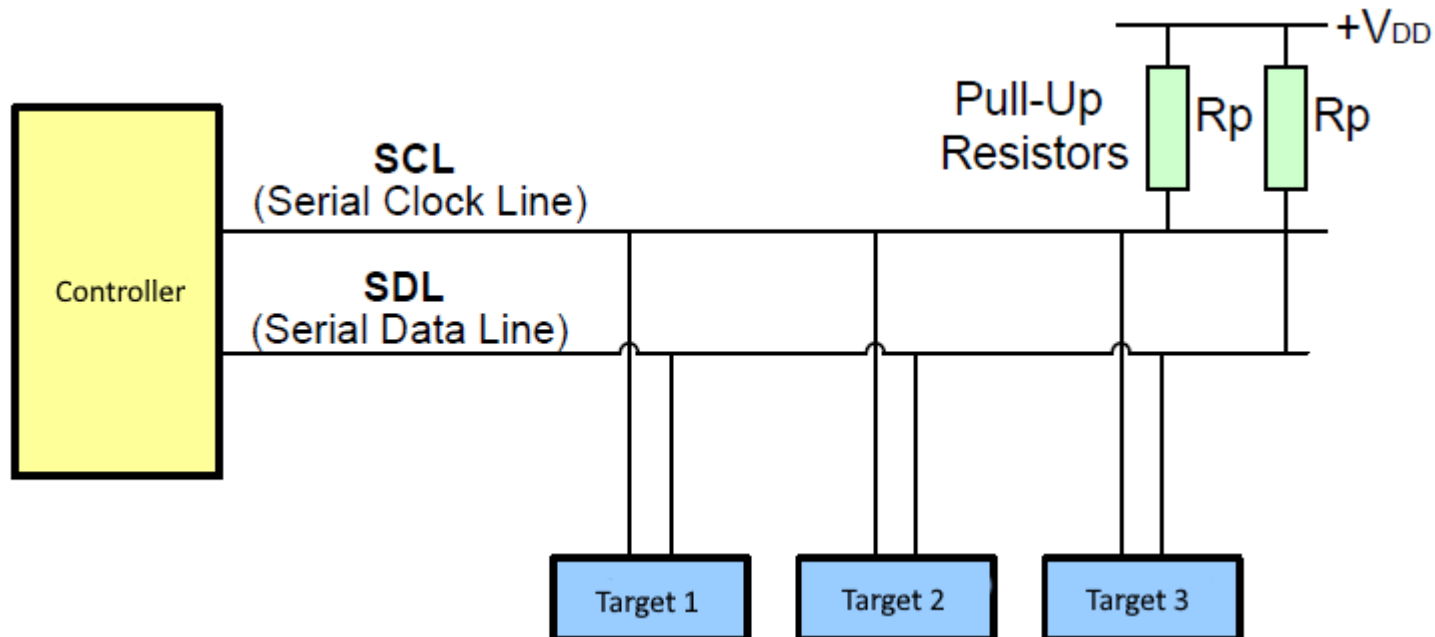


I²C bus basics

- I²C bus is a very popular bus used for communication between a controller (or multiple controllers) and a single or multiple target devices
 - A typical I²C bus for an embedded system has a single controller (microcontroller) and one or multiple target devices. Target devices can be for example sensors, IO expanders, EEPROM, etc.
- I²C bus consists of two data lines: SCL and SDA

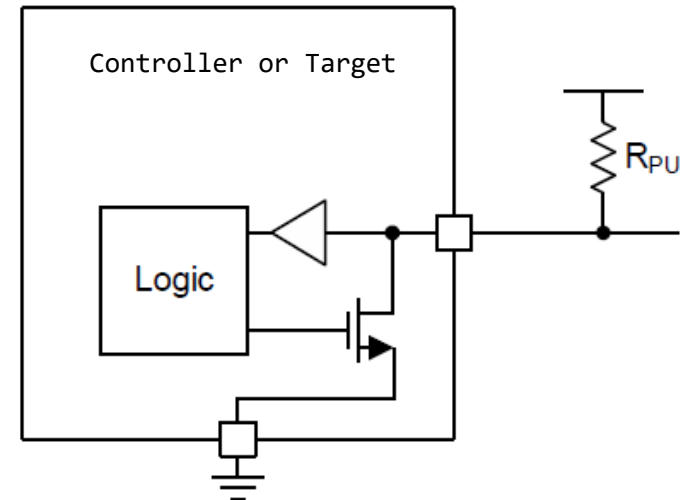


SCL/SDA lines

- SCL is the bus clock signal that is generated by a controller. SCL frequency determines the transfer rate of the bus
- SDA is a bidirectional signal for data transfer. SDA can be driven by a controller or a target depending on the direction
- Both lines are implemented as open drain outputs with an input buffer connected to the same line which allows bidirectional data flow over a single data line

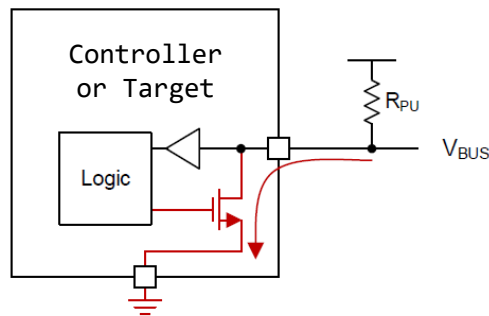
Open-drain output/input

- The figure shows basic internal structure of SCL/SDA line
- Open-drain output can pull the push down to ground or "release" the bus and let the pull-up resistor pull the line high
 - To send a zero the line is pulled low
 - To send a one the line is released
- Open-drain output has two benefits:
 - If two devices try to drive the line to different values there is no short from power rail to ground. The device that drives bus low "wins"
 - If all devices are inactive the bus is still in a known state (pull-up takes the signal high)

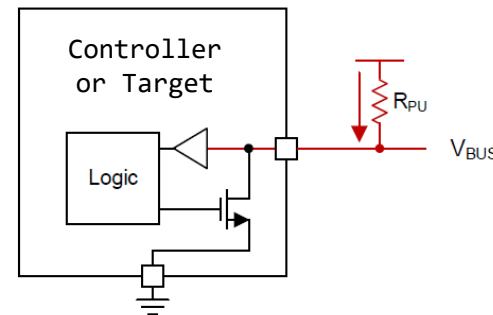
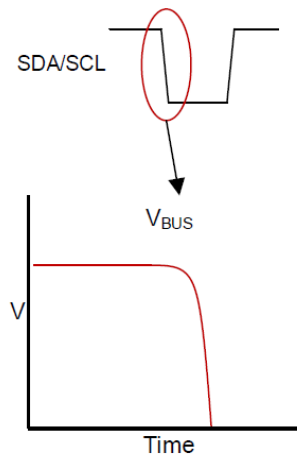


Pulling the bus low with open-drain interface

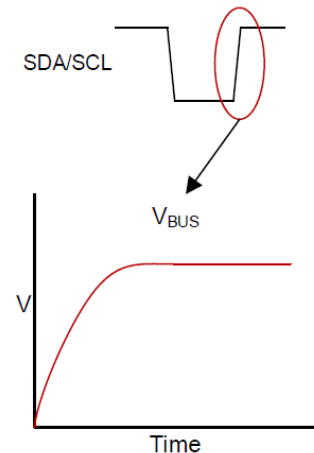
- When a device wants to transmit zero it activates the FET which will provide a low impedance path to ground (can be thought as a short to ground) pulling the line low



Pulling the bus low

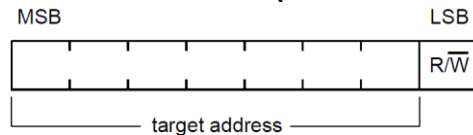


Released bus is pulled high by the pull-up



General operation

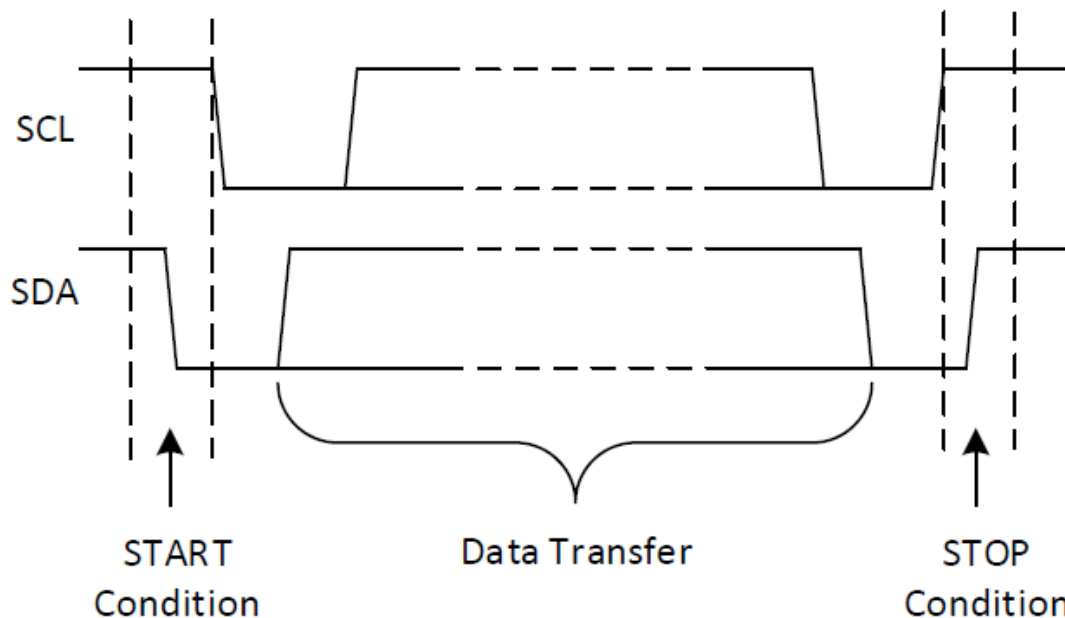
- Controller initiates all communication (and drives the clock)
- A target may not transmit data unless it has been addressed by the controller
- Data/address is transferred in 8-bit units
- Each device on the I²C bus has a 7-bit device address
 - When a device is addressed on the bus read/write bit is appended to the address ($R/W = 1 \rightarrow \text{read}$, $R/W = 0 \rightarrow \text{write}$)



- The address must be unique within the bus (two devices with the same address are not allowed)
- The number of bytes to transfer (read/write) is not limited by the standard
 - Devices may have their own limitation in the amount they can receive or transmit

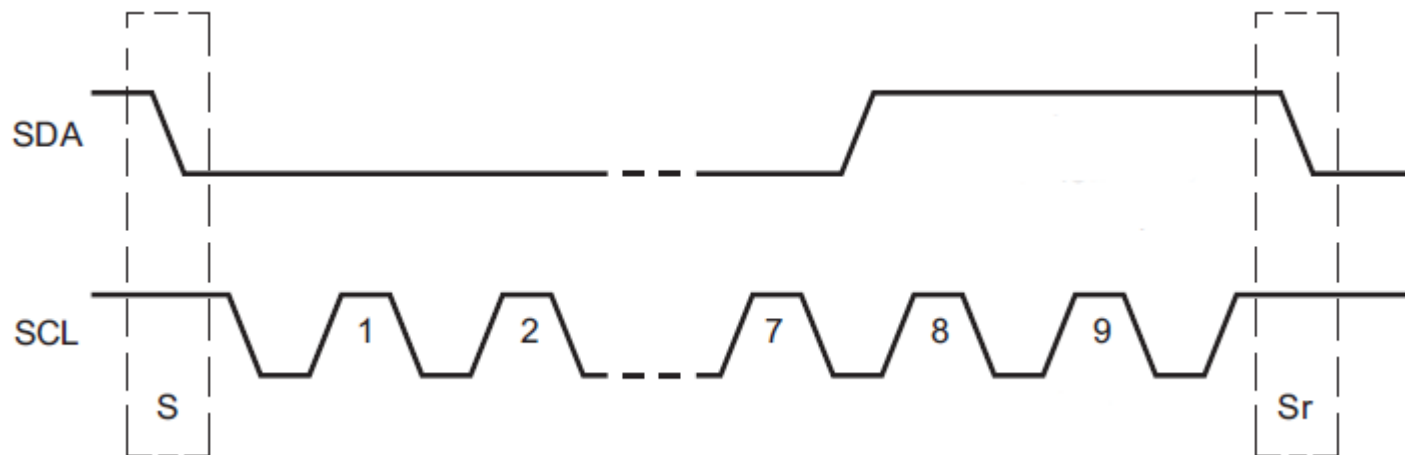
START and STOP conditions

- Data transfer may be initiated only when the bus is idle
 - Bus is considered idle if both SCL and SDA lines are high after a STOP condition
- Transfer start and end is indicated with START and STOP conditions
 - START – high to low transition on SDA while SCL is high
 - STOP – low to high condition on SDA while SCL is high

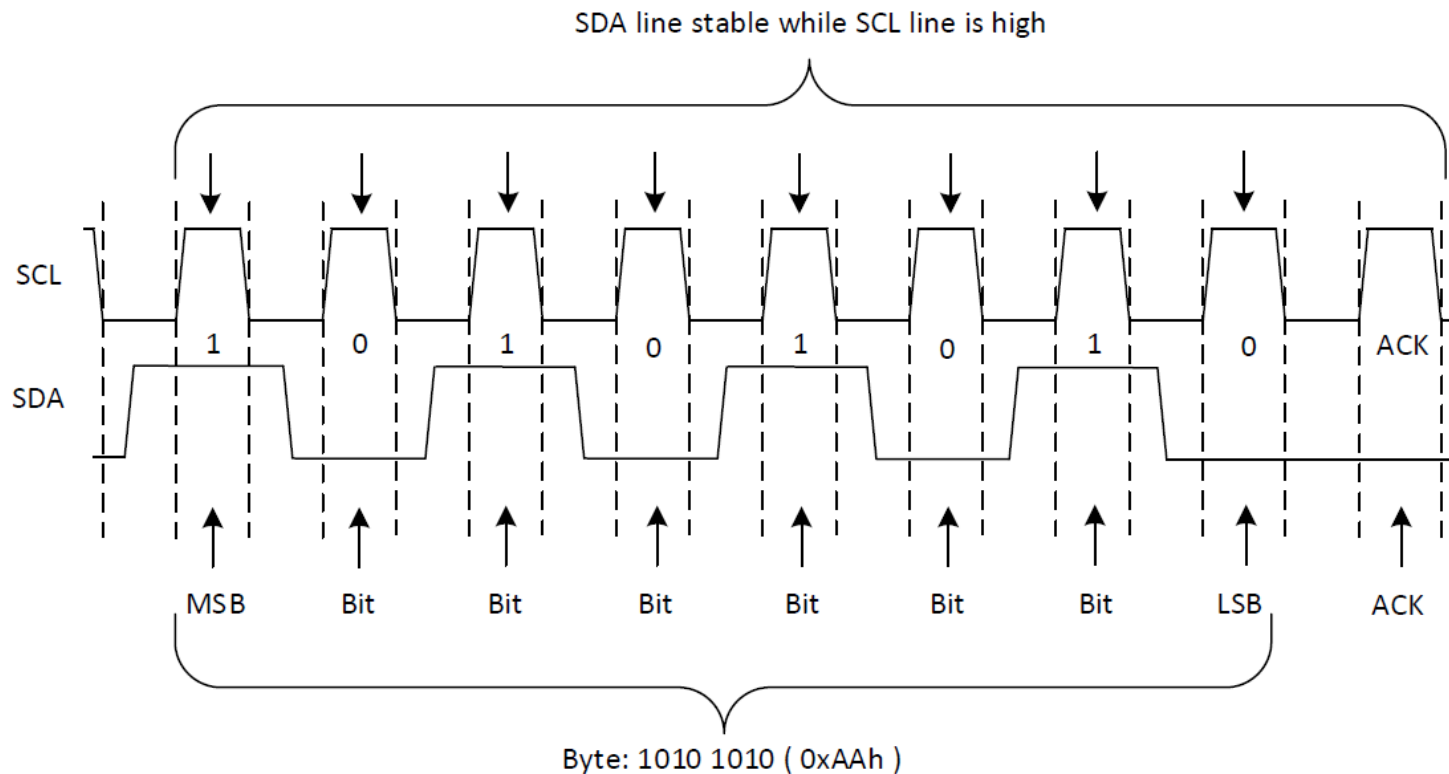


Repeated START condition

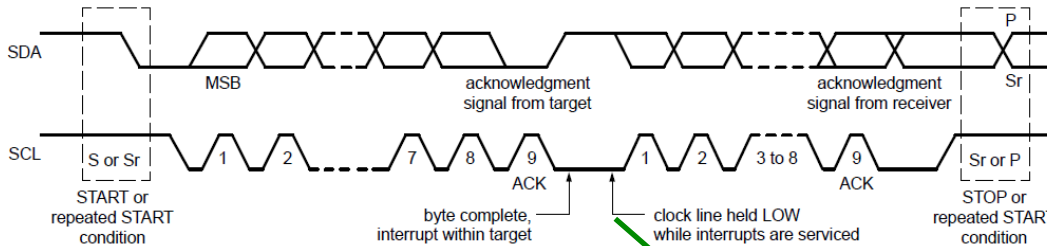
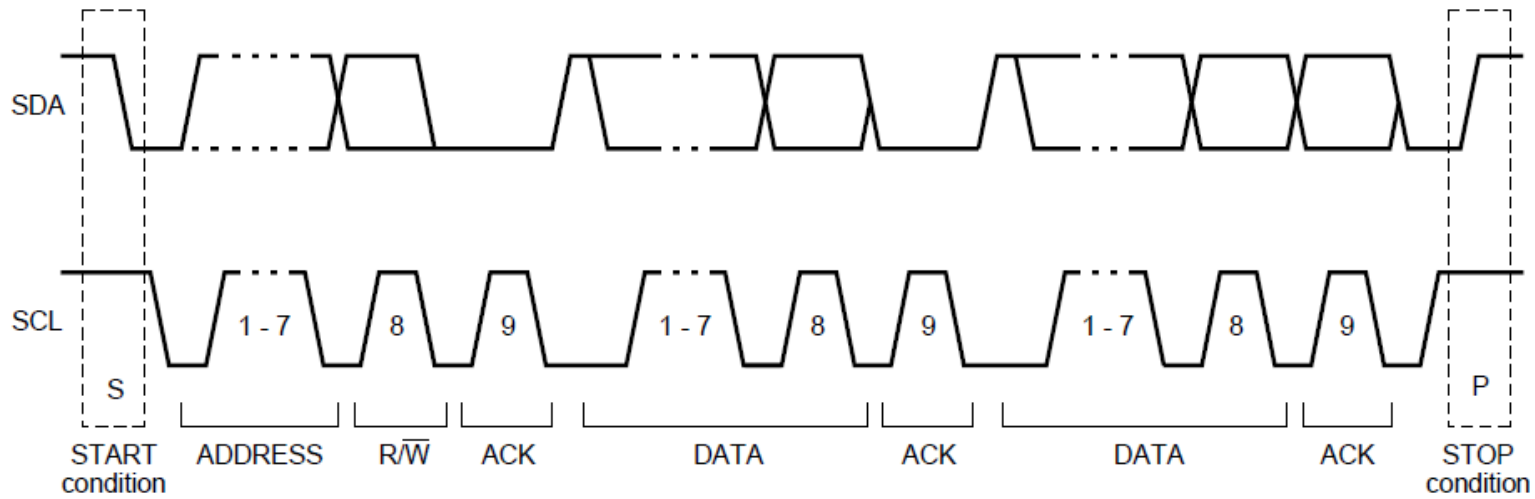
- A repeated START condition is similar to START condition. The signaling looks identical but differs from START because it happens before a STOP condition (when bus is not idle)
- A repeated start condition is used when a controller wants to start a new communication without letting the bus go idle



Example of single byte data transfer



Example of complete data transfer



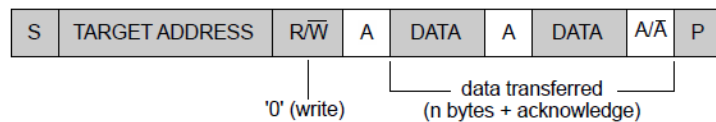
A NACK after an address is sent means no target responded to that address


Clock stretching
Optional, most devices are not capable of clock stretching


Writing to a target on the I²C bus

Writing to a target on the I²C bus

- Target acknowledges data by pulling the bus low after each byte for duration of one bit



 from controller to target

 from target to controller

A = acknowledge (SDA LOW)

\bar{A} = not acknowledge (SDA HIGH)

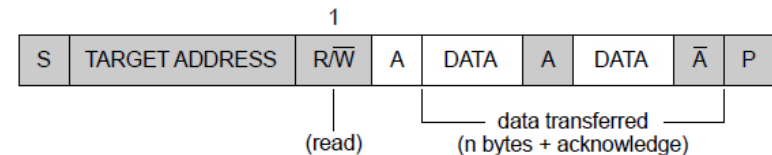
S = START condition

P = STOP condition

- A NACK after write data means the target either did not recognize the command, or that it cannot accept any more data

Reading from a target on the I²C bus

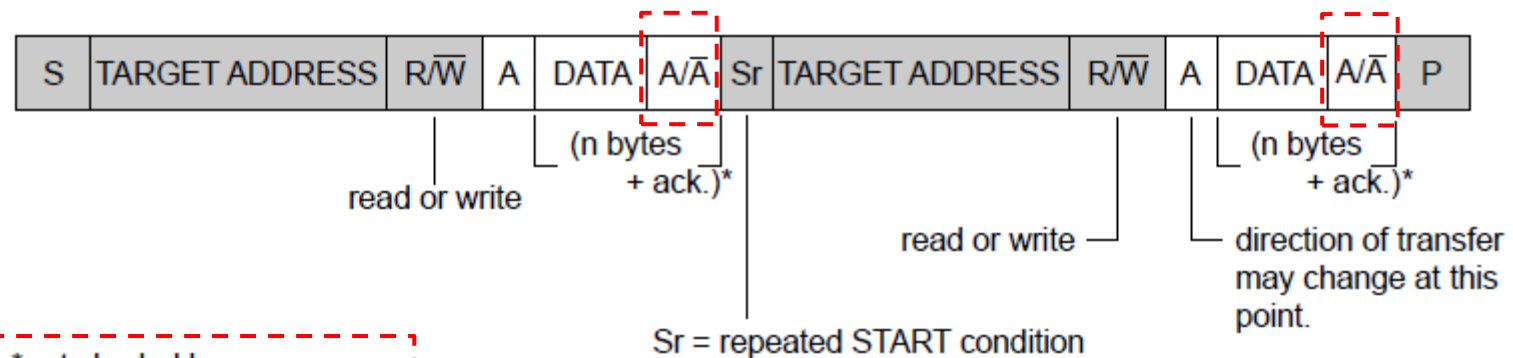
- Controller uses NACK to end write and then issues STOP condition



- A NACK during read data means the controller does not want the target to send any more bytes

Combined I²C transfer

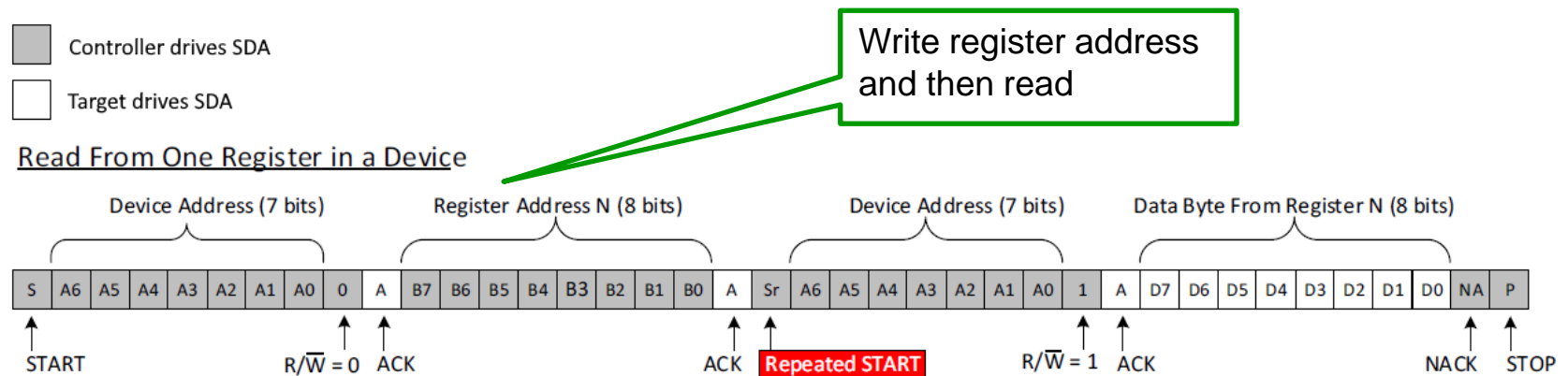
- I²C bus is very flexible when it comes to sending/receiving data
 - Transaction always starts with the device address – the rest is up to the user
- We can generalize typical transactions to the combined transfer shown below
 - Usually write comes first in a combined transaction. Writing first allows us for example, to set a register address from which to read in the second transaction



*not shaded because transfer direction of data and acknowledge bits depends on R/W bits.

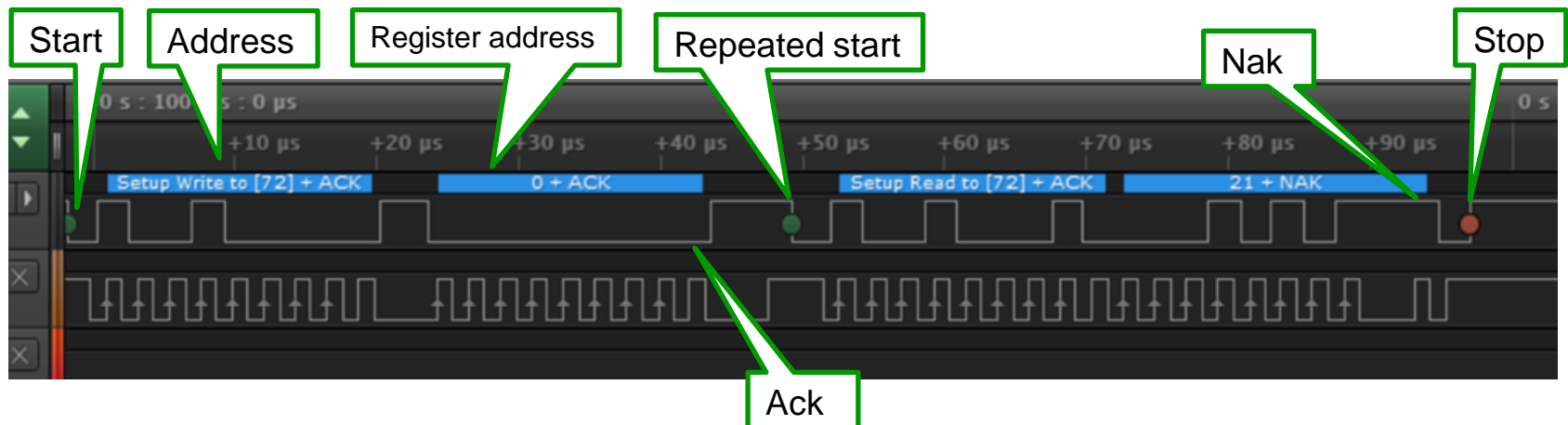
Reading from a target on the I²C bus

- The internal organization of devices is not part of the bus standard
 - Device manufacturers have their own protocols built on top of I²C bus specification
- Most devices operate so that the first byte(s) written are considered register address
 - Following byte(s) are written to the register set by write
 - Repeated start and read immediately after writing the register address reads the byte(s) from the register set by write



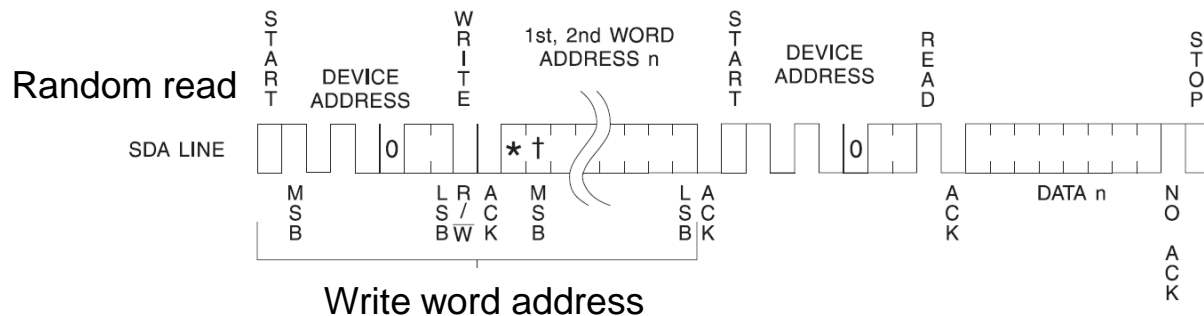
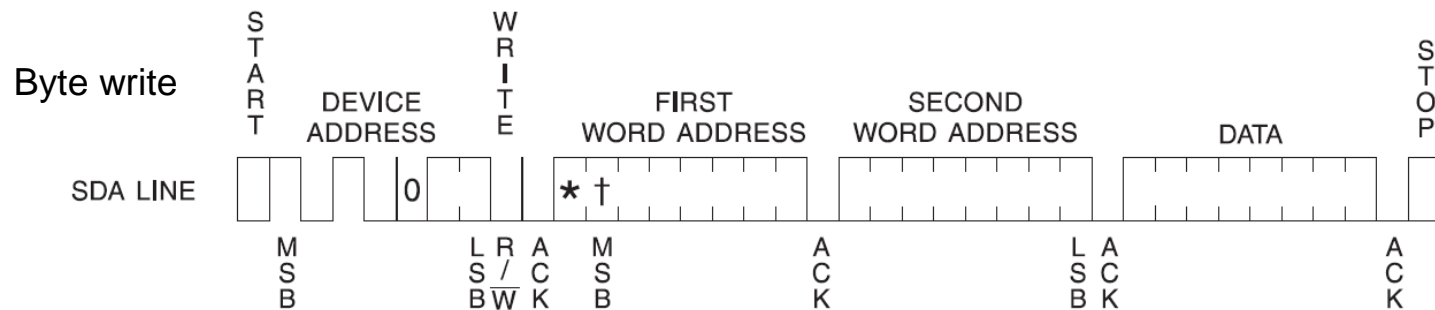
Example of Combined I²C transfer

- Below is a logic analyzer capture of an I²C transaction
 - A temperature sensor where reading register 0 returns the current temperature
- The bus standard specifies only how a data transfer takes place on the bus
 - What the data actually means or what you need to read or write can be found in the device data sheet
- All devices do not have multiple registers. In case of a single register the data can be written directly after the device address
 - Always check the device datasheet for communication details



Example of device access

- The following images are taken from the data sheet of AT24C256 which is a 256 Kbit (32 Kbyte) EEPROM
- This type of communication is industry standard for I²C memory chips – the number of address bytes varies with the size of the device. Devices that are larger than 64 Kbyte require three address bytes



Notes: (* = DON'T CARE bit)
(† = DON'T CARE bit for the 128K)

PicoSDK I2C-functions

- `uint i2c_init (i2c_inst_t *i2c, uint baudrate);`
 - Initialise the I2C HW block.
- `int i2c_write_blocking (i2c_inst_t *i2c, uint8_t addr, const uint8_t *src, size_t len, bool nostop);`
 - Attempt to write specified number of bytes to address, blocking.

Annotations for `i2c_write_blocking`:

 - `uint8_t addr`: Device address
 - `const uint8_t *src`: Data to write
 - `size_t len`: Number of bytes, must be >0
 - `bool nostop`: Do not create stop condition. Next transaction will do repeated start.
- `int i2c_read_blocking (i2c_inst_t *i2c, uint8_t addr, uint8_t *dst, size_t len, bool nostop);`
 - Attempt to read specified number of bytes from address, blocking.