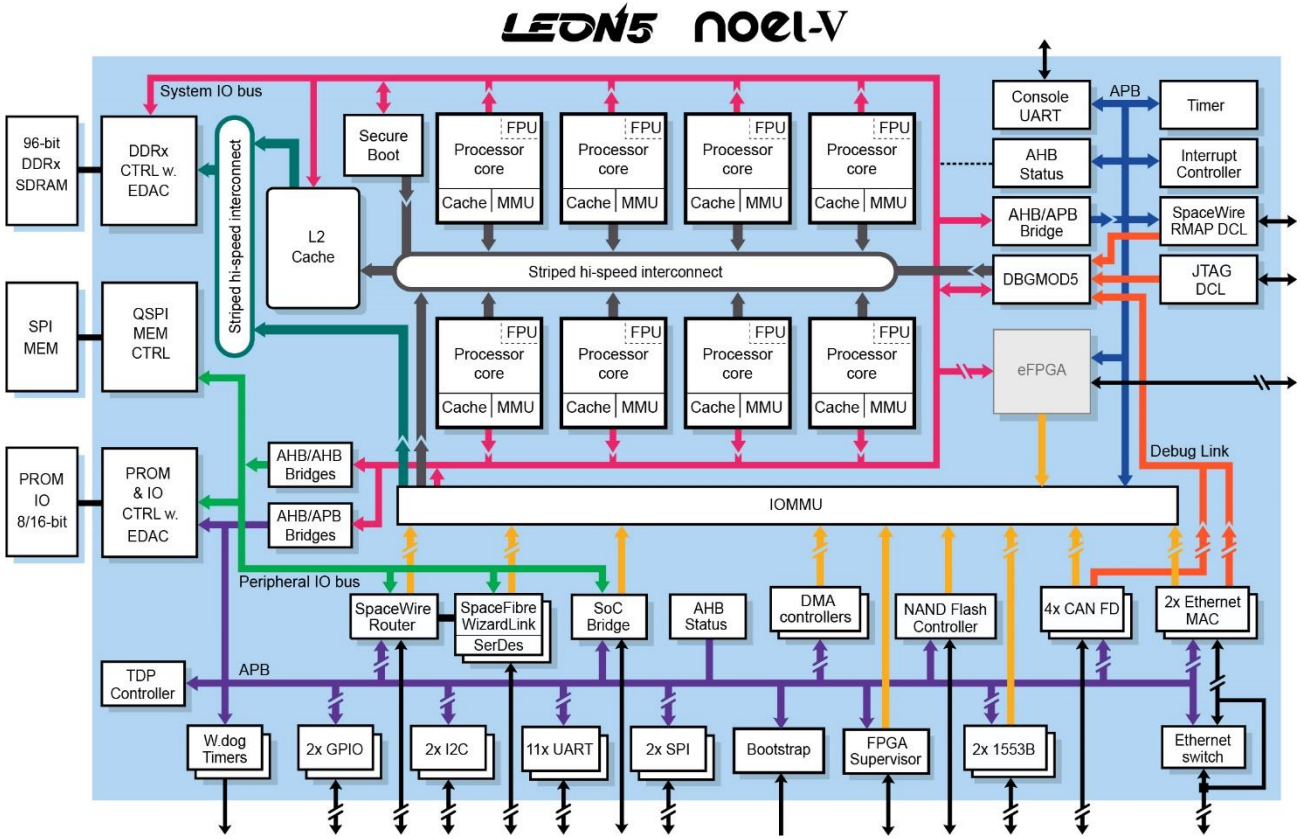
**Euclid**: Launch 2022

**Proba-3**: Launch 2023

# GR765 Octa-Core Processor

**Features**

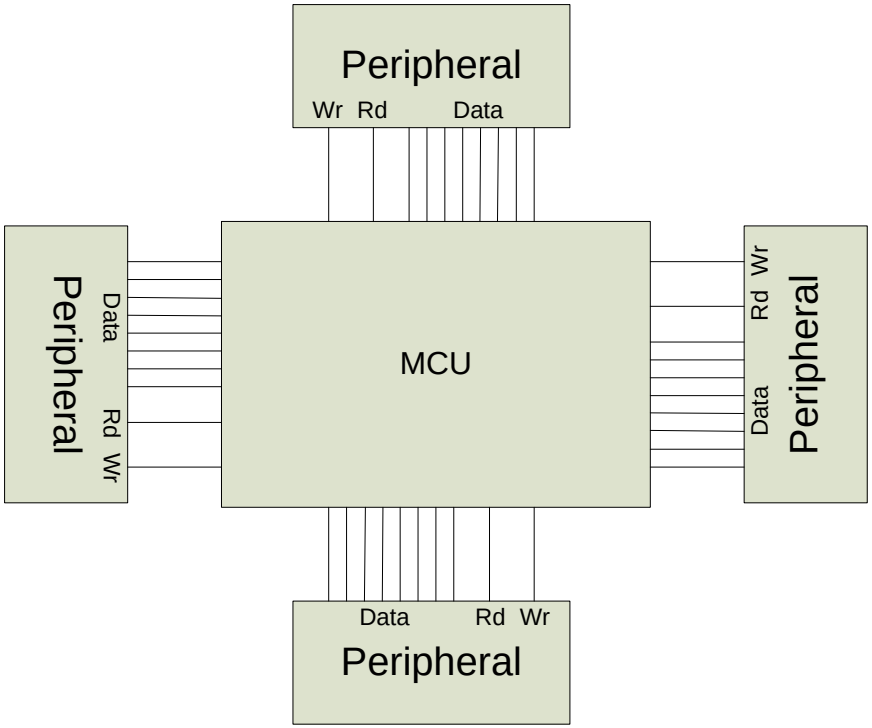
- 8x Fault-tolerant LEON5FT or NOEL-V FT with dedicated FPU and MMU
- 32 KiB per core L1 cache, connected to advanced interconnect providing multiple-parallel paths between processors and Level-2 Cache
- 2+ MiB L2 cache, 512-bit cache line, 4-ways
- DDR2/3/4 SDRAM
- DMA controllers
- SpaceFibre x4+ lanes 6.25 Gbit/s, simpler protocols (Wizard link)
- 12-port SpaceWire router with +4 internal ports
- 3x 10/100/1000 Mbit Ethernet with TT Ethernet support. TSN support TBD.
- 2x MIL-STD-1553B
- 4x CAN FD
- 12 x UART
- 2x SPI master/slave, GPIO, Timers & Watchdog
- 2x I2C interface
- NAND Flash controller interface
- FPGA Supervisor interface
- SoC Bridge interface for efficient connection to companion FPGAs
- High-pin count – LGA1752
- Debug links: Ethernet, JTAG, SpaceWire, CAN



# Types of Serial Communication

- Synchronous Serial Transmission
  - A common clock is shared by both the sender and the receiver
  - More efficient transmission, since one wire is dedicatedly used for data transferring
  - More costly since an extra clock wire is required
- Asynchronous Serial Transmission
  - The sender does not have to send a clock signal
  - Both sender and receiver agree on timing parameters in advance
  - Special bits are added to synchronize transmission

# Example System

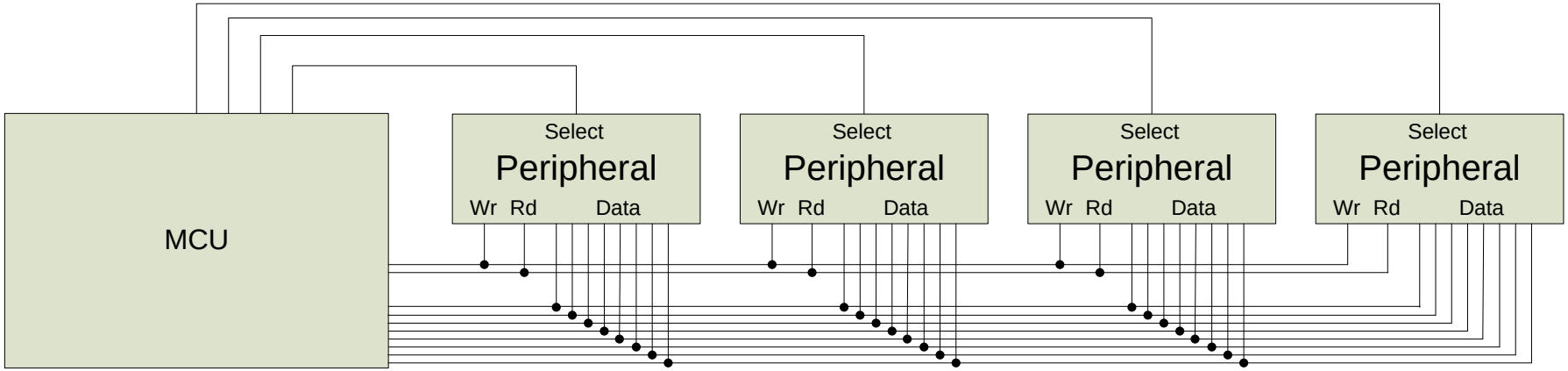



- Dedicated point-to-point connections
  - Parallel data lines, read and write lines between MCU and each peripheral

✓ Fast, allows simultaneous transfers

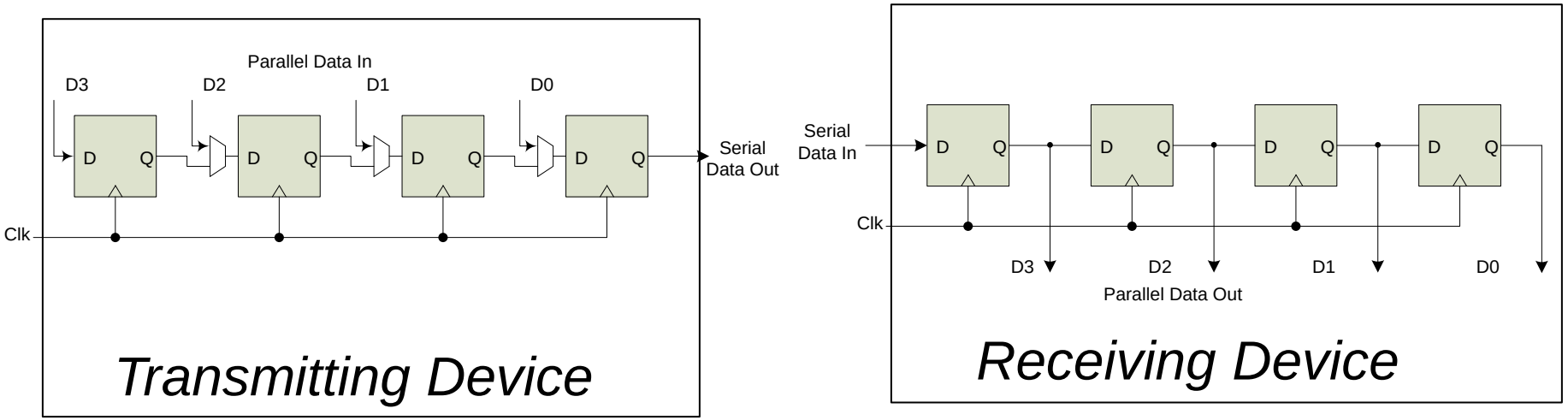
✘ Requires many connections, PCB area, scales badly

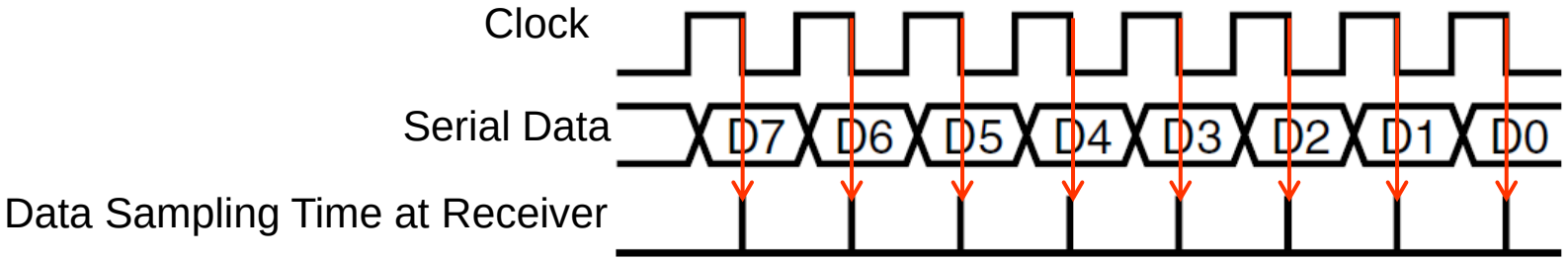
# Parallel Buses




- All devices use buses to share data, read and write signals
- MCU uses individual select lines to address each peripheral
- MCU requires fewer pins for data, but still one per data bit
- MCU can communicate with only one peripheral at a time

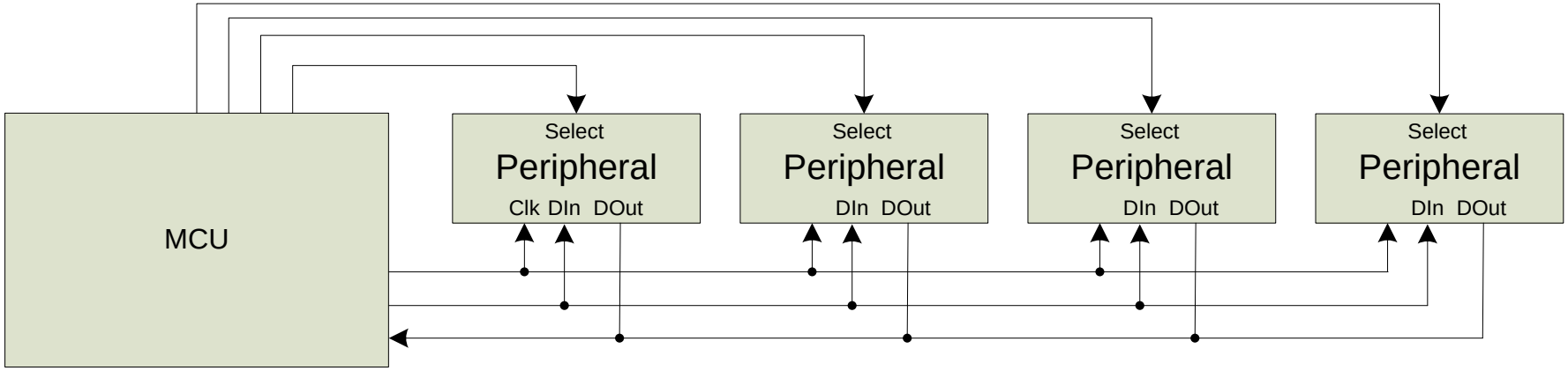
# Synchronous Serial Data Transmission



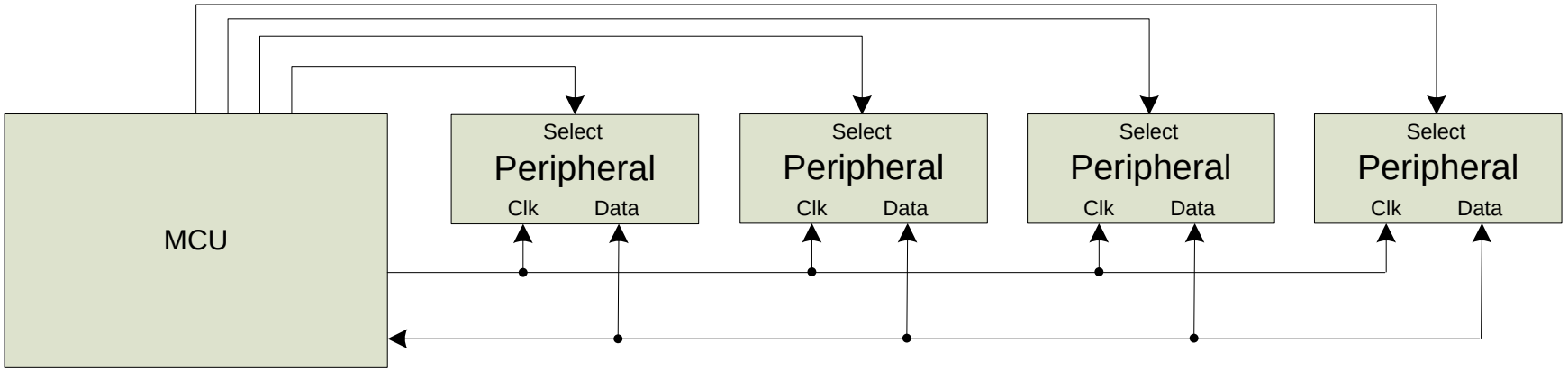

- Use **shift registers** and a clock signal to convert between serial and parallel formats
- **Synchronous**: an explicit clock signal is along with the data signal

# Synchronous Full-Duplex Serial Data Bus



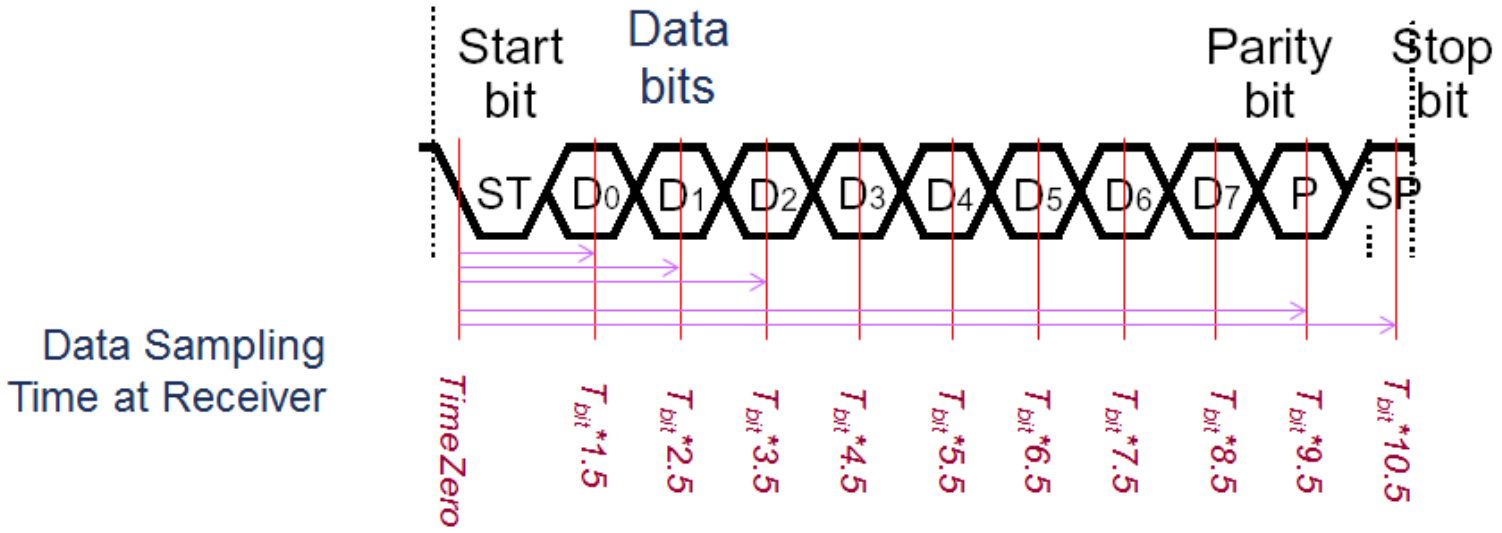

- Use two serial data lines - one for reading, one for writing
  - Allows simultaneous send and receive ***full-duplex communication***

# Synchronous Half-Duplex Serial Data Bus




- Share the serial data line
- Doesn't allow simultaneous send and receive – is ***half-duplex*** *communication*

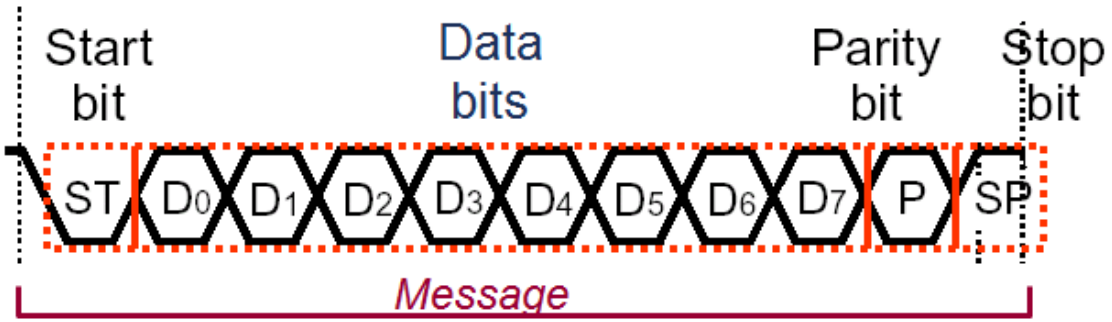
# Asynchronous Serial Communication




- **Eliminate the clock line**!
- Transmitter and receiver must generate clock **locally**
- Transmitter must add **start bit** (always same value) to indicate start of each data frame
- Receiver detects leading edge of start bit, then uses it as a timing reference for sampling data line to extract each data bit N at time Tbit*(N+5)
- **Stop bit** is also used to detect some timing errors

# Serial Communication Specifics

- Data frame fields
  - **Start bit** (one bit)
  - **Data** (LSB first or MSB, and size – 7, 8, 9 bits)
  - Optional **parity bit** is used to make total number of ones in data even or odd
  - **Stop bit** (one or two bits)



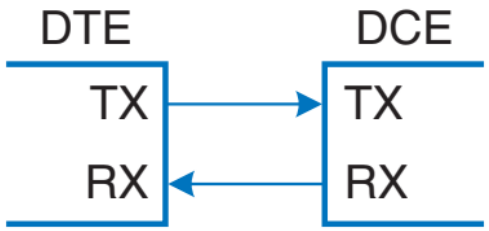

- All devices must use the same communications parameters
  - E.g. comm. speed (300 baud, 600, 1200, 2400, 9600, 14400, 19200, etc.)
- Sophisticated network protocols have more information in each data frame
  - Medium access control – when multiple nodes are on bus, they must arbitrate for permission to transmit
  - Addressing information – for which node is this message intended?
  - Larger data payload
  - Stronger error detection or error correction information
  - Request for immediate response ("in-frame")

# Error Detection

- Can send additional information to verify data was received correctly
- Need to specify which parity to expect: even, odd or none.
- **Parity bit** is set so that total number of "1" bits in data and parity is even (for **even parity**) or odd (for **odd parity**)
  - 01110111 has 6 "1" bits, so parity bit will be 1 for odd parity, 0 for even parity
  - 01100111 has 5 "1" bits, so parity bit will be 0 for odd parity, 1 for even parity
- Single parity bit detects if 1, 3, 5, 7 or 9 bits are corrupted, but doesn't detect an even number of corrupted bits
- Stronger error detection codes (e.g. Cyclic Redundancy Check, CRC) exist and use multiple bits (e.g. 8, 16), and can detect many more corruptions
  - Used for CAN, USB, Ethernet, Bluetooth, etc.

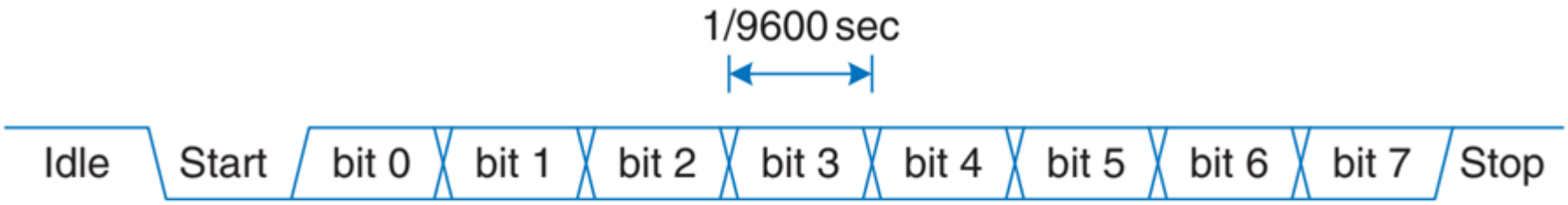
# Universal Asynchronous Receiver Transmitter (UART)

- UART (pronounced "*you-art*") is a serial I/O peripheral that communicates between two systems **without sending a clock**
- Instead, the systems must agree in advance about what data rate to use and must each locally generate its own clock
  - Transmission is **asynchronous** because the clocks are not synchronized
  - Although these system clocks may have a small frequency error and an unknown phase relationship, the UART manages reliable asynchronous communication
- UARTs are used in protocols such as RS-232 and RS-485
  - E.g. old computer serial ports use RS-232C standard (1969 by Electronics Industries Associations)
  - The standard originally envisioned connecting Data Terminal Equipment (DTE) such as a mainframe computer to Data Communication Equipment (DCE) such as a modem
- Although UART is relatively slow and prone to misconfiguration issues, standards have been around for so long that they remain important today




DTE sends data to DCE over the TX line and receives data back over the RX line

# UART Asynchronous Serial Link



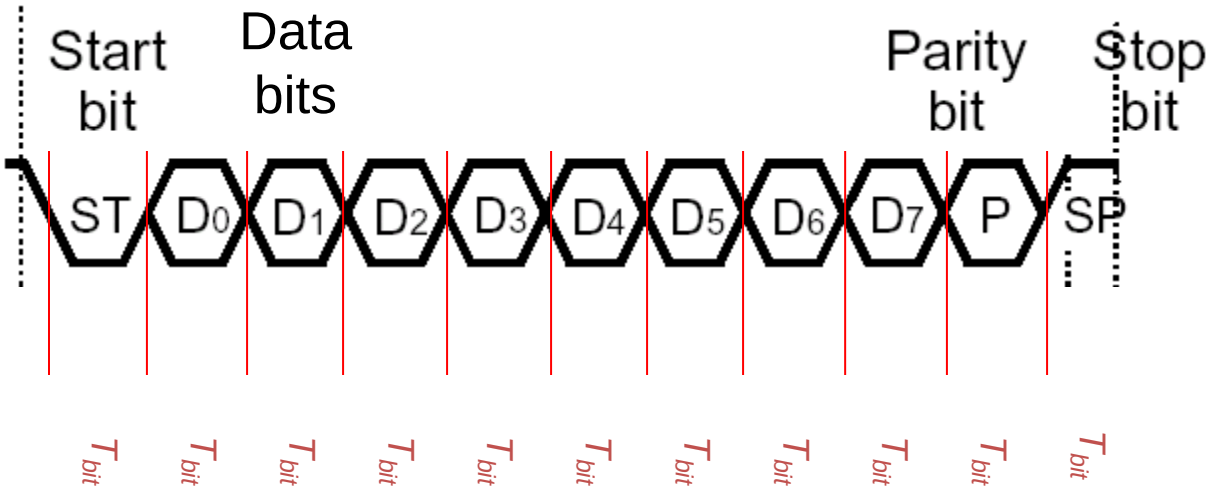

Sending a character at a data rate of 9600 baud

- Line idles at a logic ‘1’ when not in use
  - Why do you think?
- Each character is sent as a **start** bit (0), 7 or 8 **data** bits, an optional **parity** bit, and one or more **stop** bits (1’s)
- UART detects falling transition from Idle to Start to lock on to the transmission at the appropriate time

# UART Baud rate

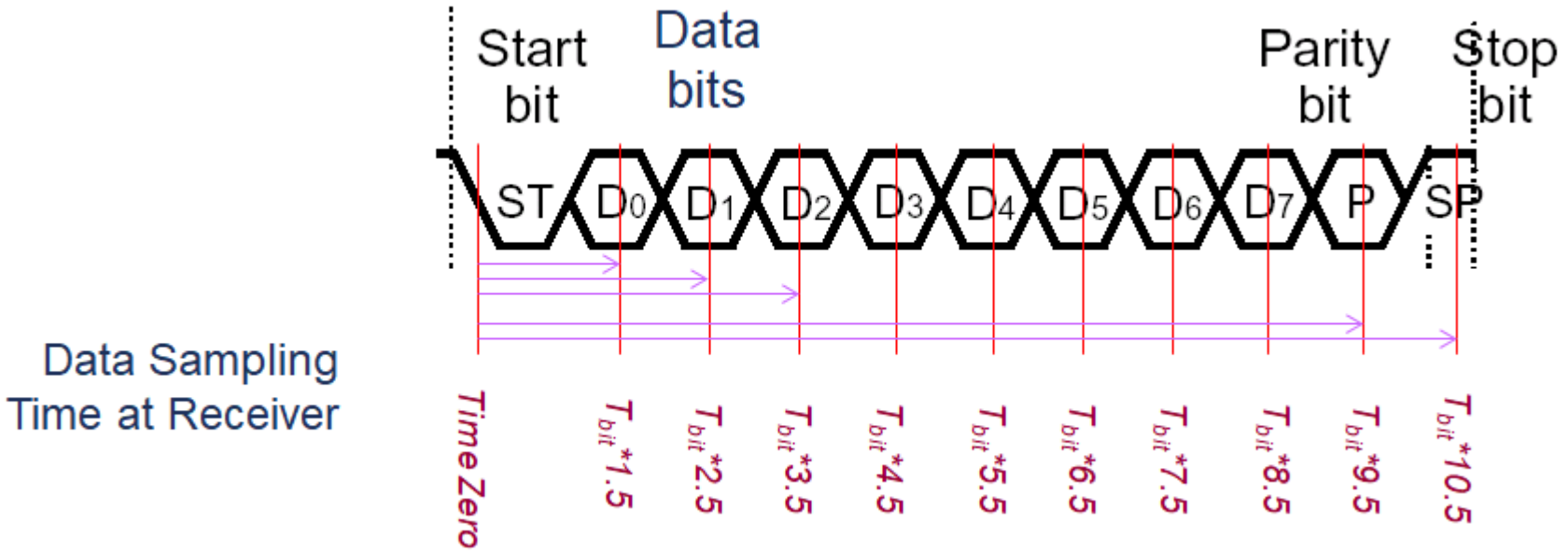
- Although 7 data bits is sufficient to send an ASCII character, 8 bits are normally used because they can convey an arbitrary byte of data
- Optional parity bit allows detection if a bit was corrupted during transmission
- A common choice is 8 data bits, no parity, and 1 stop bit, making a total of 10 symbols to convey an 8-bit character of information
- Signaling rates are referred to in units of baud rather than bits/sec
  - a baud rate of 9600 indicates 9600 symbols/sec, or 960 characters/sec
  - a baud rate of 9600 has a bit rate of (9600 symbols/second)×(8 bits/10 symbols)=7680 bits/second
- Both systems must be configured for the appropriate baud rate and number of data, parity, and stop bits or the data will be garbled
- Typical baud rates: 1200, 2400, 9600, 14400, 19200, 38400, 57600,115200,230400
- The lower rates were used in the 1970's and 1980's for modems that sent data over the phone lines as a series of tones
- In contemporary systems, 9600 and 115200 are two common baud rates
  - 9600 is encountered where speed doesn't matter
  - 115200 is the fastest standard rate, though slow compared to other modern serial I/O standards

# UART Transmitter




- If no data to send, keep sending 1 (stop bit) – ***idle line***
- When there is a data word to send:
  - Send a 0 (start bit) to indicate the start of a word
  - Send each data bit in word (use shift register for *transmit buffer*)
  - Send a 1 (stop bit) to indicate the end of the word

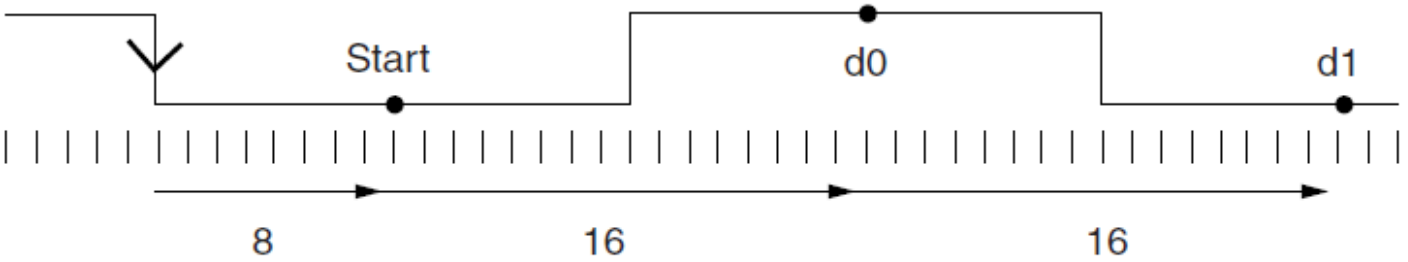
# UART Receiver




- Wait for a falling edge (beginning of a Start bit)
  - Then wait ½ bit time
  - Do the following for as many data bits in the word
    - Wait 1 bit time
    - Read the data bit and shift it into a receive buffer (shift register)
- Wait 1 bit time
- Read the bit
  - if 1 (Stop bit), then OK
  - if 0, there's a problem!

# UART Receiver Oversampling

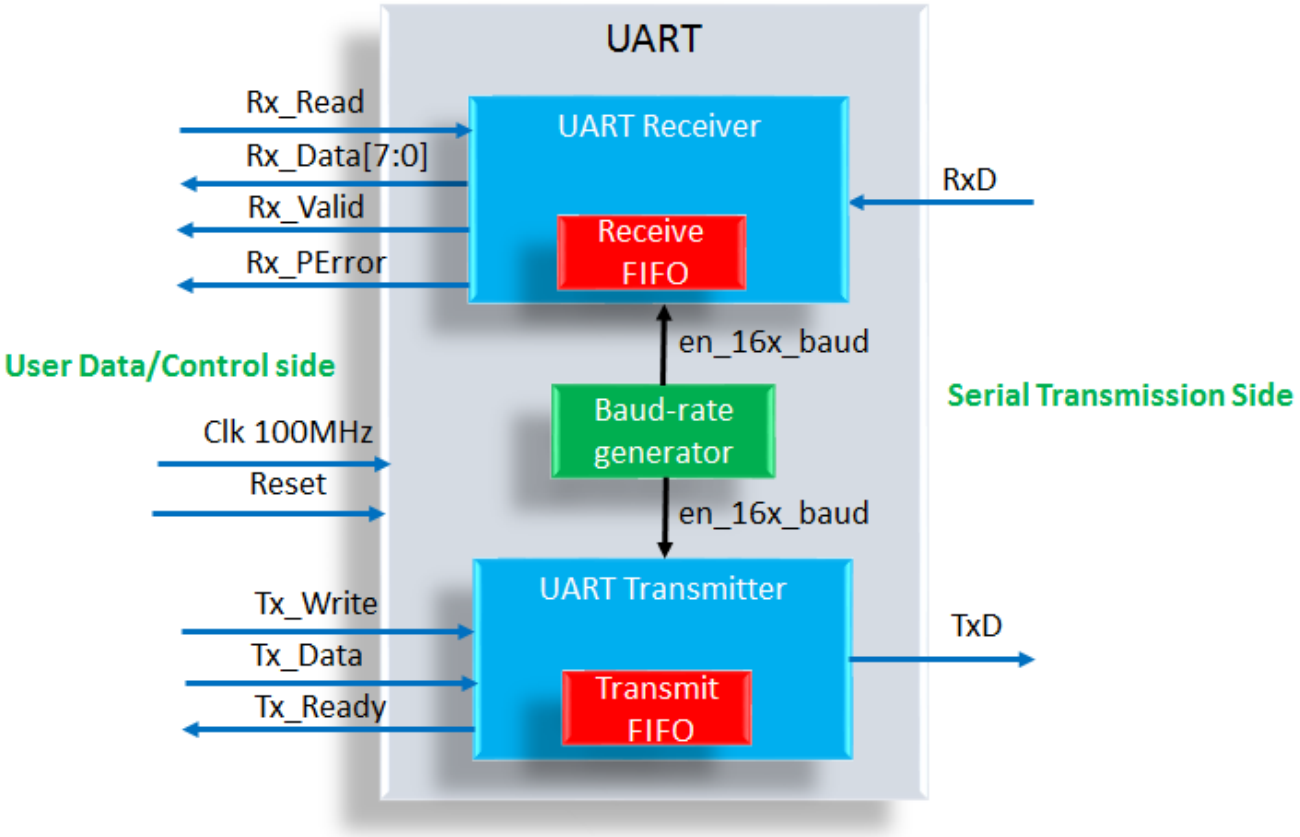
- When receiving, UART **oversamples** incoming data line
  - Extra samples allow **voting**, improving noise immunity
  - Better synchronization to incoming data, improving noise immunity
  - Common oversampling rate: 8 or 16 times the baud rate clock
  - Two voting methods: a) single sample in the center or
    b) majority vote of the three samples in the center
- Example:
  - Serial input data period 1/BaudRate
  - Receiver uses clock falling-edge to start mod-16 internal counter
  - When internal counter points to Start bit center, resets and repeats pointing to 1st data bit (d0) written in Serial-Input Parallel-Output (SIPO) register
  - Repeat for all data bits, (parity) and stop bits



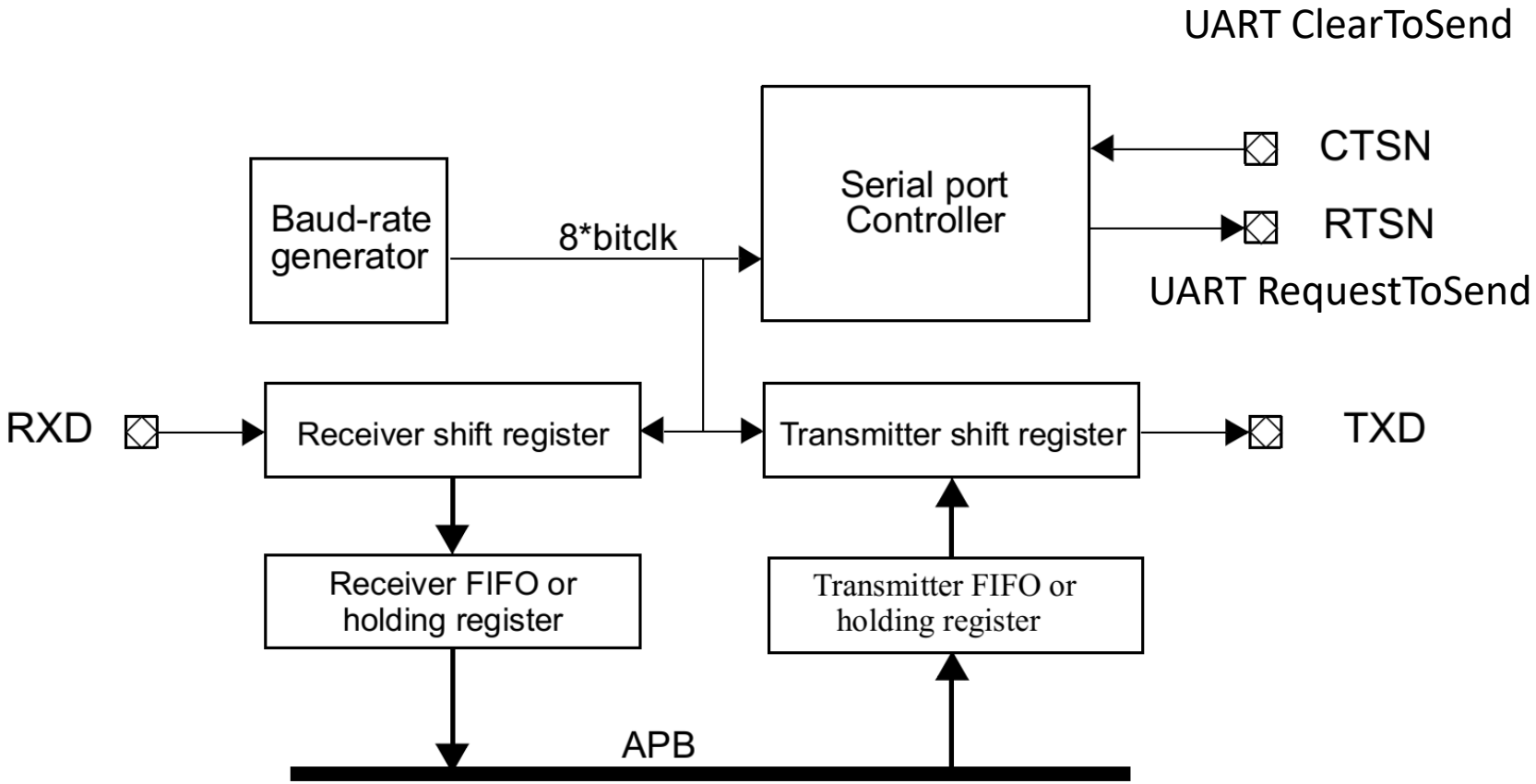
# UART Baud rate generator

- Consider a UART that has an internal system clock that must be divided down to produce a clock that is 16x the desired baud rate
- The appropriate divisor, BRD, is BRD = sys_clock/(16×baud rate)
- Baud rates do not all evenly divide system clock, so some divisors produce a frequency error
  - UART accommodates this error so long as it is small enough..
- Example:
  - Baud rate required: 38400
  - System clock: 100MHz
  - BRD: 100,000,000/(16*38400)=162.76≈163
  - Error 0.145%, acceptable if <1%
  - Baud-rate generator: pulse generator producing an enable pulse (**en_x16_baud**) used for oversampling every 163 clock cycles
-

# Example UART Architecture



## 18 APBUART - AMBA APB UART Serial Interface

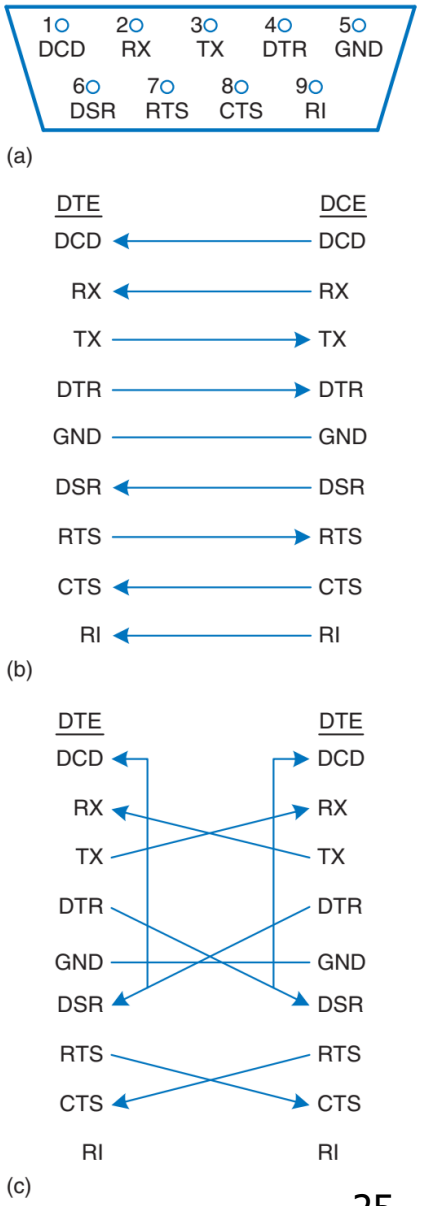


# RS-232 Standard

- RS-232 standard defines several additional signals
- Request-to-Send (RTS) and Clear-to-Send (CTS) signals can be used for hardware handshaking
- They can be operated in either of two modes:
  - **Flow control** mode: DTE clears RTS to 0 when it is ready to accept data from DCE. Likewise, DCE clears CTS to 0 when it is ready to receive data from DTE
  - (older) **simplex mode**, DTE clears RTS to 0 when it is ready to transmit. DCE replies by clearing CTS when it is ready to receive the transmission

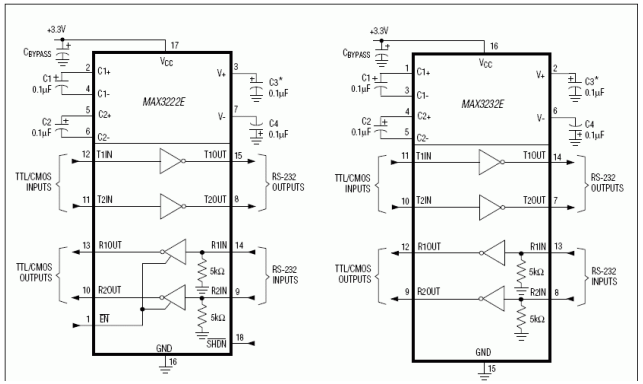
# RS-232 pinout and wiring

- Original standard recommended a massive 25-pin DB-25 connector, but PCs streamlined to a male 9-pin DE-9 connector with the pinout shown in figure (a)
- Cable wires normally connect straight across as shown in figure (b)
- However, when directly connecting two DTEs, a null modem cable shown in figure (c) may be needed to swap RX and TX and complete the handshaking
- As a final insult, some connectors are male and some are female
- In summary, it can take a large box of cables and a certain amount of guess-work to connect two systems over RS-232, again explaining the shift to USB
- Fortunately, embedded systems typically use a simplified 3- or 5-wire setup consisting of GND, TX, RX, and possibly RTS and CTS

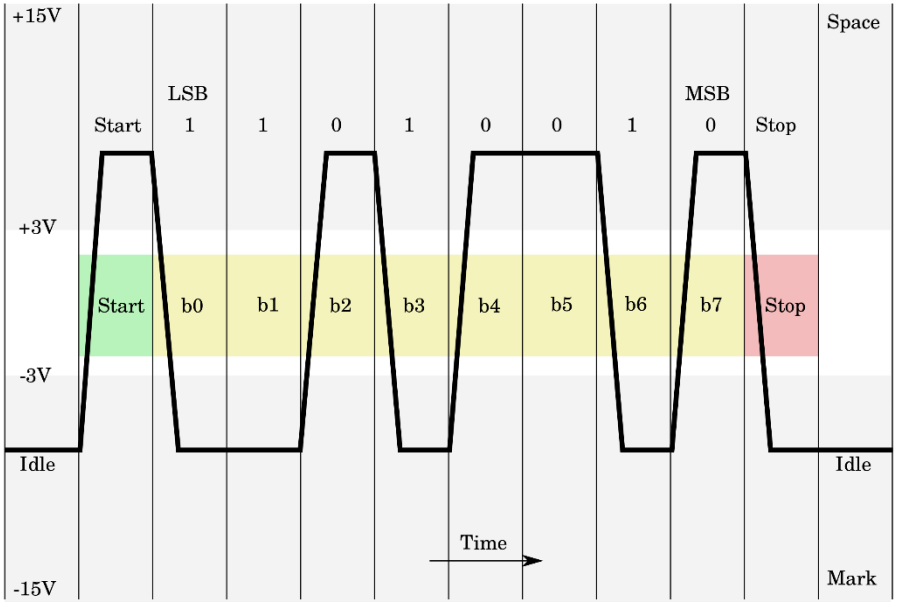


# RS-232 levels

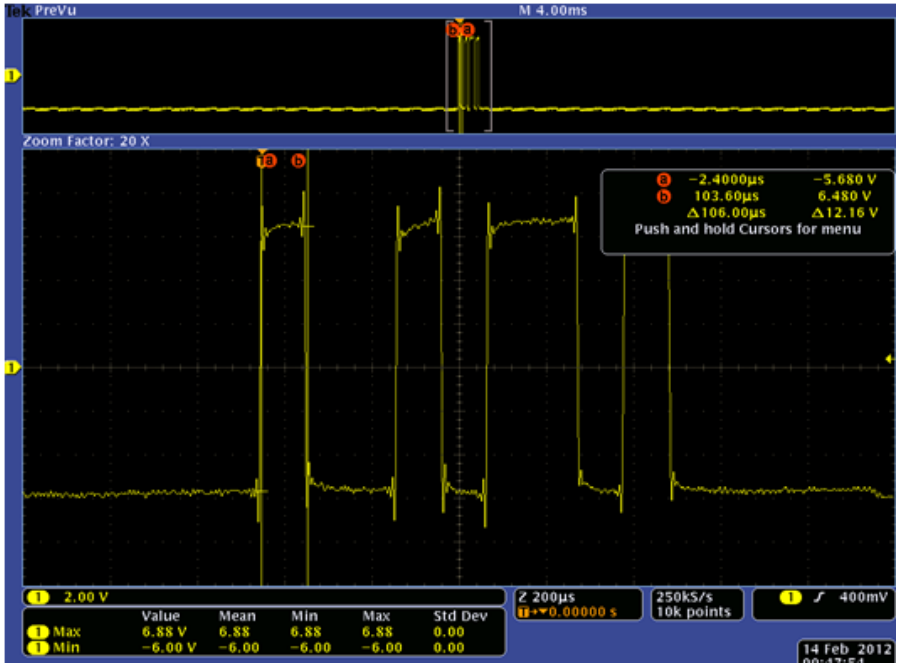
- RS-232 represents:
  - Logic 0, electrically with 3 to 15V
  - Logic 1, electrically with −3 to −15 V
  - This is called bipolar signaling
- Transceiver
  - converts UART logic levels to positive and negative levels expected by RS-232
  - provides electrostatic discharge protection when the user plugs in a cable
- MAX3232E: popular transceiver compatible with 3.3 and 5V logic
  - It contains a charge pump that, in conjunction with external capacitors, generates ±5 V outputs from a single low-voltage power supply
- Some serial peripherals intended for embedded systems omit the transceiver and just use:
  - 0V for a 0
  - and 3.3 or 5V for a 1



# RS-232 levels




Voltage levels for an ASCII "K" character (0x4B) with 1 start bit, 8 data bits (least significant bit first), 1 stop bit




RS-232 data line on the terminals of the receiver side (RxD) probed by an oscilloscope (for an ASCII "K" character (0x4B) with 1 start bit, 8 data bits, 1 stop bit, and no parity bits).

# Inter-Integrated Circuit ($I^2C$) Bus

- Inter-IC or $I^2C$-bus
  - Pronounced "*eye-squared-see*"
  - Sometimes called "*eye-two-see*"
- Simple, synchronous bidirectional 2-wire serial bus
  - Multi-controller/multi-target (a.k.a. master-slave)
- Invented by Philips Semiconductors in 1982
  - (now NXP Semiconductors)
  - Was a patented protocol
  - Since October 10, 2006, no licensing fees are required to implement $I^2C$ protocol
  - However, fees are required to obtain $I^2C$ slave addresses allocated by NXP…
- UM10204 $I^2C$-bus specification and user manual
  - https://www.nxp.com/docs/en/user-guide/UM10204.pdf


$I^2C$ is appropriate for peripherals where simplicity and low manufacturing cost are more important than speed

# I²C Applications

- Originally used by Philips inside televisions
- Now very common peripheral bus standard
- Intended for use in embedded systems
  - Also used (a lot) in (New) **space avionics!!!**
- Today, a variety of devices are available with I²C
  - Microcontroller, EEPROM, real-time clock (RTC), LCD drivers, A/D converters
  - Sensors (accelerometers, temperature, pressure)
  - Onboard Computers (OBCs), cameras, etc for control
- I²C-bus is a de facto world standard used in various control architectures

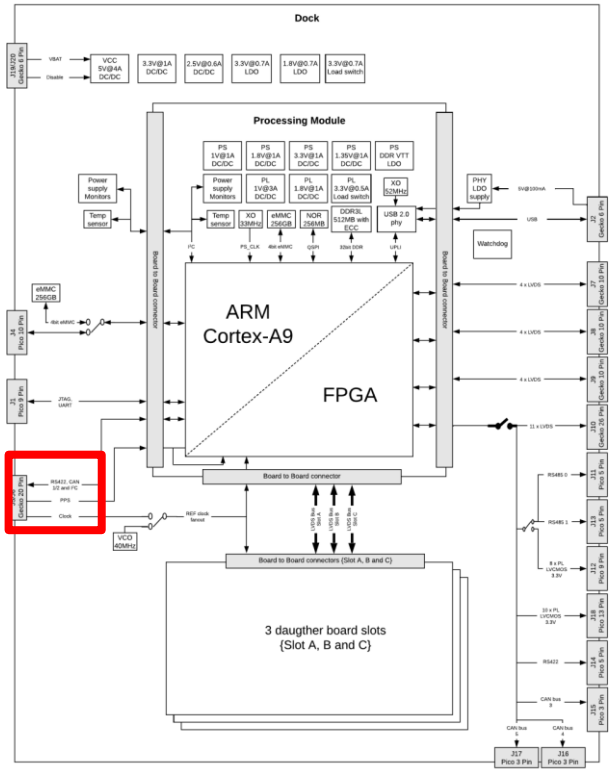
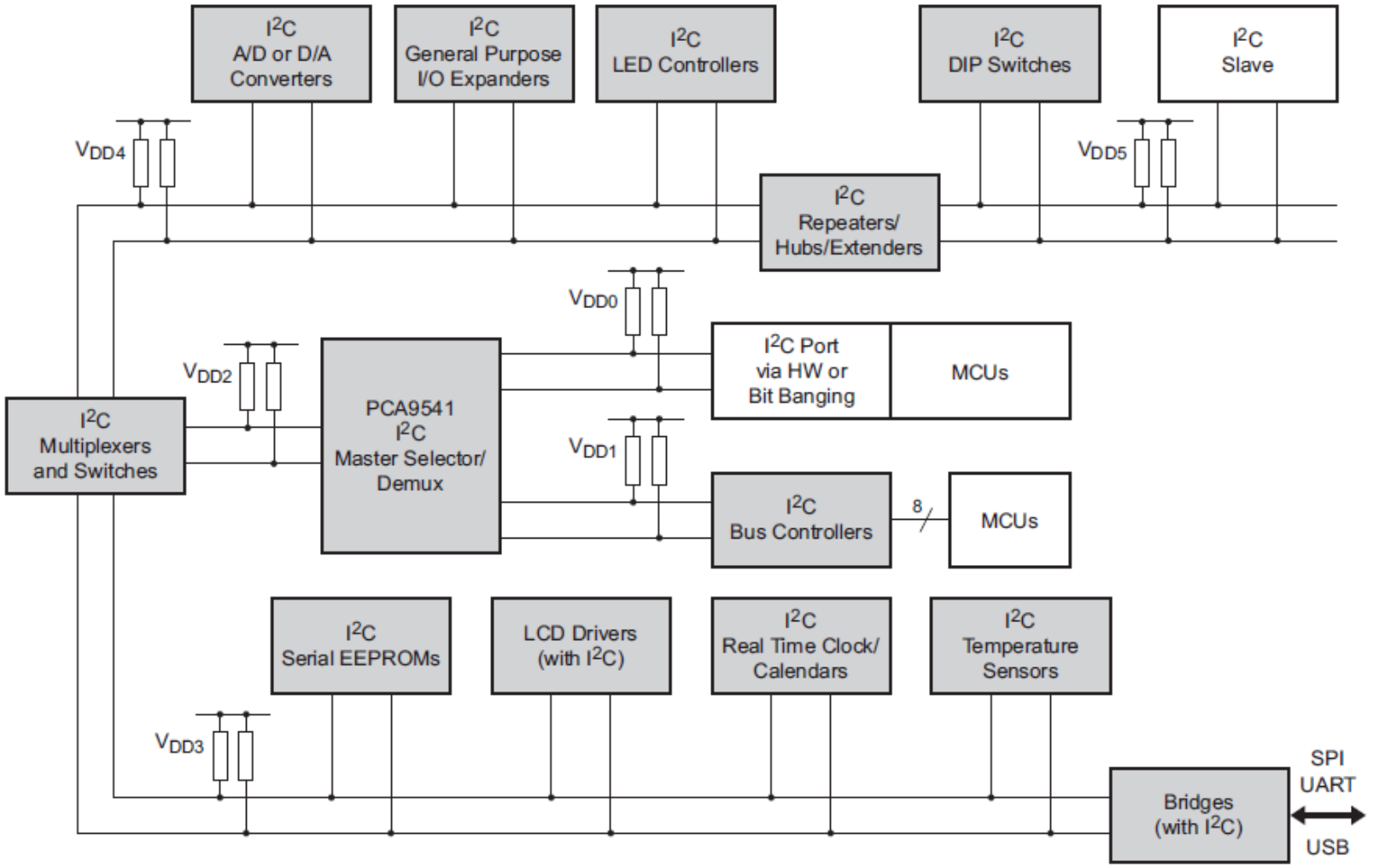



Figure 3.1: NanoMind HP MK3 Block diagram

# $I^2C$ Features

- Only 2 bus lines are required:
  - a **Serial Data (SDA)**
  - a **Serial Clock (SCL)**
- Each device connected to the bus is SW addressable by a unique address with a simple master/slave relationship
  - Masters can operate as master-transmitters or as master-receiver
- Multi-master bus including collision detection & arbitration if two or more masters simultaneously initiate data transfer
- Serial, 8-bit oriented, bidirectional data transfers can be made at:
  - up to 100 kbit/s in the Standard-mode
  - up to 400 kbit/s in the Fast-mode
  - up to 1 Mbit/s in Fast-mode Plus
  - up to 3.4 Mbit/s in High-speed mode
- Serial, 8-bit oriented, unidirectional data transfers
  - up to 5 Mbit/s in Ultra Fast-mode
- # of ICs connected to same bus limited only by a max bus capacitance

# I²C Bus Applications



# Definition of $I^2C$-bus terminology

| Term | Description |
|---|---|
| Transmitter | the device which sends data to the bus |
| Receiver | the device which receives data from the bus |
| Master | the device which initiates a transfer, generates clock signals and terminates a transfer |
| Slave | the device addressed by a master |
| Multi-master | more than one master can attempt to control the bus at the same time without corrupting the message |
| Arbitration | procedure to ensure that, if more than one master simultaneously tries to control the bus, only one is allowed to do so and the winning message is not corrupted |
| Synchronization | procedure to synchronize the clock signals of two or more devices |

# $I^2C$ Bus Protocol

- Two-wires carry information between devices connected to the bus
  - Serial data (SDA)
  - Serial clock (SCL)
- Each device is recognized by a unique address and can operate as either a transmitter or receiver, depending on the device function
  - E.g. an LCD driver may be only a receiver
  - E.g. a memory can both receive and transmit data
- Devices can also be considered as masters or slaves
  - A master is the device which initiates a data transfer on the bus and generates the clock signals to permit that transfer.
  - At that time, any device addressed is considered a slave
- Multi-master bus
  - More than one device capable of controlling the bus can be connected

# $I^2C$ Clock

- Not a “traditional” clock
- Normally is **kept “high”** using a pull-up
- Pulsed by the master during data transmission
  - Master could be either the transmitter or receiver
- Slave device can hold clock low if needs more time
  - Allows for flow control

# $I^2C$ Transaction

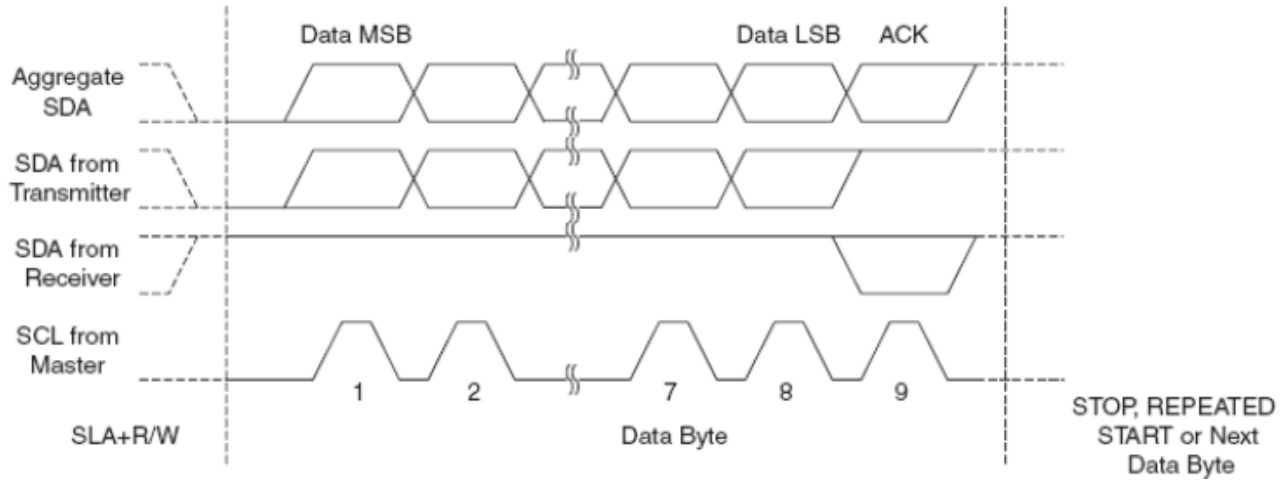
- Transmitter/receiver differs from master/slave
  - Master initiates transactions
  - Slave responds
- Transmitter sets data on SDA line, slave ACKs
  - For a read, slave is transmitter
  - For a write, master is transmitter

# $I^2C$ Address Transmission

- Data is always sampled on the rising clock edge
- Address is 7 bits
- An 8-th bit indicated read or write
  - High for read
  - Low for write
- Addresses assigned by Philips/NXP
  - For a fee
  - Was covered by patent

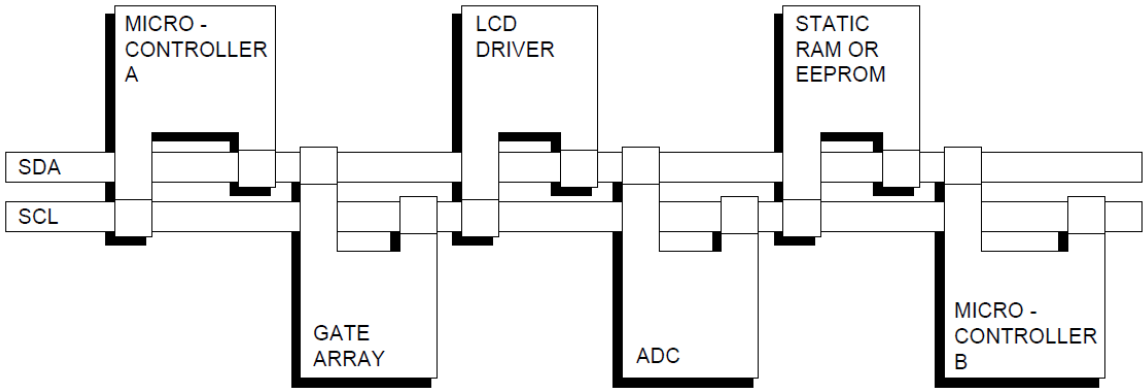
# I²C Data Transmission

- Transmitted just like address (8 bits)
- For a write, master transmits, slave acknowledges
- For a read, slave transmits, master acknowledges
- Transmission continues
  - Subsequent bytes sent
  - Continue until master creates stop condition



# Example I²C Bus Configuration

- Example: Data transfer between 2 microcontrollers (µC)
- Case 1: µC A wants to send information to µC B:
  - µC A (master), addresses µC B (slave)
  - µC A (master-transmitter), sends data to µC B (slave-receiver)
  - µC A terminates the transfer
- Case 2: If µC A wants to receive information from µC B:
  - µC A (master) addresses µC B (slave)
  - µC A (master-receiver) receives data from µC B (slave-transmitter)
  - µC A terminates the transfer
- **Even in Case 2, master (µC A) generates timing and terminates the transfer**

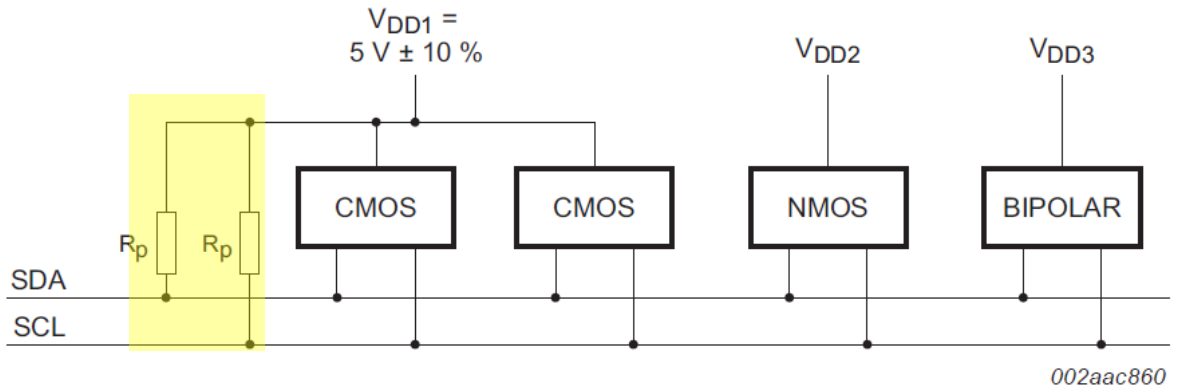


# Clock synchronization and arbitration

- Connecting more than one μC to $I^2C$-bus means more than one master could try to initiate a data transfer at the same time
- An arbitration procedure is developed
- Relies on the wired-AND connection of all $I^2C$ interfaces to $I^2C$-bus
- If two or more masters try to put information onto bus, the first to produce a ‘one’ when other produces a ‘zero’ loses arbitration
- The clock signals during arbitration are a synchronized combination of the clocks generated by the masters using the wired-AND connection to the SCL line

# SDA and SCL signals

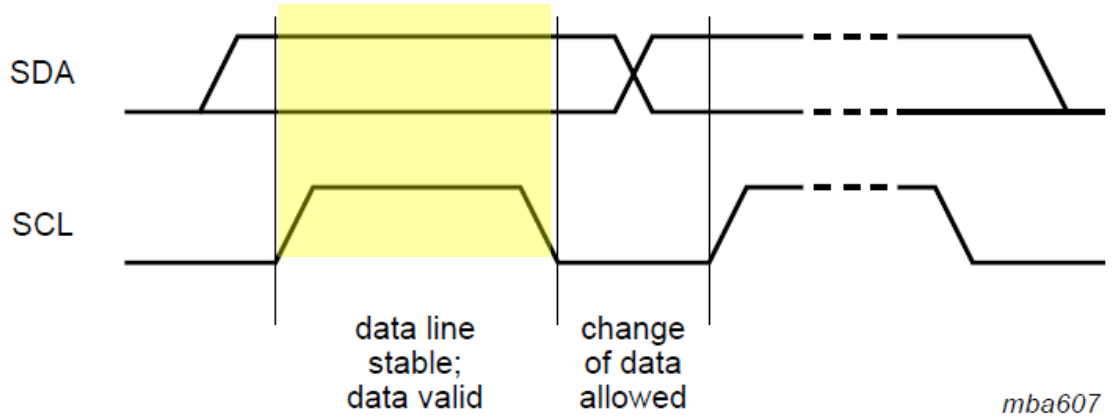
- SDA and SCL are bidirectional lines, connected to a positive supply voltage via pull-up resistor
  - When the bus is free, both lines are HIGH
- Output stages of devices connected to bus must have open-drain or open-collector to perform wired-AND function
- Bus capacitance limits # of interfaces connected to bus




$V_{DD2}$, $V_{DD3}$ are device-dependent (for example, 12 V).

# Data Validity

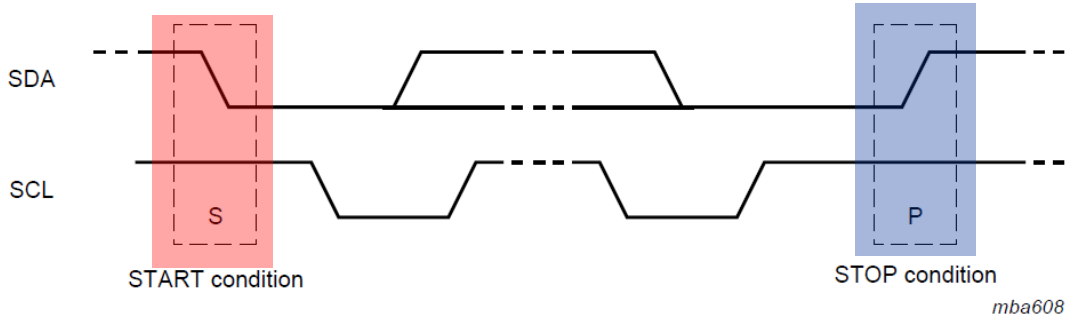
- Data on SDA line must be **stable** during HIGH clock period
  - HIGH or LOW state of SDA can only change when clock on SCL is LOW
- One clock pulse is generated for each data bit transferred




**Bit transfer on the I²C-bus**

# START and STOP conditions

- All transactions begin with a START (S)
  - **START condition**: A HIGH to LOW transition on SDA line while SCL is HIGH
  - Bus considered to be busy after the START condition
- All transactions are terminated by a STOP (P)
  - **STOP condition**: A LOW to HIGH transition on the SDA line while SCL is HIGH
  - Bus considered to be free again a certain time after the STOP condition
- START and STOP conditions are always generated by the Master
- Bus stays busy if repeated START (Sr) generated instead of STOP condition
- Detection of START and STOP conditions by devices is easy if they incorporate the necessary interfacing HW
  - µCs without such I/F must sample SDA line at least twice per clock period to sense the transition




START and STOP conditions

# Byte Format

- Every byte put on SDA line must be 8 bits
- Number of bytes that can be transmitted per transfer is unrestricted
- Each byte must be followed by an acknowledge (ACK) bit
- Data is transferred with the MSB first
- If a slave cannot receive or transmit another complete byte of data until it has performed some other function (e.g. servicing internal interrupt), it can **hold the clock line SCL LOW to force the master into a wait state**
  - Data transfer continues when slave is ready for another data byte and releases clock line SCL

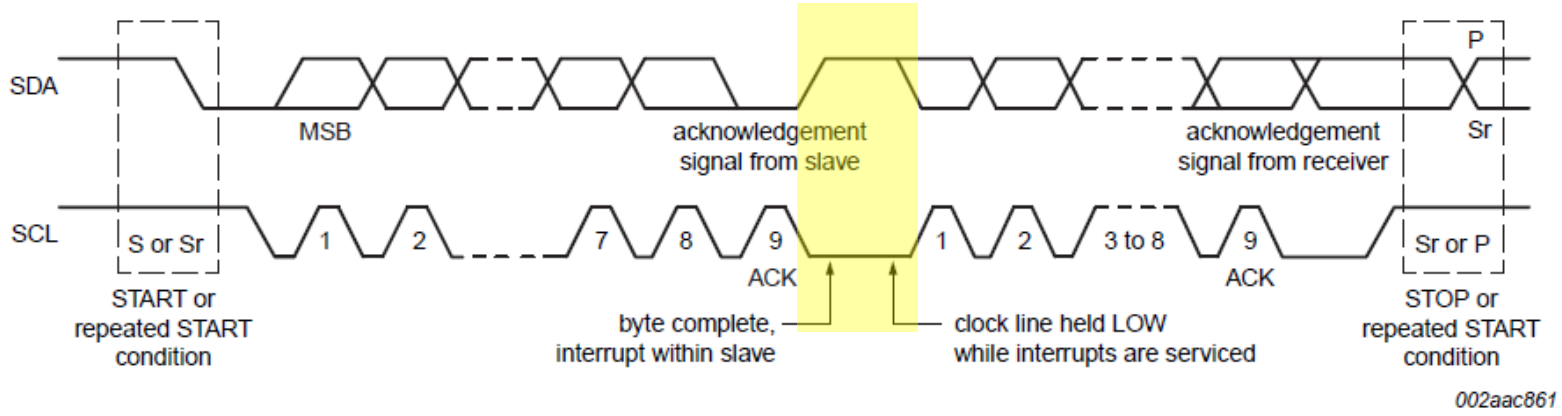



**Fig 6. Data transfer on the I²C-bus**

# Acknowledge (ACK) and Not Acknowledge (NACK)

- Acknowledge takes place after every byte
- ACK bit allows the receiver to signal the transmitter that the byte was successfully received, and another byte may be sent
- Master generates all clock pulses, including ACK 9th clock pulse
- Acknowledge signal: Transmitter releases the SDA line during the acknowledge clock pulse so the receiver can **pull the SDA line LOW** and it remains stable LOW during the HIGH period of this clock pulse

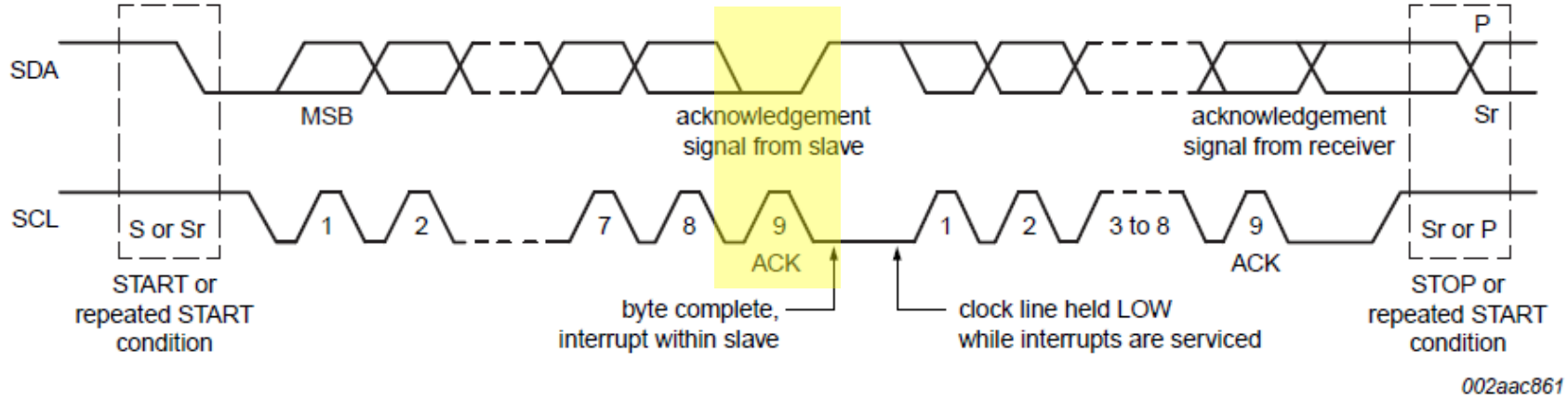



**Fig 6. Data transfer on the I²C-bus**

# Acknowledge (ACK) and Not Acknowledge (NACK)

- When SDA **remains HIGH** during 9th clock is defined as Not Acknowledge (NACK)
- The master can then generate either a STOP condition to abort the transfer, or a repeated START condition to start a new transfer
- There are 5 conditions that lead to the generation of a NACK:
  - No receiver is present on bus with the transmitted address > no device to respond with ACK
  - The receiver is unable to receive or transmit because it is performing some real-time function and is not ready to start communication with the master
  - During the transfer, the receiver gets data or commands that it does not understand
  - During the transfer, the receiver cannot receive any more data bytes
  - A master-receiver must signal the end of the transfer to the slave transmitter

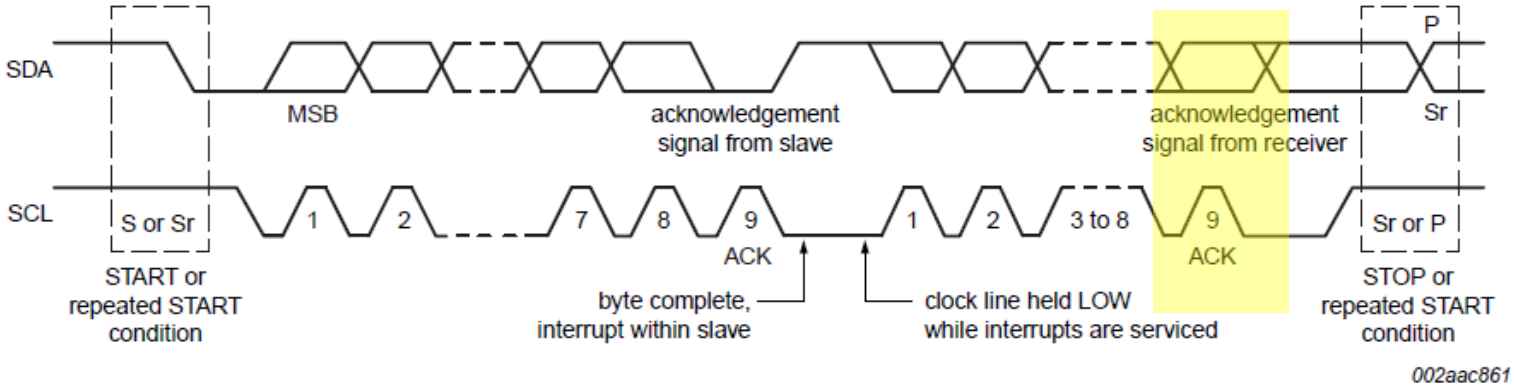
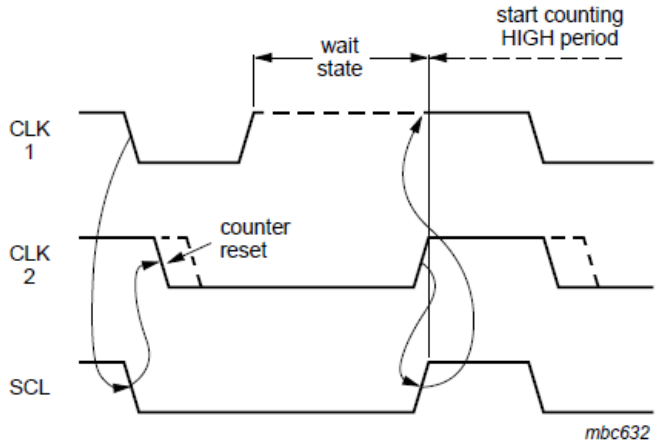



**Fig 6. Data transfer on the I²C-bus**

# Clock synchronization

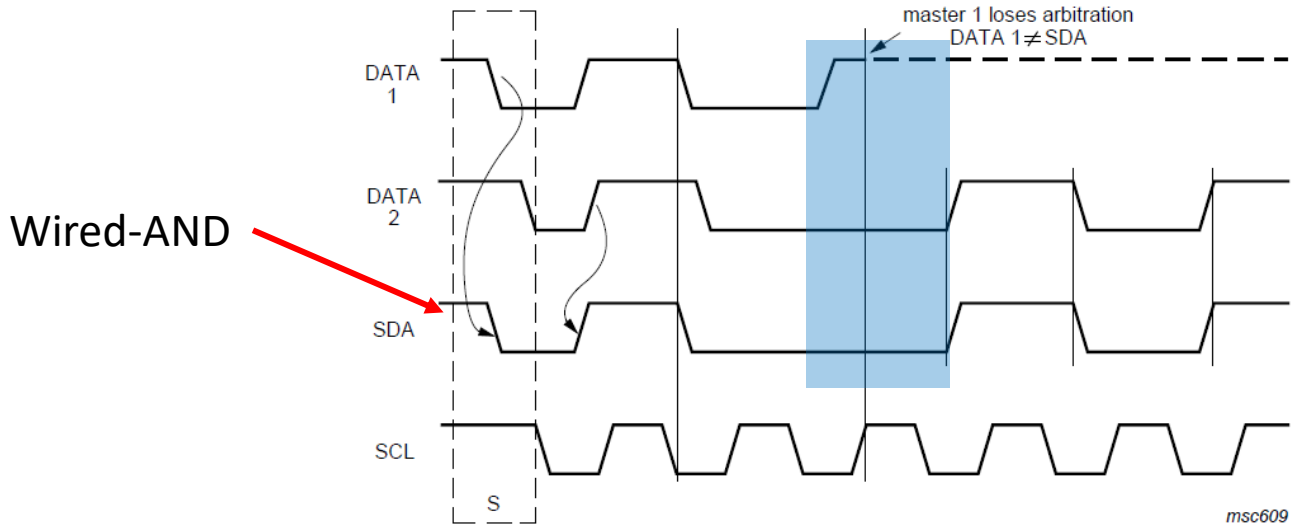
- Two masters can begin transmitting on a free bus at the same time
- **Clock synchronization and arbitration**: method for deciding which master takes control of the bus and complete its transmission
  - In single master systems, clock synchronization & arbitration not needed
- Clock synchronization performed using wired-AND to SCL line
- A synchronized SCL is generated with its LOW period determined by the master with the longest clock LOW period, and its HIGH period determined by the one with the shortest clock HIGH period




Clock synchronization during the arbitration procedure

# Arbitration

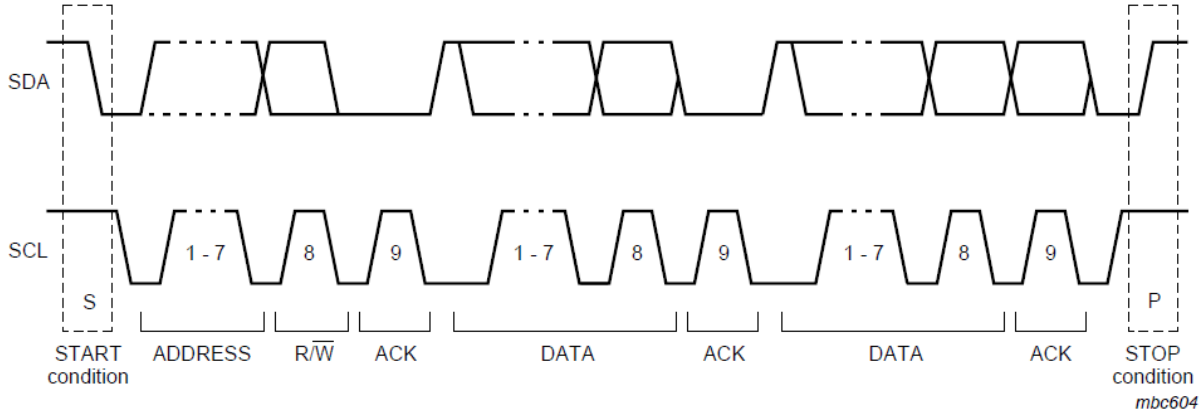
- A master may start a transfer only if the bus is free
- Arbitration is required to determine which master will complete its transmission
- Arbitration proceeds bit by bit
- The first time a master tries to send HIGH, but detects that SDA level is LOW (different from what expected and concludes that another node is transmitting) master knows it has lost the arbitration and turns off its SDA output driver. The other master goes on to complete its transaction
- A master that loses arbitration can generate clock pulses until the end of the byte in which it loses the arbitration and must restart its transaction when bus is free



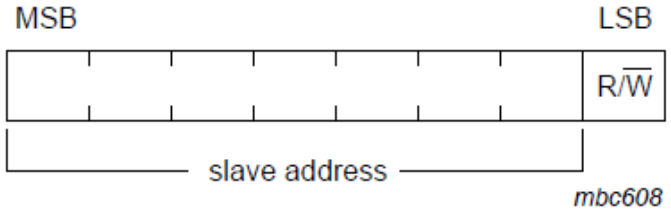

Arbitration procedure of two masters

# Slave Address and R/$\overline{W}$ bit

- After the START condition (S), a slave address is sent
- Address is 7 bits long followed by an 8th bit which is a data direction bit (R/$\overline{W}$ )
  - 'Zero' indicates a transmission (WRITE), a 'one' indicates a request for data (READ)
- A data transfer is always terminated by a STOP condition (P) generated by the master
- If a master still wishes to communicate on the bus, it can generate a repeated START condition (Sr) and address another slave without first generating a STOP condition.
- Various combinations of read/write formats are then possible within such a transfer



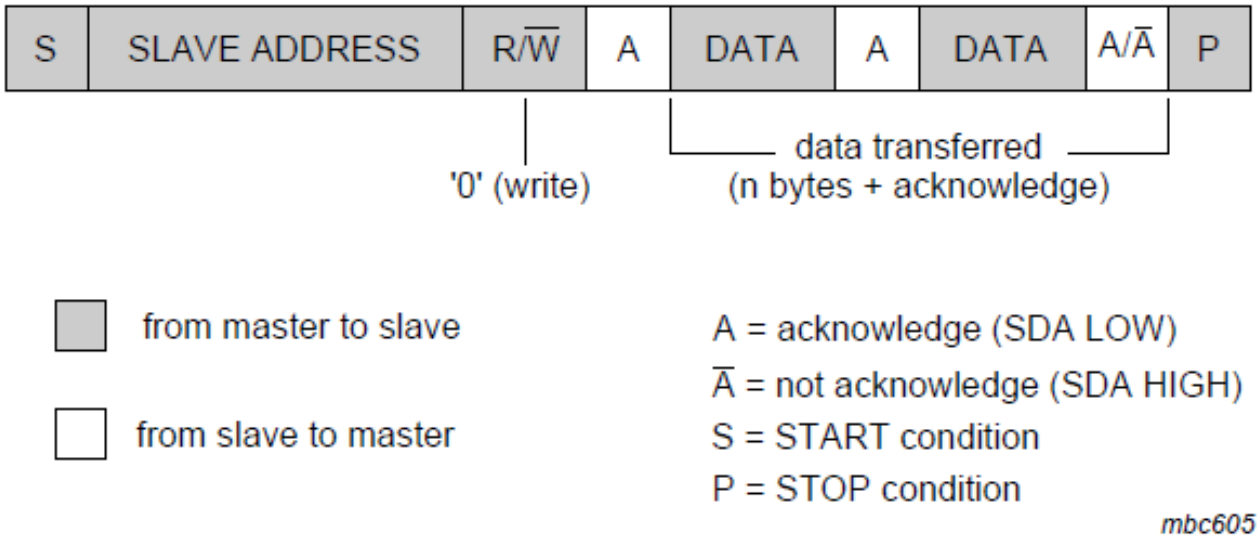

A complete data transfer




First byte after the START procedure

# Data transfer formats

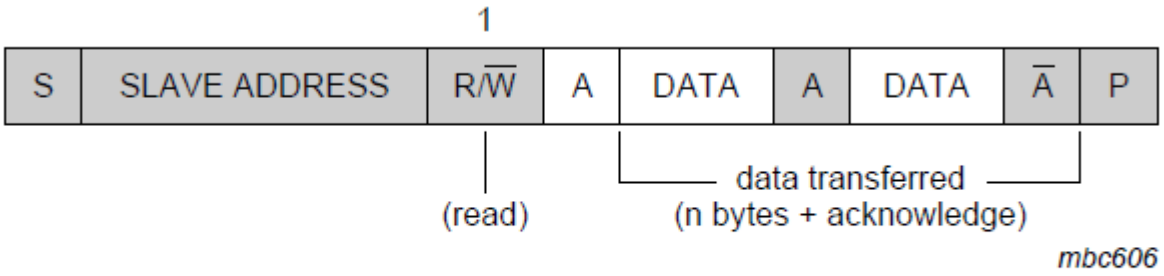
- Master-transmitter transmits to slave-receiver
  - The transfer direction is not changed
  - The slave receiver acknowledges each byte




**A master-transmitter addressing a slave receiver with a 7-bit address (the transfer direction is not changed)**

# Data transfer formats

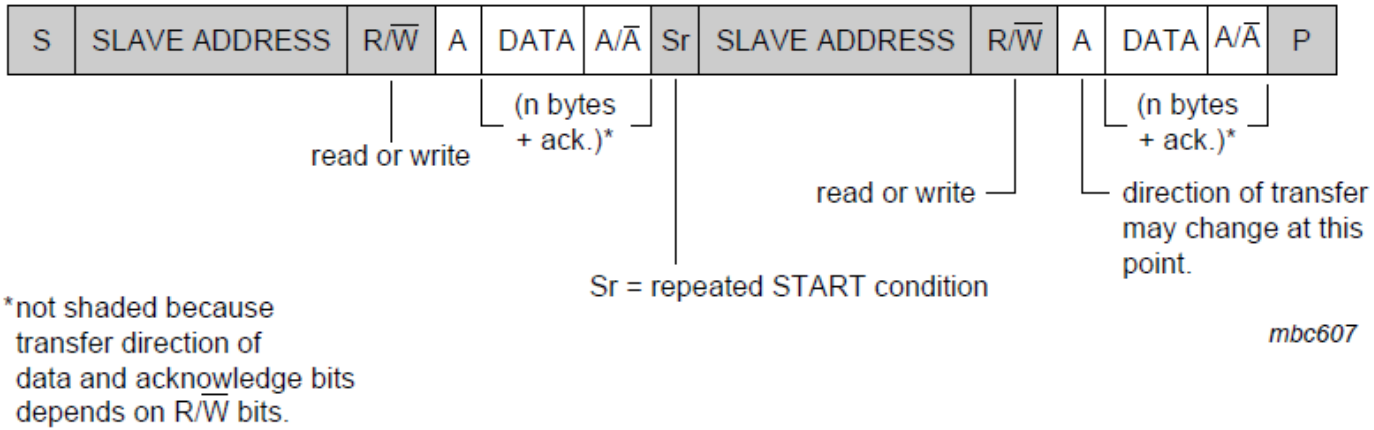
- Master reads slave immediately after first byte
  - At the moment of the first acknowledge, the master-transmitter becomes a master-receiver and the slave-receiver becomes a slave-transmitter. This first acknowledge is still generated by the slave
  - The master generates subsequent acknowledges
  - The STOP condition is generated by the master, which sends a not-acknowledge ($\bar{A}$) just before the STOP condition




**A master reads a slave immediately after the first byte**

# Data transfer formats

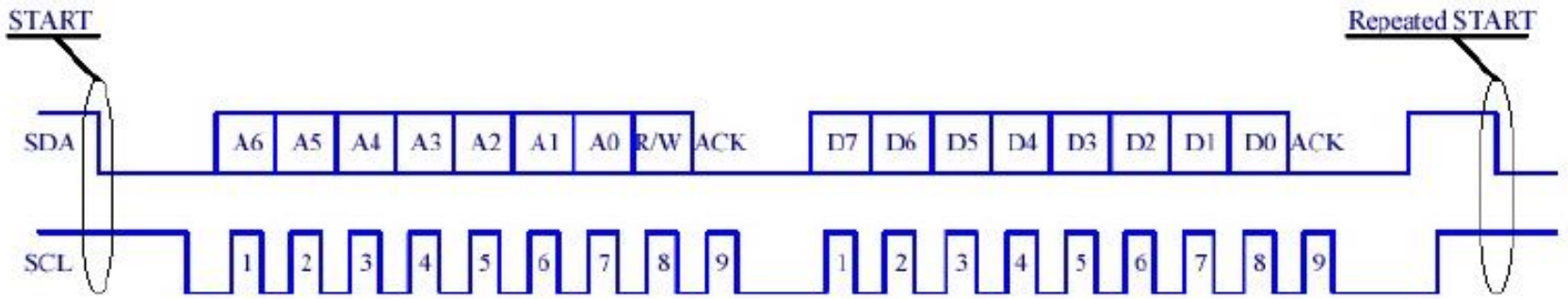
- Combined format
  - During a change of direction within a transfer, the START condition and the slave address are both repeated, but with the R/W bit reversed
  - If a master-receiver sends a repeated START condition, it sends a not-acknowledge ($\bar{A}$) just before the repeated START condition
  - Combined formats can be used, for example, to control a serial memory
    - The internal memory location must be written during the first data byte
    - After the START condition and slave address is repeated, data can be transferred




**3. Combined format**

# I²C Write Sequence

- A typical I²C bus sequence for writing to a slave:
  - Send a START sequence.
  - Send the I²C device address.
  - Send the data byte.
  - Optionally send additional data bytes (after repeating START)
  - Send the STOP sequence after all data bytes have been sent
- The Slave responds by setting the ACK bit (Acknowledge)

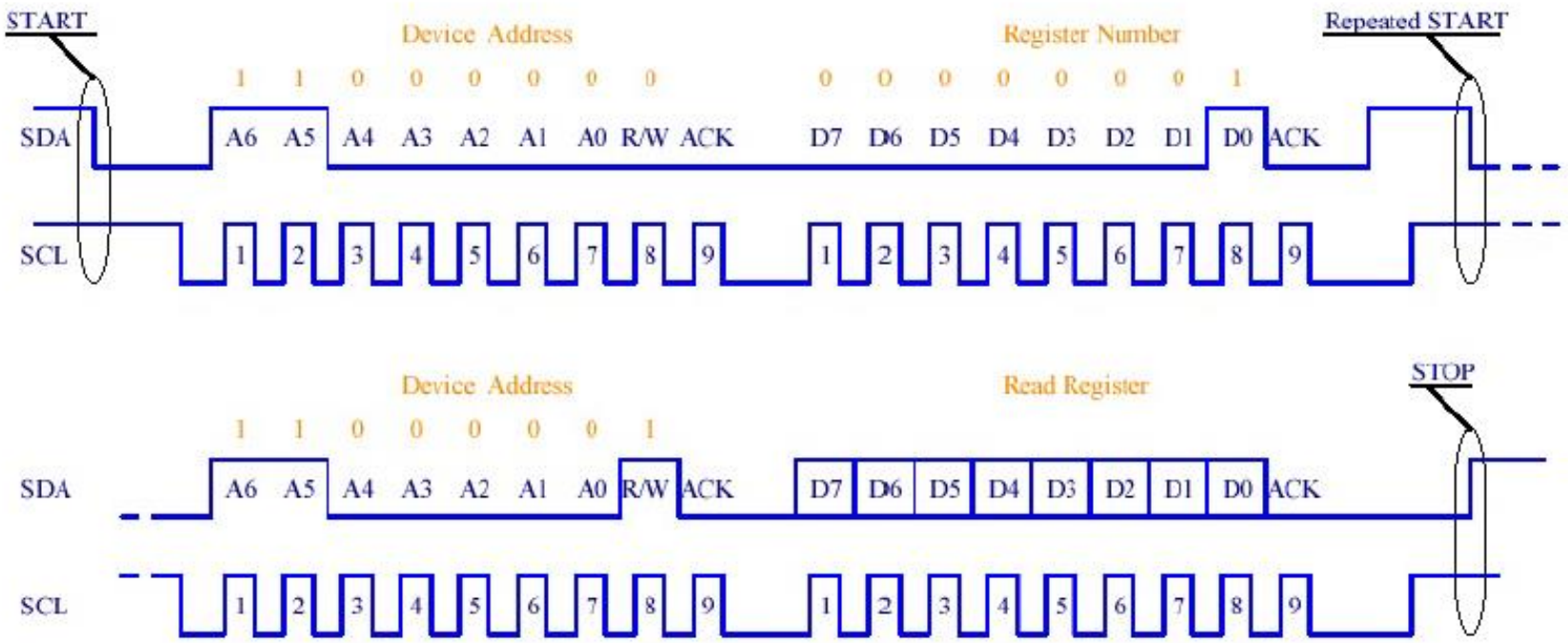


# $I^2C$ Read Sequence

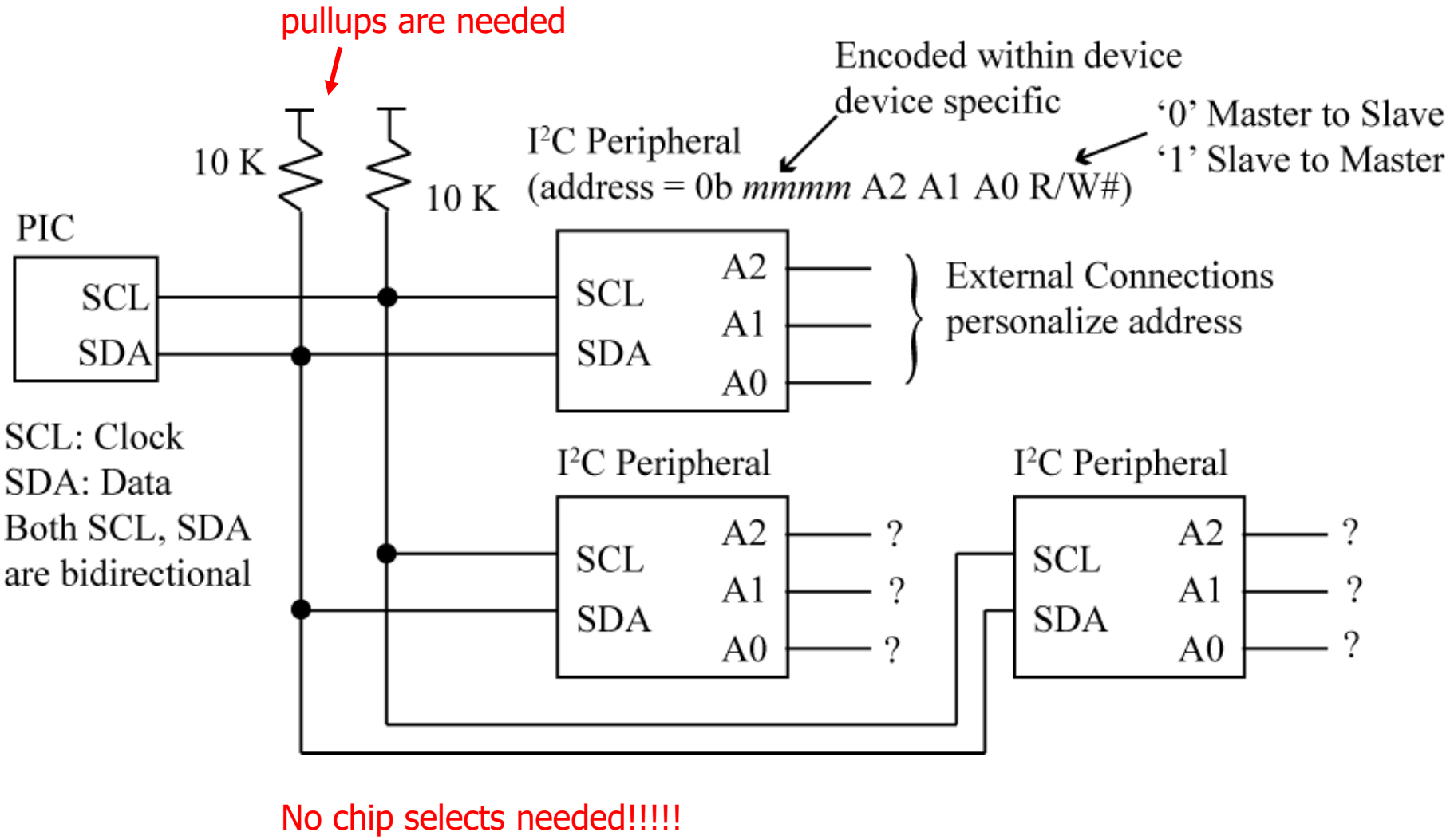
- Reading an $I^2C$ Slave device usually begins by writing to it
  - You must tell the chip which internal register you want to read
- $I^2C$ Read Sequence:
  - Send the START condition
  - Send the device address
  - Send the number of the register you want to read
  - Send a repeated START condition
  - Send the device address
  - Read the data byte from the slave
  - Send the STOP sequence

# I²C Read Example

- I²C Read example using device address 1100000 and reading register number 1

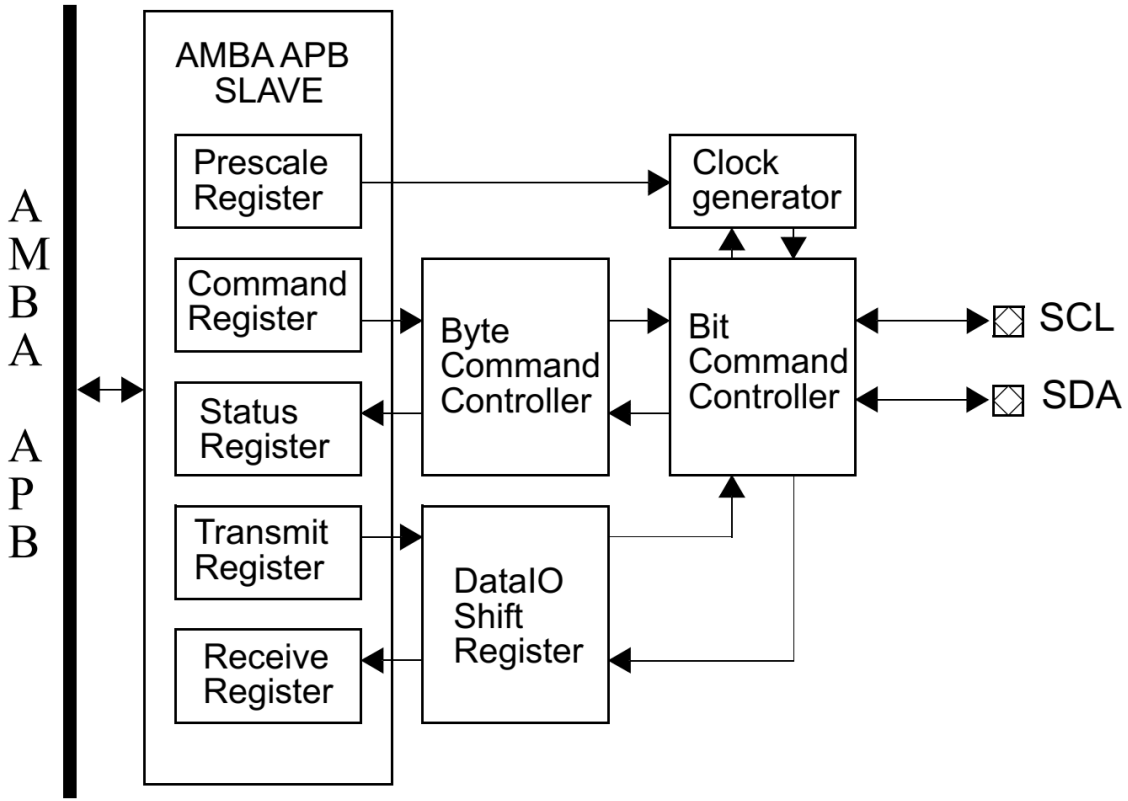


# I²C Bus Addressing



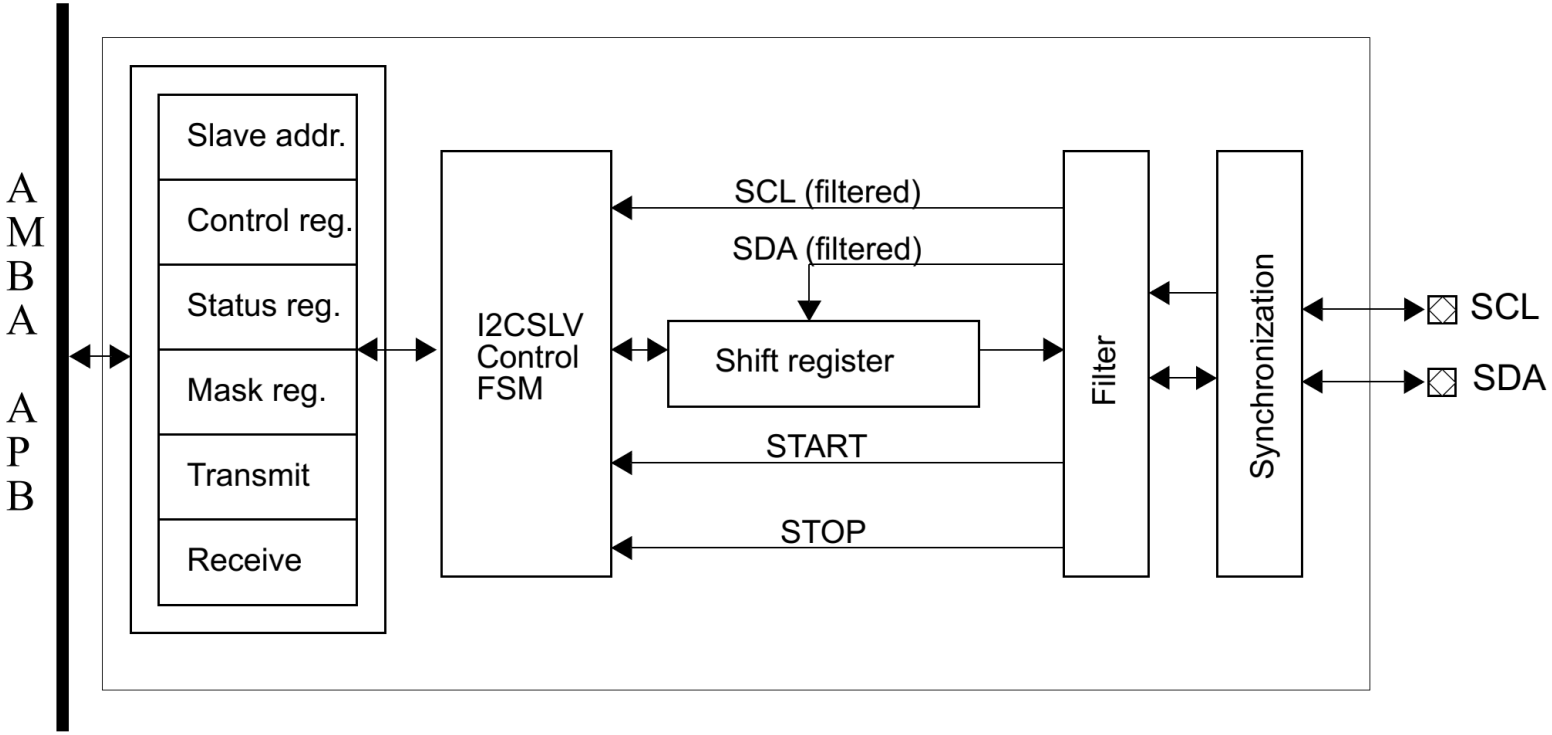

No chip selects needed!!!!!

## 86 I2CMST - I²C-mast




The I²C-master core is a modified version of the OpenCores I²C-Master with an AMBA APB interface. The core is compatible with Philips I²C standard and supports 7- and 10-bit addressing. Standard-mode (100 kb/s) and Fast-mode (400 kb/s) operation are supported directly. External pull-up resistors must be supplied for both bus lines.

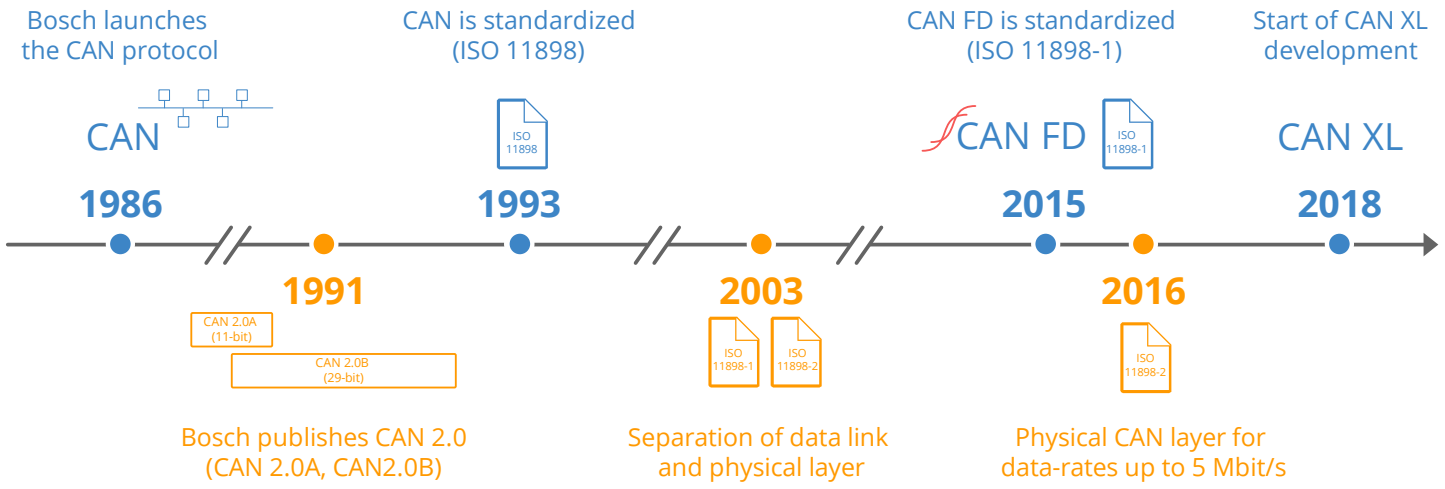
## 87 I2CSLV - I²C slave




The I²C slave core is a simple I²C slave that provides a link between the I²C bus and the AMBA APB. The core is compatible with Philips I²C standard and supports 7- and 10-bit addressing with an optionally software programmable address. Standard-mode (100 kb/s) and Fast-mode (400 kb/s) operation are supported directly. External pull-up resistors must be supplied for both bus lines.

# Controller Area Network (CAN) bus

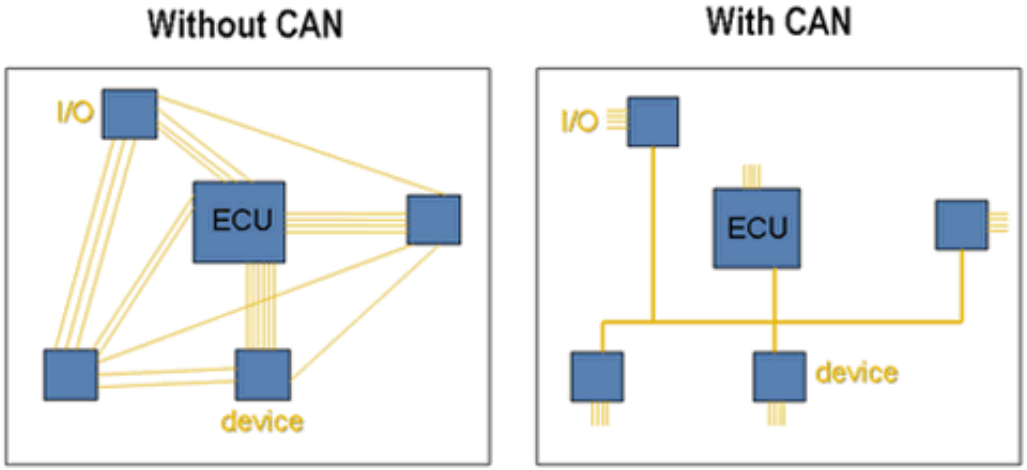
- CAN is **multi-master** 2-wire differential serial-bus message-based protocol
- Launched in 1986 by Bosch GmbH to provide a cost-effective communications bus for automotive applications
  - ADAS, transmission, airbus, ABS, cruise control, power windows, ……
  - First CAN controller chip: Intel 1987
  - First car with CAN: 1991 Mercedes W140 S-class (5 CAN bus nodes)
- CAN today successfully replaces point-to-point connections in many applications
  - Automotive, space avionics, industrial machines, building automation, elevators, escalators, medical instruments and equipment etc

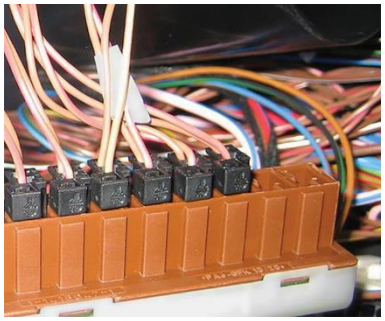


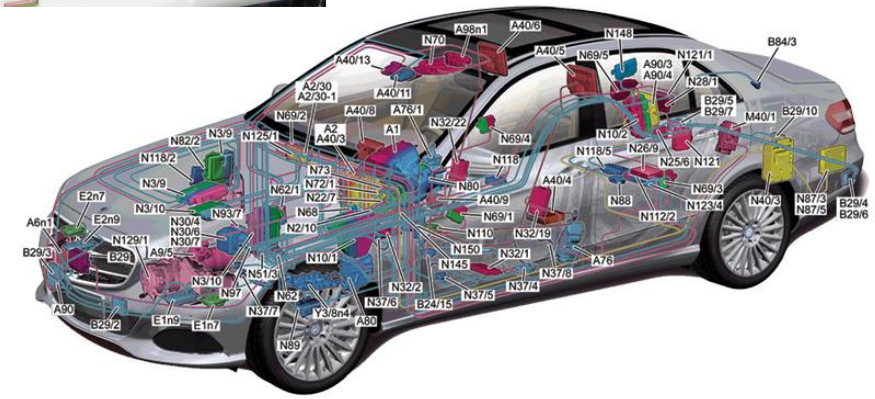
# CAN bus

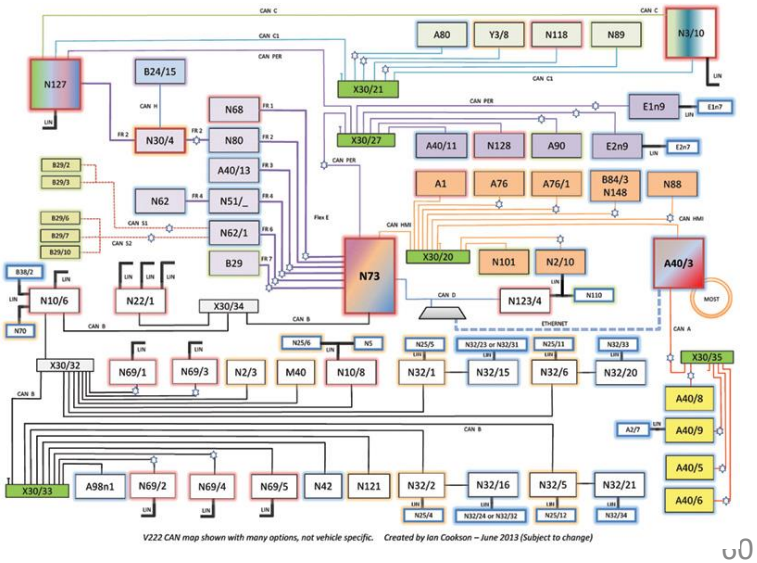
- CAN is attractive for embedded control systems
  - **High Reliability**: CAN ensures robust data transmission in noisy environments making it suitable for critical applications such as automotive & aerospace systems
  - **Scalability**: CAN supports a scalable network architecture, allowing the addition of nodes without significant impact on the overall system performance
  - **Deterministic Communication**: With its time-triggered communication mechanism, CAN provides deterministic and predictable data transmission which is critical for automotive safety systems and aerospace
  - **Efficient Bandwidth Utilization**: CAN efficiently utilizes the available bandwidth by prioritizing messages based on their identifiers. This ensures that critical messages can be transmitted without delay, enhancing overall system efficiency.
  - **Error Detection and Handling**: CAN protocol incorporates robust error detection and handling mechanisms. It can detect errors (e.g. bit errors or frame errors), enabling the identification and correction of issues, which is vital for maintaining system integrity

# CAN bus advantage

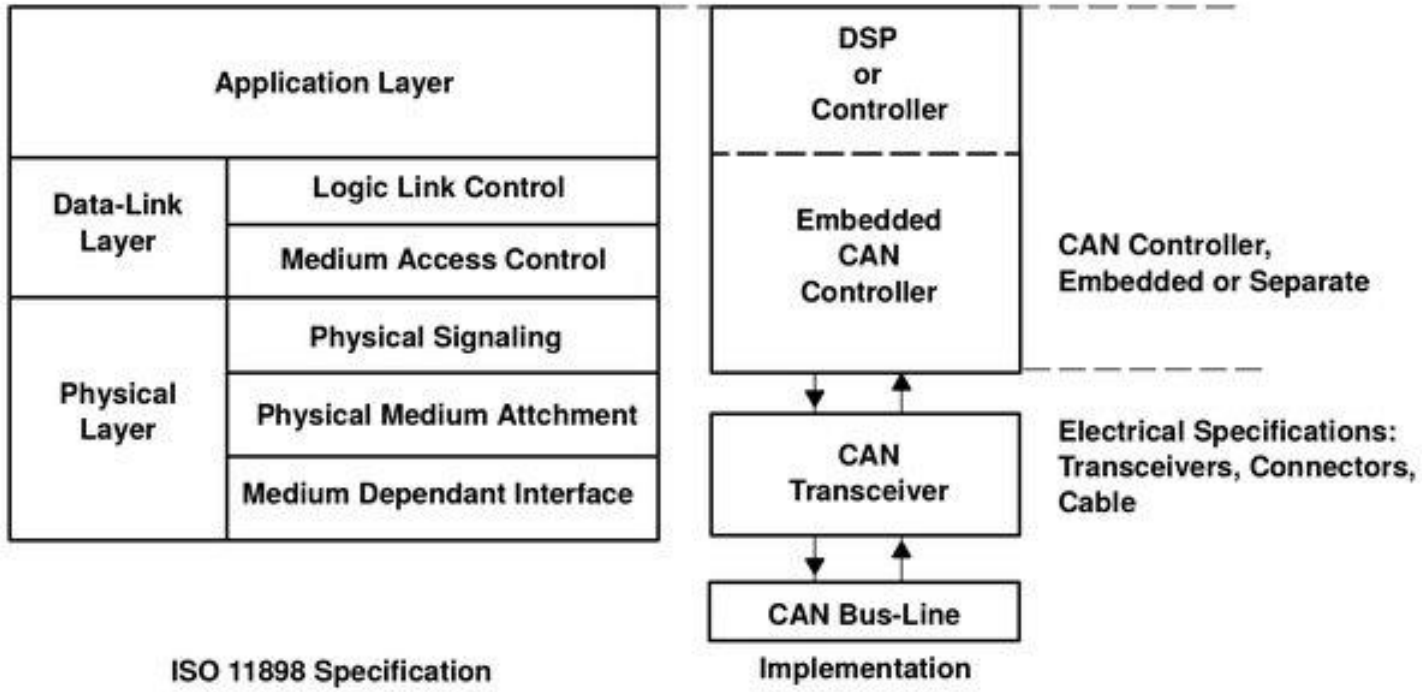




# CAN bus and OSI Layers

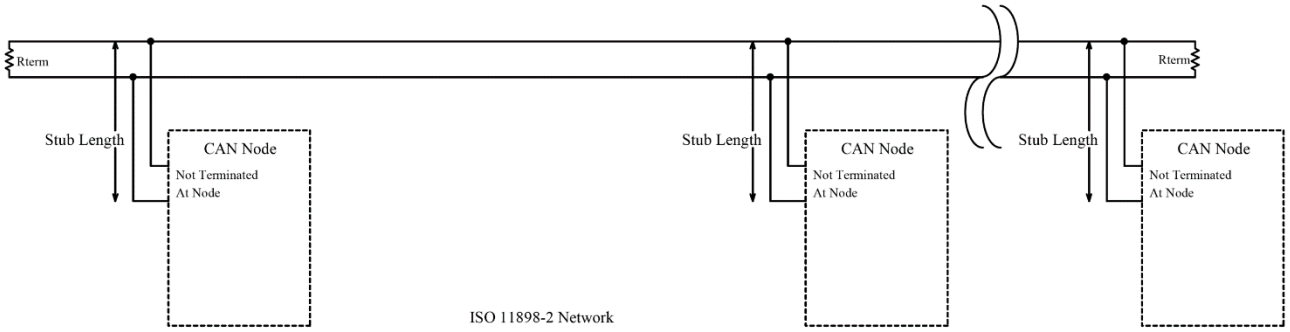


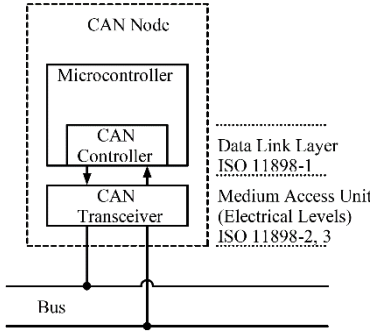
# CAN Bus Types

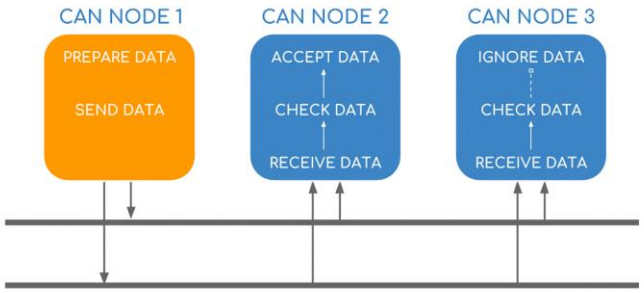
- CAN High Speed (CAN 2.0B)
  - Speed: Up to 1Mbps
  - Range: 40m
  - 29bit Message Identifier
  - Termination with 120 Ω Resistor
- CAN Low Speed (CAN 2.0A)
  - Speed: Up to 125Kbps
  - Range: 500m
  - 11bit Message Identifier
  - Overall termination resistance ≈100 Ω
- CAN FD (Flexible Data Rate)
  - Speed: Up to 15Mbps
  - Range: 10m

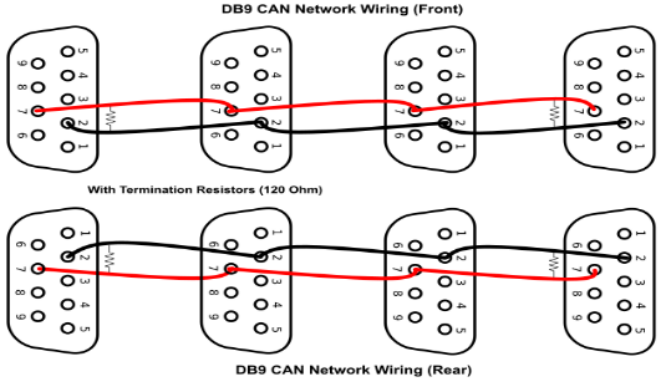
# CAN Bus Topology

- CAN allows multiple devices ("nodes") to communicate
  - Two or more nodes are required on the CAN network to communicate



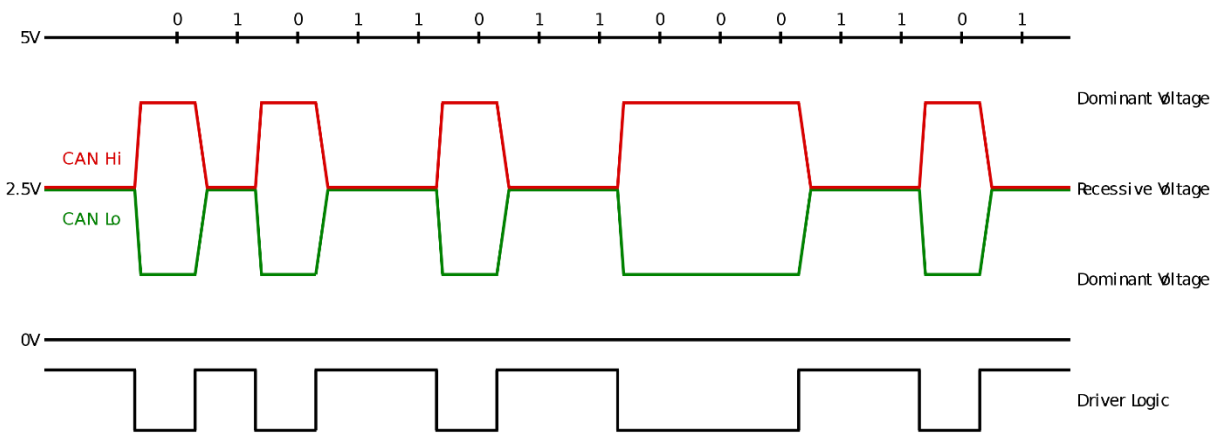



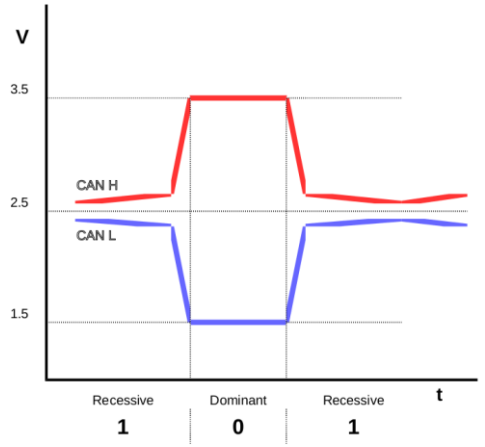




- pin 2: CAN-Low (CAN−)
- pin 3: GND (ground)
- pin 7: CAN-High (CAN+)
- pin 9: CAN V+ (power)

# CAN Communication

- All nodes are connected to each other through a two-wire bus
  - Wires are a twisted pair with a 120 Ω (nominal) characteristic impedance
- CAN bus uses **differential wired-AND** signals
- Two signals, CAN-high (CANH) and CAN-low (CANL) are either driven to:
  - Dominant state (logic 0) with CANH > CANL
  - Recessive state (logic 1), using pull-up resistors, with CANH ≤ CANL
- A **0 data bit encodes a dominant state**, while a 1 data bit encodes a recessive state
  - Supports wired-AND convention, which gives nodes with lower ID numbers bus priority
- Bus is always in recessive state (logic 1)
  - When a node has to transmit 1, it leaves bus in default state
  - When a node has to transmit 0, it drives bus in dominant state



# CAN bus Frames

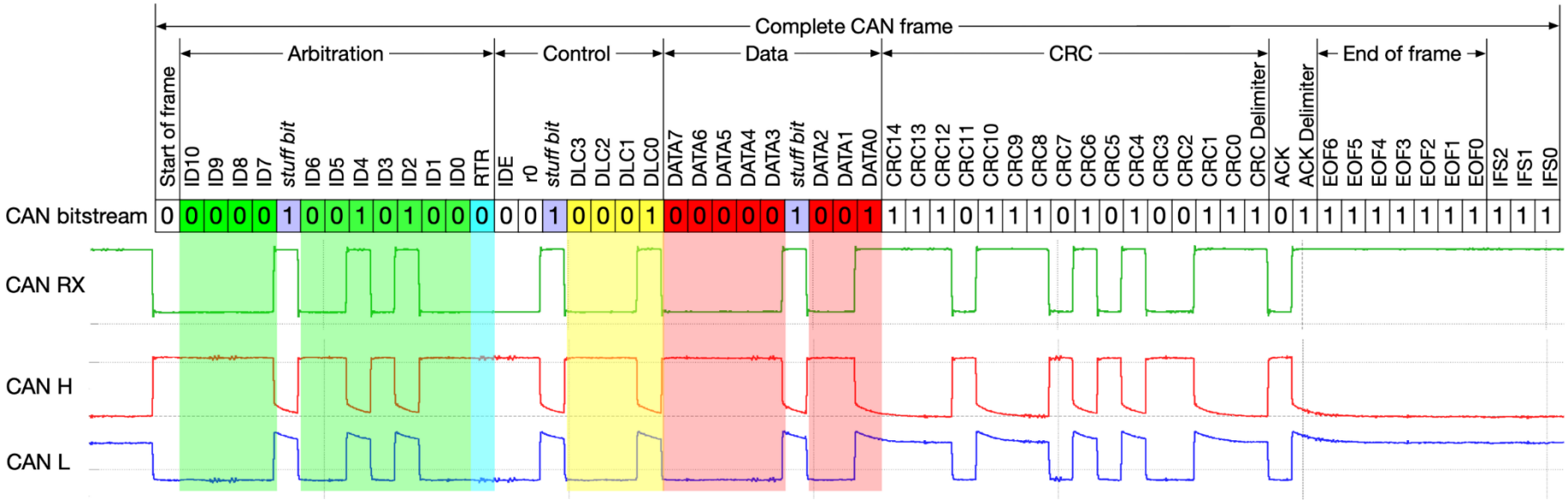
- CAN bus has four frame types:
  - **Data frame**: containing node data for transmission
  - **Remote frame**: requesting transmission of a specific identifier
  - **Error frame**: transmitted by any node detecting an error
  - **Overload frame**: to inject a delay between data or remote frame

# CAN Data Frame

- CAN nodes transmit data in the form of CAN data frames
  - Standard-or base format CAN frame - 11 bits identifier frame (CAN 2.0A)
  - Extended 29-bit identifier frame (CAN 2.0B)

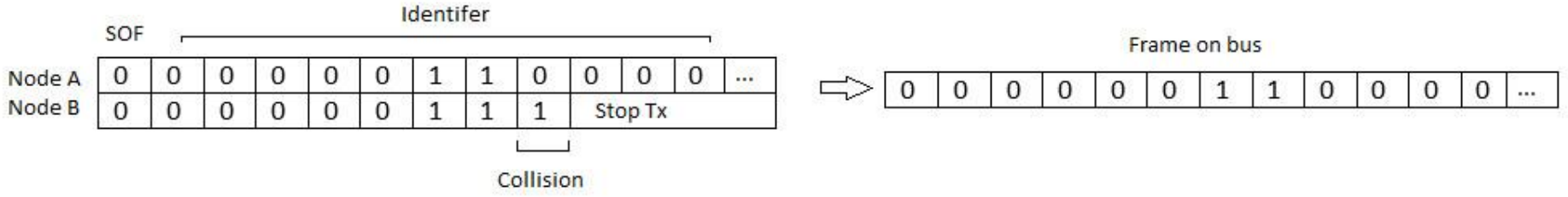

- **SOF: Start of Frame** is a 'dominant 0' to tell other nodes that a CAN node intends to talk
- **ID: Unique frame identifier** - lower values have higher priority
- RTR: Remote Transmission Request indicates whether identifies whether the frame is a data frame (dominant 0) or a request (remote) frame (recessive 1)
- **Control**: contains Identifier Extension bit (IDE) which is a 'dominant 0' for 11-bit. It also contains 4-bit Data Length Code (DLC) specifying length of data bytes to be transmitted (0 to 8 bytes)
- **Data**: contains data bytes, including CAN signals that can be extracted and decoded for information
- **CRC**: The Cyclic Redundancy Check is used to ensure data integrity
- **ACK**: indicates if node has acknowledged and received the data correctly and transmits a dominant level (0) and thus overrides the recessive level (1) of transmitter. A receiving node can transmit a recessive (1) to indicate that it did not receive a valid frame, but another node that did receive a valid frame may override this with a dominant. Transmitting node cannot know that the message has been received by all of the nodes on the CAN network.
- **EOF**: The EOF marks the end of the CAN frame

# CAN Frame

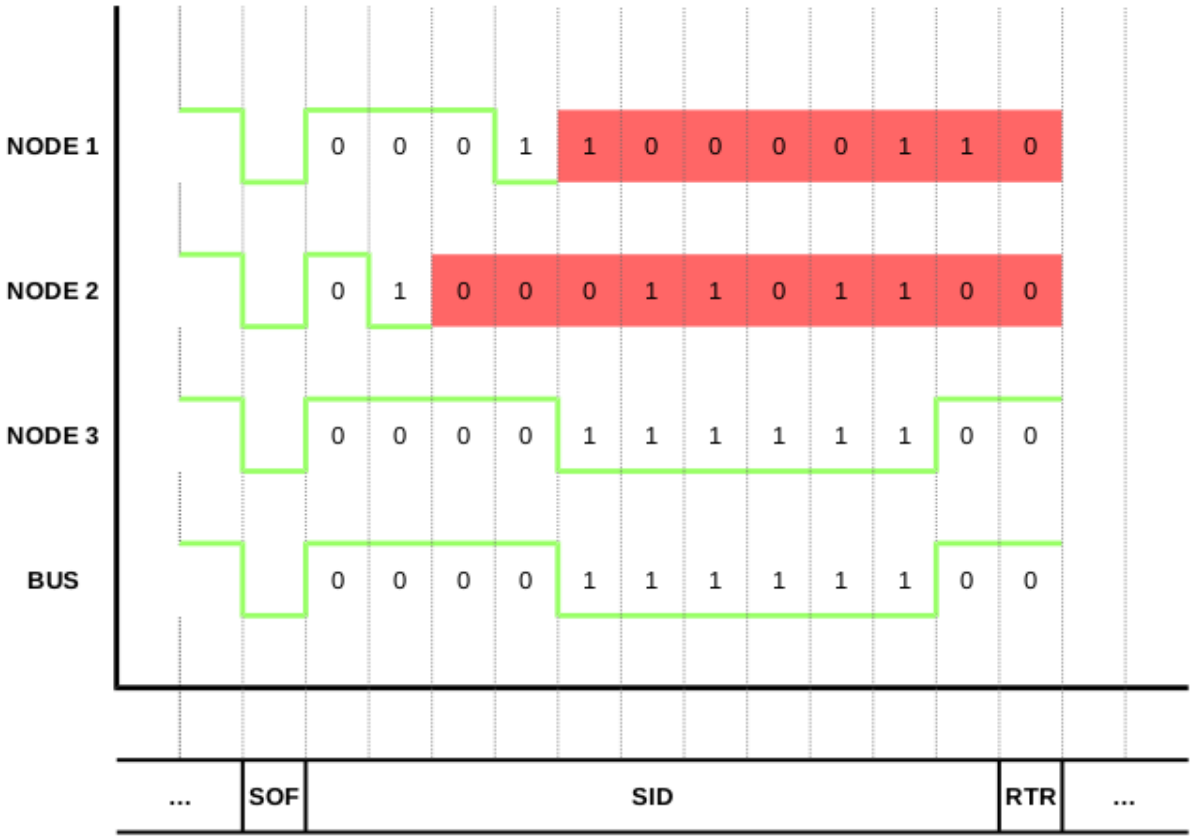

Complete CAN frame
Arbitration
Control
Data
CRC
End of frame
Start of frame
ID10 ID9 ID8 ID7 stuff bit ID6 ID5 ID4 ID3 ID2 ID1 ID0 RTR
IDE r0 stuff bit DLC3 DLC2 DLC1 DLC0
DATA7 DATA6 DATA5 DATA4 DATA3 stuff bit DATA2 DATA1 DATA0
CRC14 CRC13 CRC12 CRC11 CRC10 CRC9 CRC8 CRC7 CRC6 CRC5 CRC4 CRC3 CRC2 CRC1 CRC0
CRC Delimiter
ACK
ACK Delimiter
EOF6 EOF5 EOF4 EOF3 EOF2 EOF1 EOF0
IFS2 IFS1 IFS0
CAN bitstream
0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 1 0 0 1 1 1 1 0 1 1 1 0 1 0 1 0 1 0 0 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1
CAN RX
CAN H
CAN L

# CAN Arbitration Example (1)

- Two nodes (A,B) start transmitting (SOF) at same time
- After SOF, they start transmitting ID
- Each node transmits a bit and then observes bus
  - If bit sent is same as bit sensed, it continues to transmit the identifier
  - If bit sent and the bit sensed back are different, it knows that a higher priority message is being transmitted on the bus and it starts listening and stops transmitting
- Node with lower identifier (A) will continue to transmit as it will drive the bus to dominant state (0) while node (B) that intend to keep it in default state (recessive state 1), will read back the dominant state on the bus and stop transmitting
- <u>This mechanism thus preserves the data as the frame with lower priority is not corrupted due to simultaneous transmissions</u>



# CAN Arbitration Example (2)



# CAN Error Handling

- CAN bus errors can occur due to:
  - Faulty cables
  - Noise
  - Incorrect termination
  - Malfunctioning CAN nodes
- CAN bus error handling identifies and rejects erroneous messages, enabling a sender to re-transmit the message
  - Ensures that temporary local disturbances will not result in invalid/lost data
  - Identify & disconnect nodes that consistently transmit erroneous messages
  - Transmitter attempts to re-send the message
    - If it wins arbitration (and there are no errors), the message is successfully sent
- The ability of problematic CAN nodes to transmit data is thus gracefully reduced to avoid further CAN errors and bus jamming
  - CAN nodes keep track of their own CAN error counters and change state (active, passive, bus off) depending on their counters (see next slides)

# CAN Error Detection

- When a CAN node detects a frame error, it transmits an **Error Flag**
- Error flag is normally detected by the node transmitting the invalid frame, which then retransmits to correct the error
  - Retransmission starts over from SOF, thus arbitration with other nodes can occur again
- CAN nodes detect the following errors:
  - Bit error
  - Stuff error
  - CRC error
  - Form error
  - Acknowledgment error

# CAN Errors

- **Bit Error (Transmitter)**

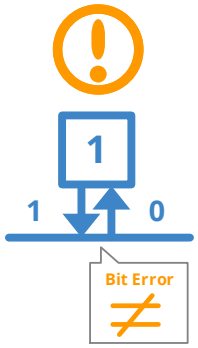
  **Evaluated at bit-level**

  – CAN nodes monitor CAN bus on a bit-by-bit basis

  – If bit monitored is different from bit transmitted, a **Bit Error** is detected

  – Node raises an **Active Error Flag** to inform other nodes

  – Bit error check applies only to the following fields of the transmitted frame: Data Length Code, Data Bytes, CRC




- **Bit Stuffing Error (Receiver)**

  – See next slides…

- **CRC Error (Receiver)**

  **Evaluated at message-level**

  – Detected by a receiving node when calculated CRC differs from actual CRC in frame

- **Form Error (Receiver)**

  – Occurs upon a violation of the standard CAN frame encoding

    - E.g. if a CAN node begins transmitting SOF bit for a new frame before EOF sequence completes for a previous frame (does not wait for bus idle)

- **ACK Error (Transmitter)**

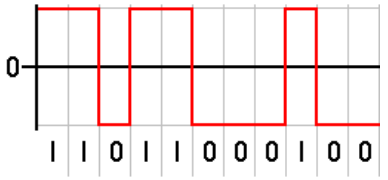
  – Detected by a transmitting node when it does not detect a dominant ACK bit

**Active Error Flag**: 6 consecutive dominant (0) bits (violating rule of bit stuffing)

# CAN bit stuffing

- In CAN frames, stuff bits are added from SOF through the end of CRC
  - To ensure enough transitions to maintain synchronization
  - Necessary due to the **non-return-to-zero (NRZ)** coding used
- After every 5 identical bits (dominant or recessive) a **complementary** bit is inserted
  - These stuff bits are not calculated into the checksum
  - The stuffed data frames are destuffed by the receiver
- In the fields where bit stuffing is applied, 6 consecutive bits of the same polarity (i.e. 111111 or 000000) are considered an error
- Node can transmit an Active Error Flag when an error has been detected
- Bit stuffing -> increase data frames size
  - E.g. 11111000011110000...
  - is stuffed as (stuffing bits in red): 11111**0**0000**1**1111**0**0000**1**...
    - Bit stuffing itself may be the first of the five consecutive identical bits
    - Worst case: 1 stuffing bit per 4 original bits

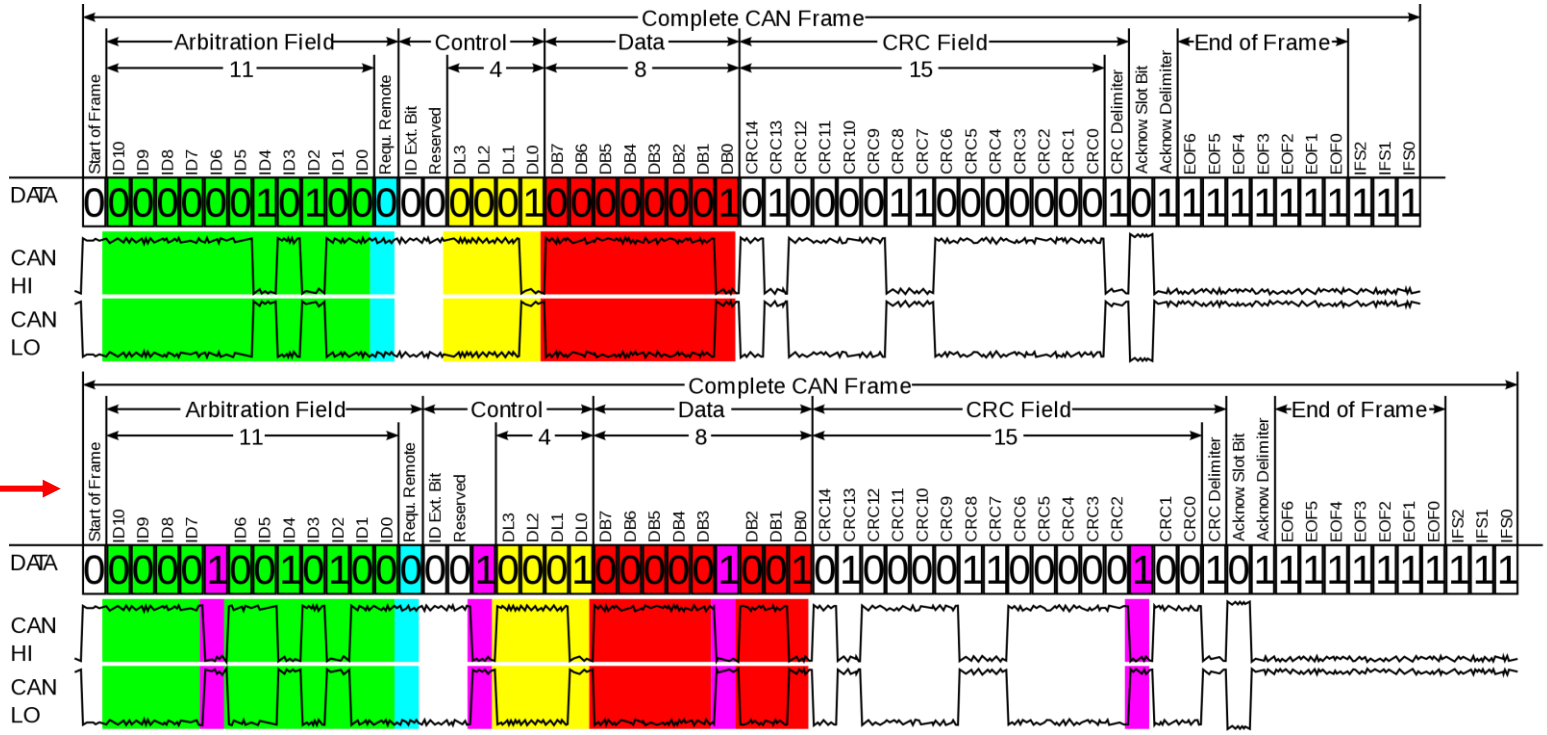
**NRZ (Non Return to Zero):** the binary signals to be transmitted are mapped directly: a logic “1” to a high level, a logic “0” to a low level.



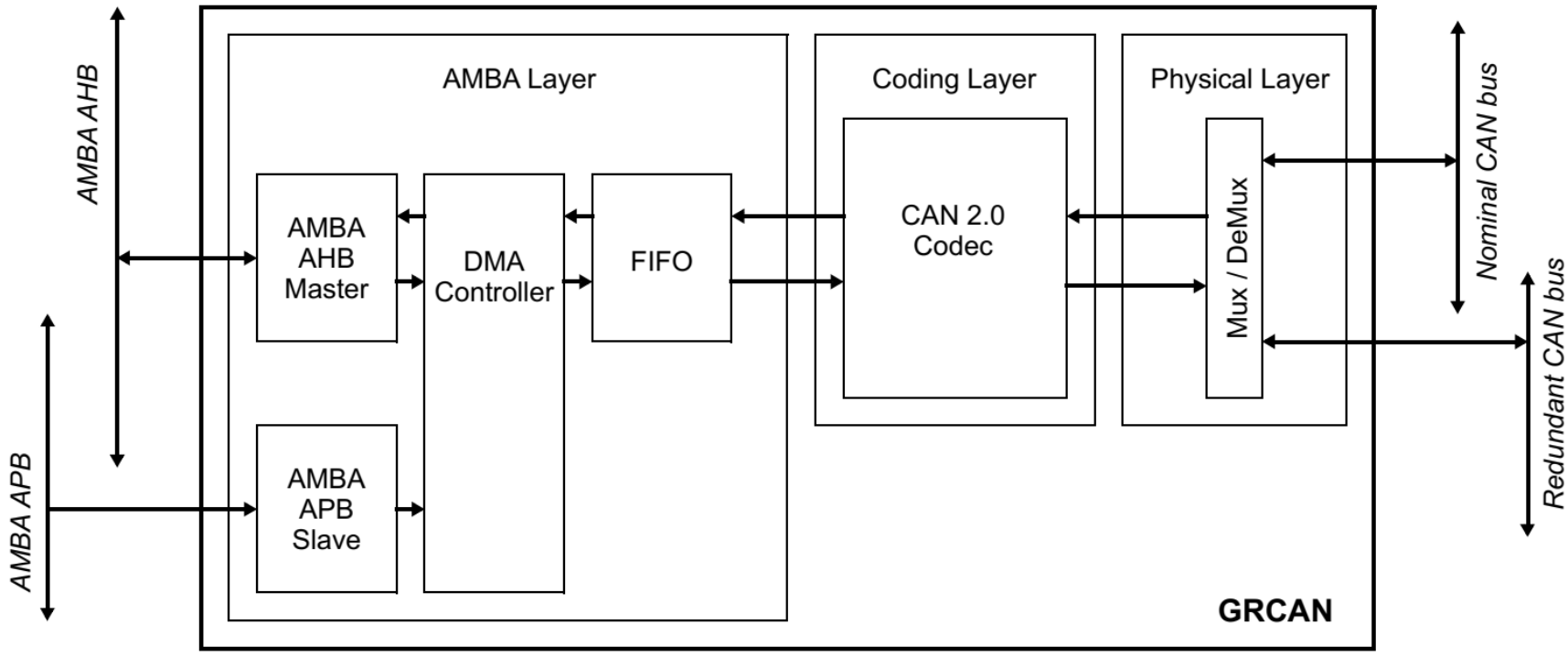
# CAN bus bit stuffing example

CAN frame

CAN frame after bit stuffing (in purple). An incorrect CRC is used for bit stuffing illustration purposes.



## 42 GRCAN - CAN 2.0 Controller with DMA



# CAN vs $I^2C$

- **Synchronization**
  - $I^2C$ is synchronous
  - CAN is asynchronous
- **Addressing Method**
  - $I^2C$ requires a unique slave address for communication
  - CAN uses identifiers for messages instead of device addresses
- **Communication Orientation**
  - $I^2C$ : node-oriented, meaning communication happens between master and slave nodes
  - CAN: message-oriented, messages are broadcast, and the node interested in the message will pick it up
- **Physical Layer**
  - $I^2C$ uses two lines (SDA for data and SCL for clock)
  - CAN uses a differential bus which makes it more resistant to noise.
- **Speed**
  - $I^2C$ operates at speeds from 100kbps (Std mode) up to 3.4mbps (High-speed mode)
  - CAN operates at speeds from 250kbps up to 1mbps.
- **Noise Immunity**
  - CAN's differential signaling provides better noise immunity compared to I2C
- **Complexity**
  - $I^2C$ : simpler & easier to use for short distance low-speed interconnections between ICs on a PCB
  - CAN: more complex but provides robust communication over longer distances in noisy environments **(i.e. space avionics)**