UART

- Universal Asynchronous Receiver/Transmitter
- A common integrated feature in most microcontrollers
- Takes bytes of data and transmits the individual bits in a sequential fashion. At the receiving end (a second UART) re-assembles the bits into complete bytes
- Serial transmission requires only one signal wire (+ground) which makes it more cost effective than parallel transmission. Serial transmission allows much longer wire than parallel transmission.
- Asynchronous transmission allow data to be transmitted without a dedicated clock line from sender to receiver
    - Sender and receiver must agree on timing parameters (bits/second) and transmission format (number of bits, number of syncronization bits)

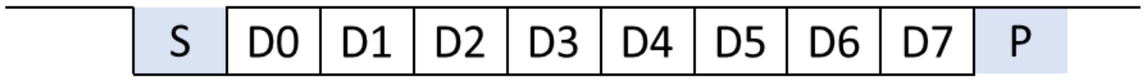
UART protocol

- In the absence of a clock signal the timing is based on bit length which must be known by both of the communicating parties
- Common data rates are 9600 bps, 38400 bps and 115200 bps
- Timing is based on start and stop bits → some overhead in the transmission
- Transmission
  - Sender keeps the line at logical high until transmission starts
  - Transmission starts with a start bit (logical low)
  - Then sender sends a byte of data, one bit a time
  - After one byte the sender sends a stop bit (logical high)
  - Repeat for each byte of data to transmit

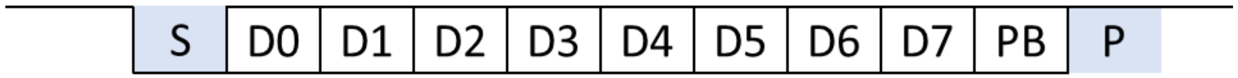
UART timing diagram

- Receiver synchronizes to start bit
- Short syncronization interval allows for some jitter in timing
  - Resynchronize on every start bit
- Transmission is essentially a state machine with states: Idle, Start, D0, D1, D2, D3, D4, D5, D6, D7, Stop
- The number of data bits can be 5 to 9 and there can also be a parity bit and the number of stop bits can be 1 or 2
  - Typically the number of data bits is 8
  - The most common setting is 8,N,1 (8 data bits, no parity bit, 1 stop bit) for general data transfer

8,N,1 data frame



8,E,1 data frame



UART timing

- UART timing is derived from the system clock with clock dividers
- To set up the timing you need to know:
  - System clock rate (some processors have a separate peripheral clock)
  - Transmission bit rate
- Internally most UARTs use 16x clock for accurate start bit detection and centering of sampling points
- The formulas to calculate the dividers can be found in the data sheet

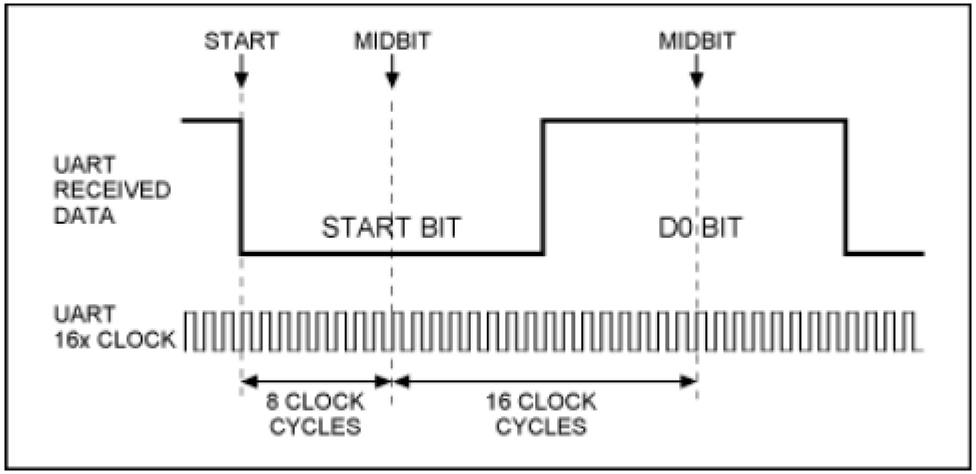
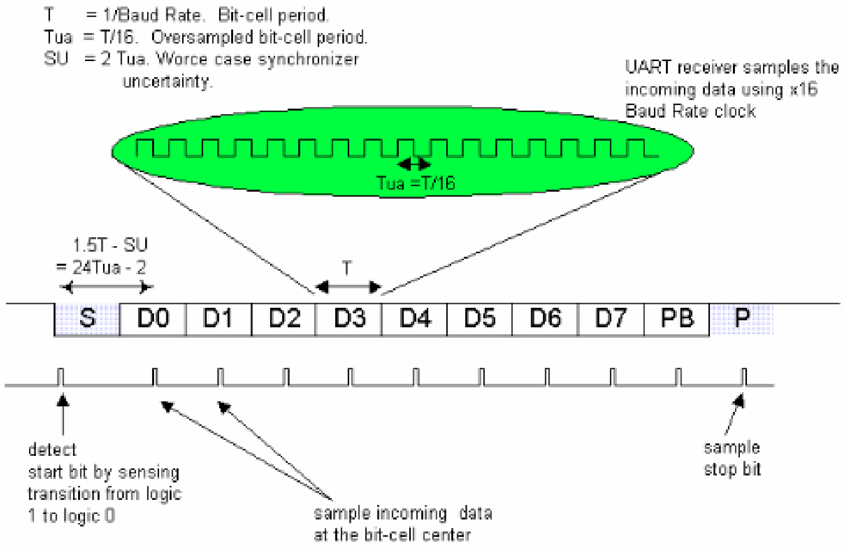
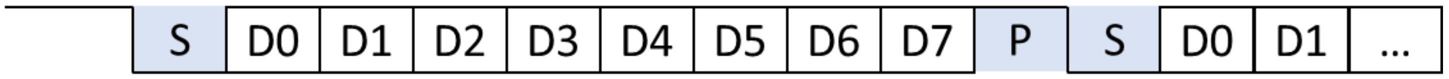


Figure 2. UART receive frame synchronization and data sampling points.
Image source: Application Note 2141: http://www.maximintegrated.com/an2141


T = 1/Baud Rate. Bit-cell period.
Tua = T/16. Oversampled bit-cell period.
SU = 2 Tua. Worce case synchronizer uncertainty.
UART receiver samples the incoming data using x16 Baud Rate clock
Tua =T/16
1.5T - SU = 24Tua - 2
T
S
D0
D1
D2
D3
D4
D5
D6
D7
PB
P
detect start bit by sensing transition from logic 1 to logic 0
sample incoming data at the bit-cell center
sample stop bit
Image source: https://tutorial.cytron.io/2012/02/16/uart-universal-asynchronous-receiver-and-transmitter/

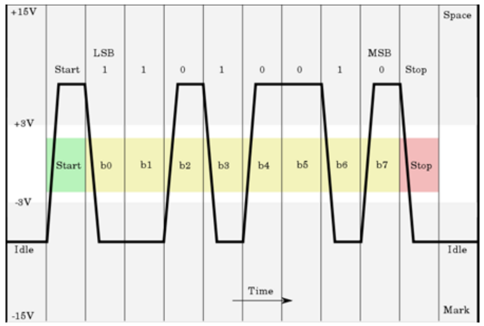
Transmission overhead and net data rate

- The minimum overhead per transmitted byte is 2 bits
- Typical 8,N,1 framing sends 10 bits per byte
  - The (maximum) net data rate = UART bit rate / 10
    - 9600 bps → 960 bytes/s
    - 115200 bps → 11520 bytes/s



RS-232 voltage levels

- RS-232 (Recommended Standard 232) is a standard for serial binary data signals connecting between a Data Terminal Equipment (DTE) and a Data Communication Equipment (DCE).
  - It is commonly used in computer serial ports.
  - One of the significant differences between TTL level UART and RS-232 is the voltage level
- Modern laptops don’t have a built-in RS-232 serial port – USB-converters are typically used
  - Converters either use RS-232 voltage levels or TTL levels
    - TTL level converter can be connected directly to MCU pins
    - RS-232 converter requires a voltage converter on the MCU side



Logic	Voltage
Low	+3 – +15V
High	-3 – -15V

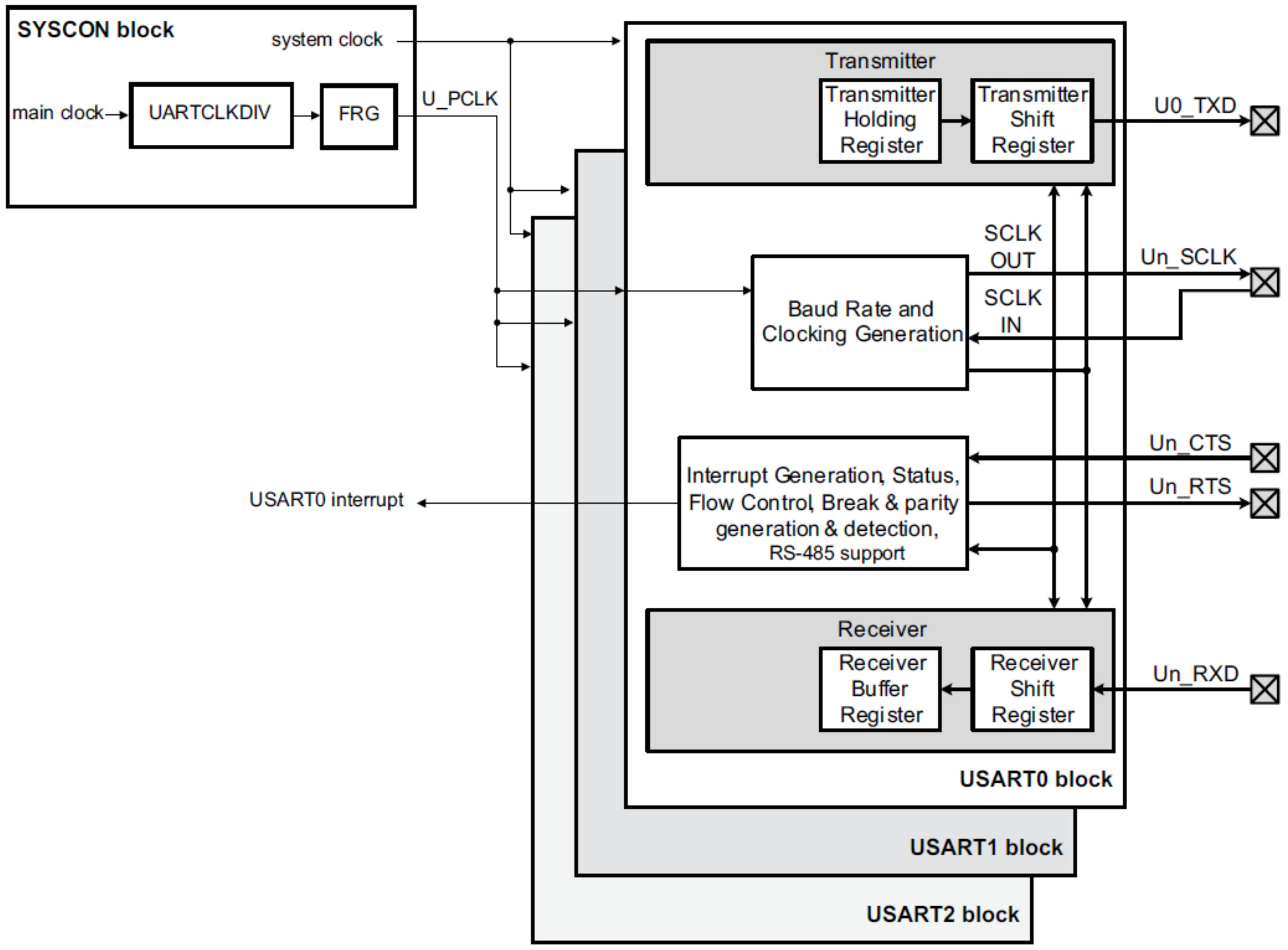
Typical USB UART converters

RS-232 → D9 connector

TTL → wires/pin header

Source: https://tutorial.cytron.io/2012/02/16/uart-universal-asynchronous-receiver-and-transmitter/

LPC1549 USART



UART operating principle

- Data is written to THR (Transmit Holding Register)
  - Data may only be written if transmit holding register is ”empty”
  - THR status (empty/full) can be read from UART status register
  - THR becomes ”empty” when data is copied to transmit shift register
- Received data goes RBR (Receive Buffer Register)
  - Data may be read only when receive buffer is ”full”
  - RBR status can be read from UART status register
  - RBR becomes ”empty” when data is read (copied) to a CPU register (or DMA engine)
  - Overrun occurs if new value is ready in Receive Shift Register but BRB is still ”full” – the newly received character is discarded and overrun bit is set in status register
- UART can be configred to generate an interrupt when status bits are set
  - Which bits generate interrupts is configurable
- Polling works fine as long as you poll often enough

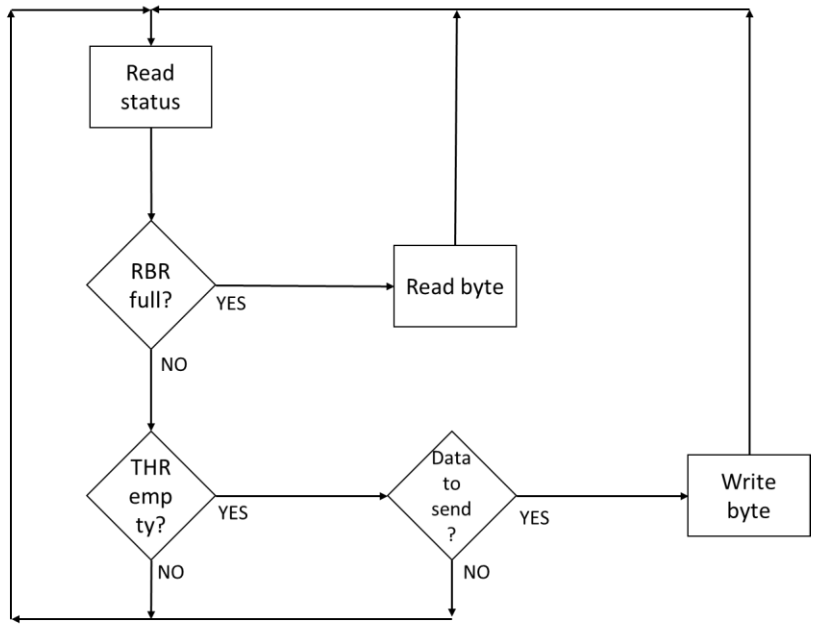
Data rate and interrupt load

- UART can induce a significant interrupt load when data is tranferred at maximum data rate
  - 9600 bps – interrupts at 1 ms intervals → manageable
  - 115200 bps – interrupts at 86 μs intervals → risk of losing data if other interrupts are active or code contains critical sections
- Modern microcontrollers address this in two different ways
  - Buffered UART – for example 16 byte FIFO reduces interrupt load and allows longer ISR response time without losing data
  - DMA based transfer – risk of losing data is minimal
    - DMA transfers are quite challenging to program compared to traditional ISR based transfers
    - LPC1549 can do DMA or interrupt/character ISRs (no UART buffering available)

UART

- The following priciples apply both to ISR driven and polled UART handling

- Prioritize reading over writing
    - Usually, you can buffer/delay your writing, but the only way to prevent overrun is to read characters before the next one is received
- Handshake signals can be used to to tell the sender if it is OK to send or not
    - Handshaking adds complexity to UART handling



Debug UART example

```
Board_UARTPutSTR("\r\nHello, World\r\n");
Board_UARTPutChar('!');
Board_UARTPutChar('\r');
Board_UARTPutChar('\n');
int c;
while(1) { // echo back what we receive
  c = Board_UARTGetChar();
  if(c != EOF) {
    if(c == '\n') Board_UARTPutChar('\r'); // precede linefeed with carriage return
    Board_UARTPutChar(c);
    if(c == '\r') Board_UARTPutChar('\n'); // send line feed after carriage return
    }
 }
```

Note that we read into **int** not char

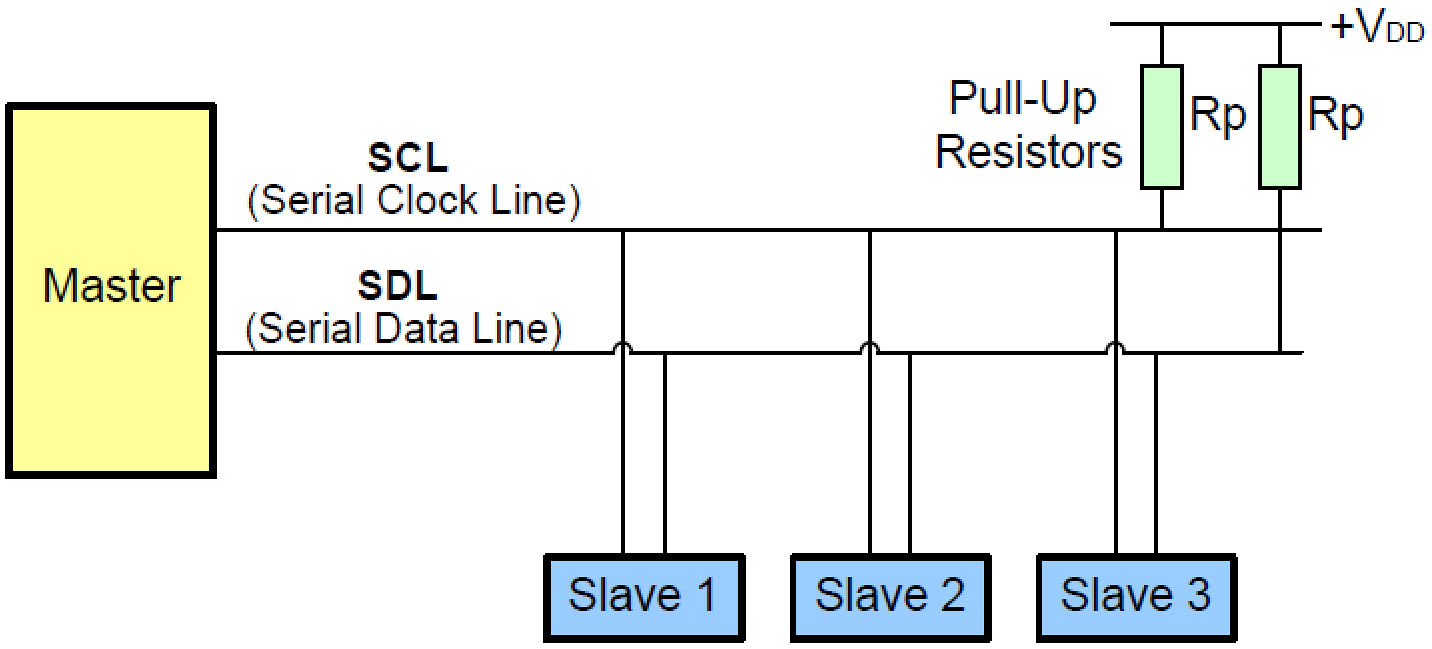
Read one character

Read status → EOF means no character was available

- Note that you are dealing with raw data – there is no backspace or end of line processing
  - What happens when you press enter depends on your terminal program by default PuTTy sends carriage return when you press enter
- Some terminal programs (e.g. PuTTy) send characters as you type – others (e.g. termite) send a whole line when you press enter. The latter requires polling at maximum receive rate (9600 bps → must poll every ms)

I^{2}C bus basics

- I^{2}C bus is a very popular bus used for communication between a master (or multiple masters) and a single or multiple slave devices
  - A typical I^{2}C bus for an embedded system has a single master (microcontroller) and one or multiple slave devices. Slave devices can for example sensors, IO expanders, EEPROM, etc.
- I^{2}C bus consists of two data lines: SCL and SDA

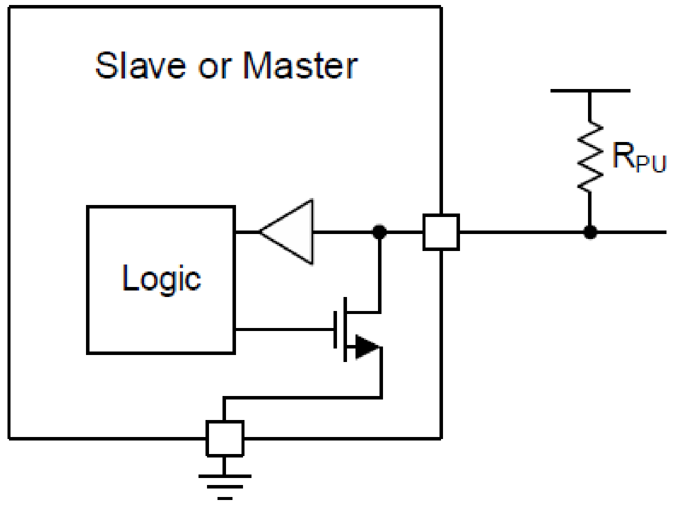


SCL/SDA lines

- SCL is the bus clock signal that is generated by a master. SCL frequency determines the transfer rate of the bus
- SDA is a bidirectional signal for data transfer. SDA can be driven by a master or a slave 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

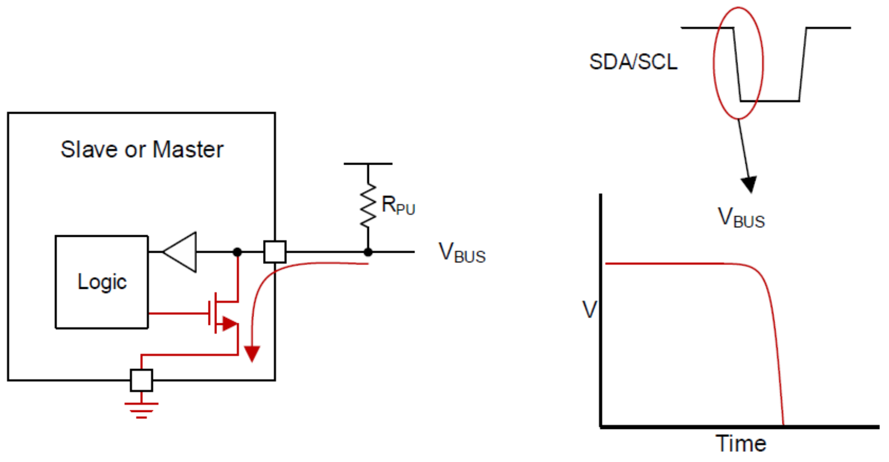
Open-drain output/input

- 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)

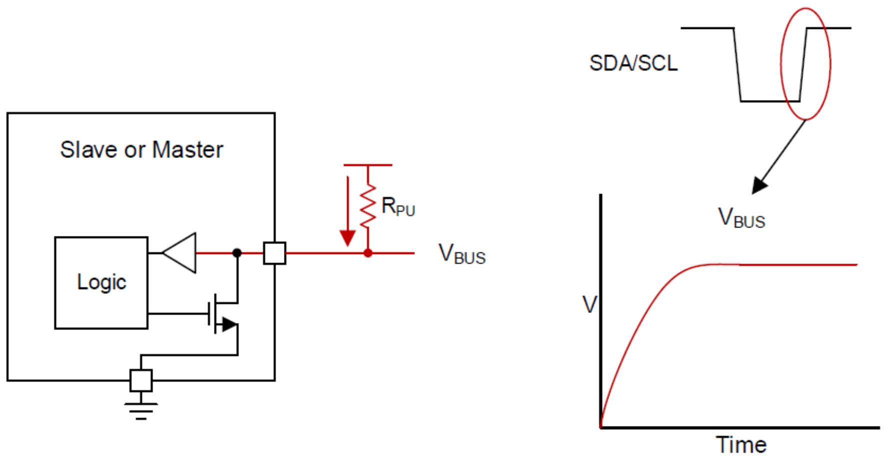


Pulling the bus low with open-drain interface

- When 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



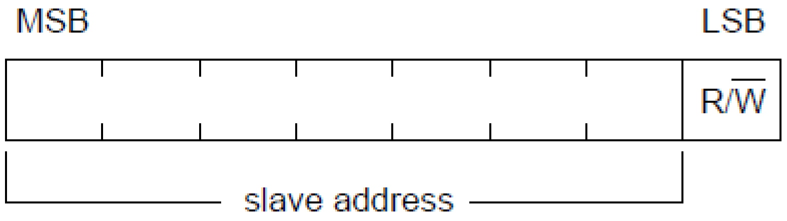
Pulling the bus low



Relased bus is pulled high by the pull-up

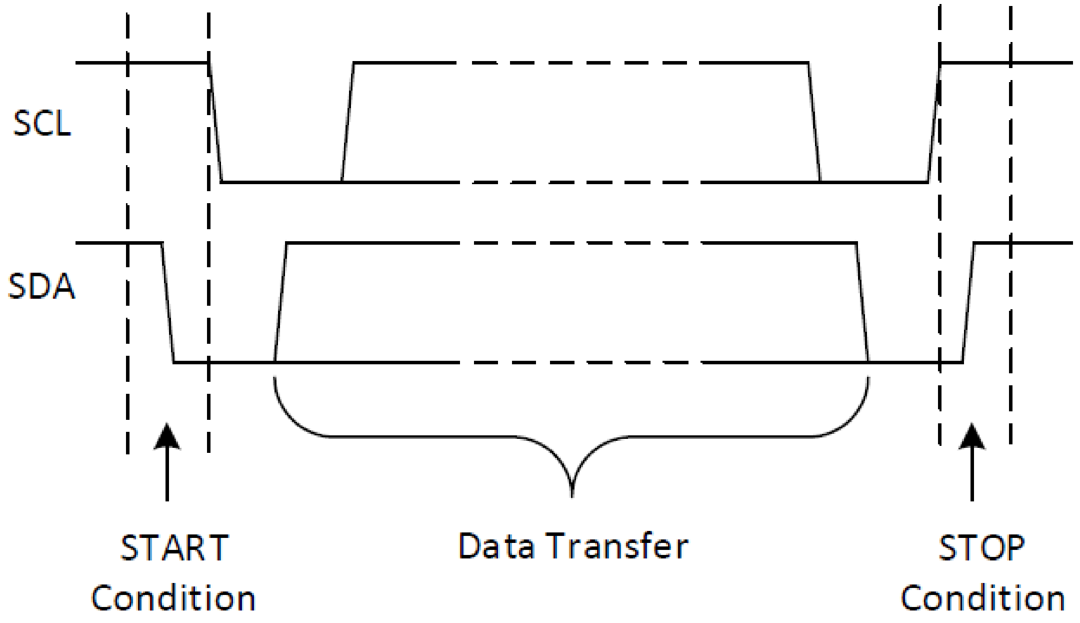
General operation

- Master initiates all communication (and drives the clock)
- A slave may not transmit data unless it has been addressed by the master
- Each device on the I^{2}C bus has a 7-bit device address
  - When device is addressed on the bus read/write bit is appended to the address (R/W = 1 → read, R/W = 0 → write)
- The address must be unique within the bus (two devices with the same address are not allowed)
- A device can have one or multiple registers where data is stored, written, or read
- Data/address is transferred in 8-bit units



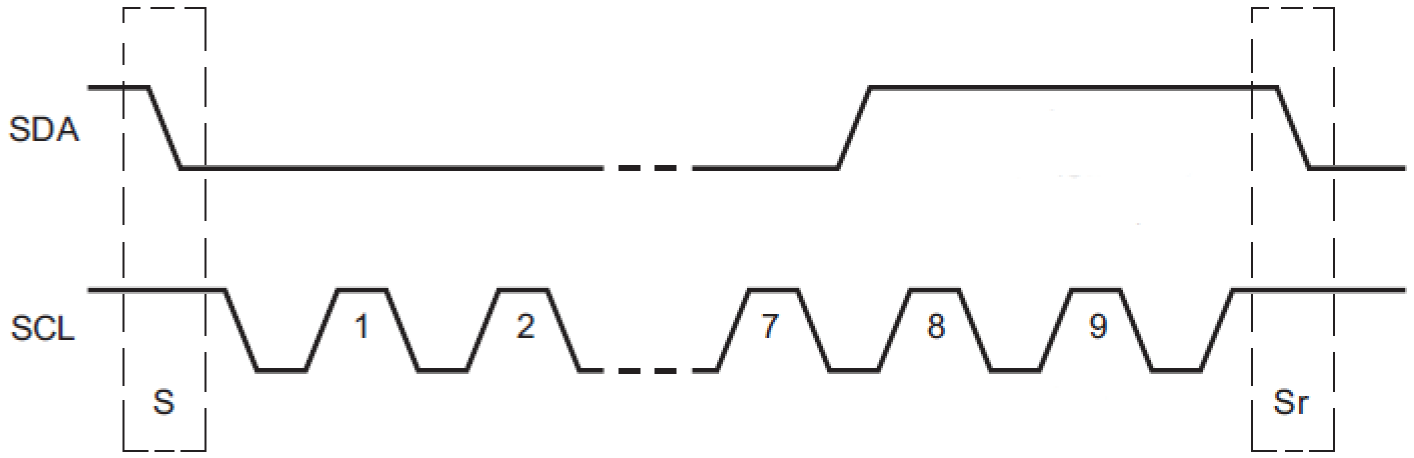
START and STOP conditions

- 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

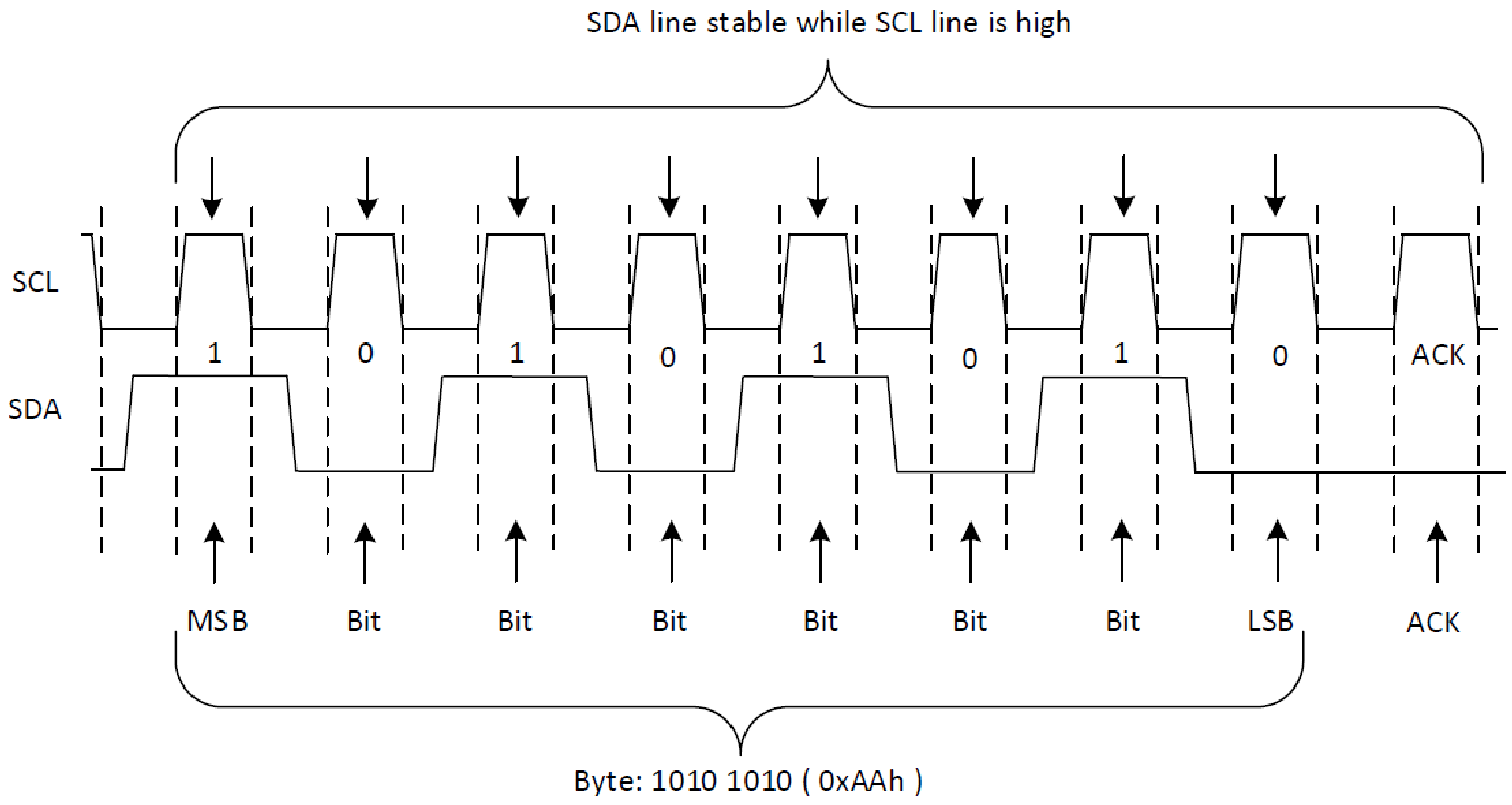


Repeated START condition

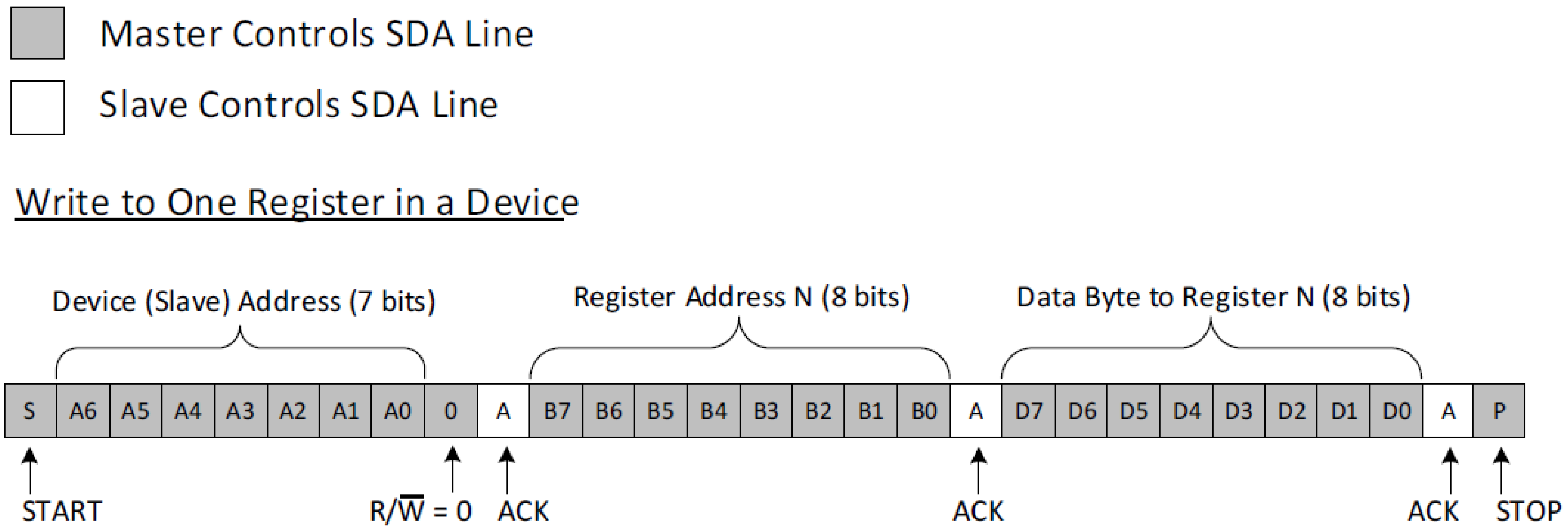
- A repeated START condition is similar to START condition. The signaling looks identical but differs from START because it happens before a STOP condition (when bus is not idle)
- A repeted start condition is used when a master wants to start a new communication without letting the bus go idle



Example of single byte data transfer


SDA line stable while SCL line is high
SCL
SDA
1
0
1
0
1
0
1
0
ACK
MSB
Bit
Bit
Bit
Bit
Bit
Bit
LSB
ACK
Byte: 1010 1010 (0xAAh)

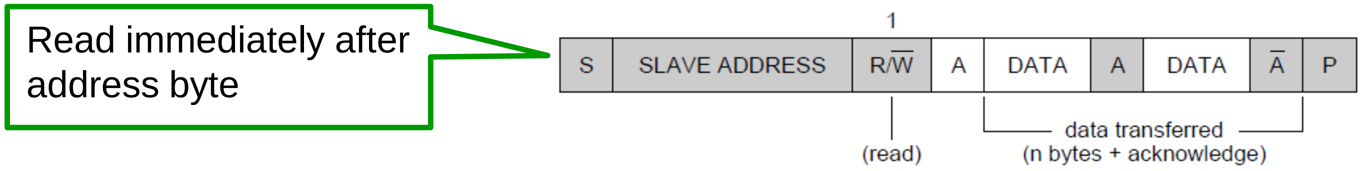
Writing to a slave on the I²C bus

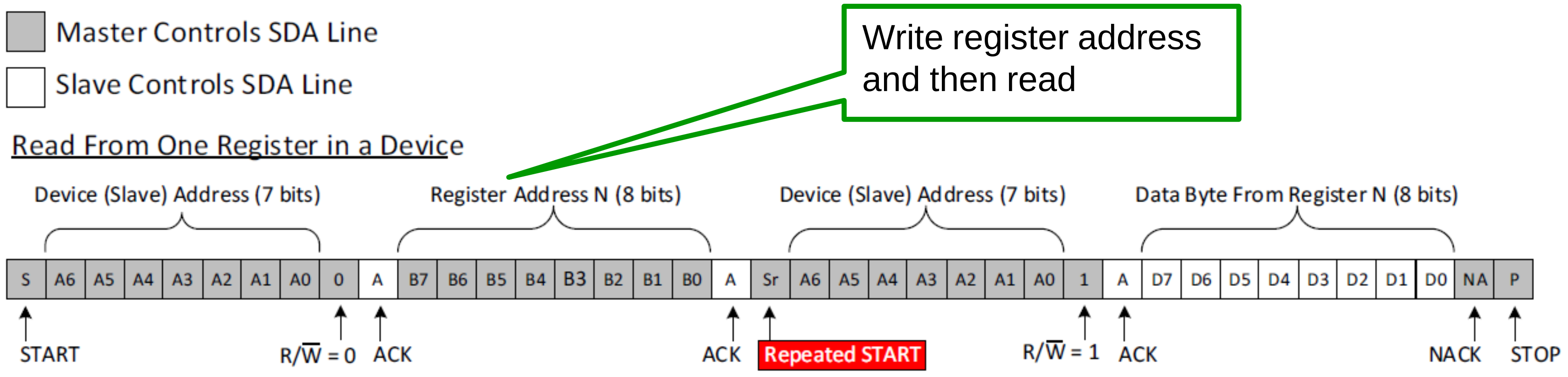


- Slave acknowledges data by pulling the bus low after each byte for duration of one bit
- All devices do not have multiple registers. In case of a single register the data can be written directly after the device address
  - Always check device datasheet for communication details

Reading from a slave on the I²C bus

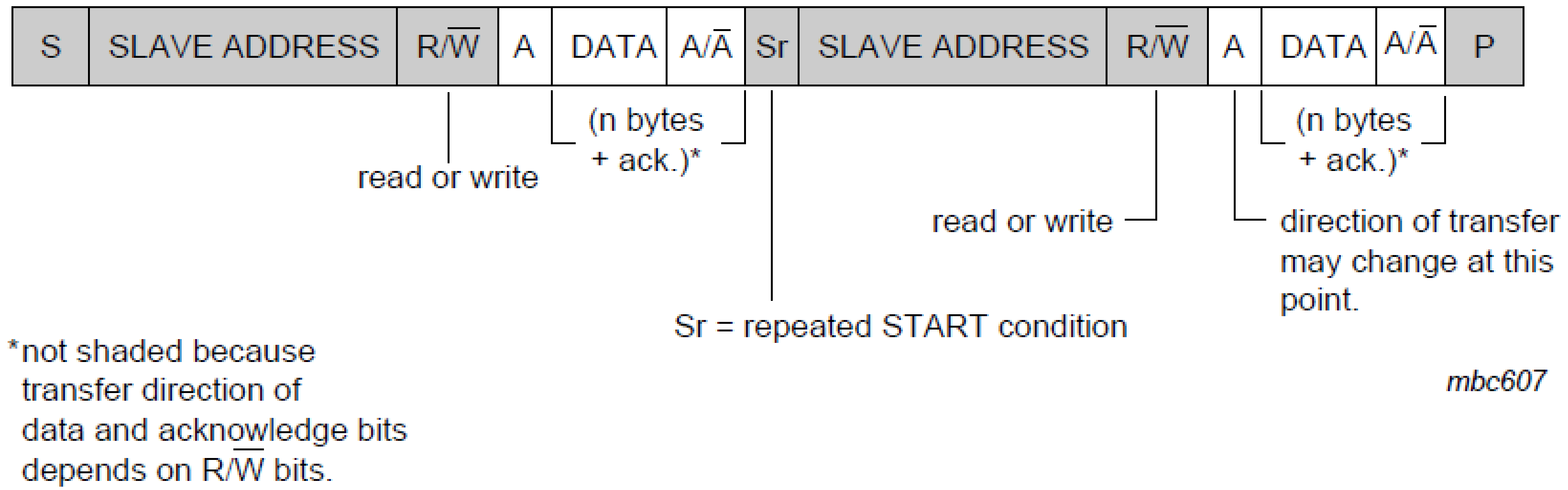
- To read from a specific device register the device is addressed with read bit cleared and the register address is written then a repeated start condition is produced and device is addressed in read mode (read bit set)




▨ Master Controls SDA Line
☐ Slave Controls SDA Line
Read From One Register in a Device
Write register address and then read
Device (Slave) Address (7 bits)
Register Address N (8 bits)
Device (Slave) Address (7 bits)
Data Byte From Register N (8 bits)
S A6 A5 A4 A3 A2 A1 A0 0 A B7 B6 B5 B4 B3 B2 B1 B0 A Sr A6 A5 A4 A3 A2 A1 A0 1 A D7 D6 D5 D4 D3 D2 D1 D0 NA P
START
R/W̄ = 0
ACK
ACK
Repeated START
R/W̄ = 1
ACK
NACK
STOP

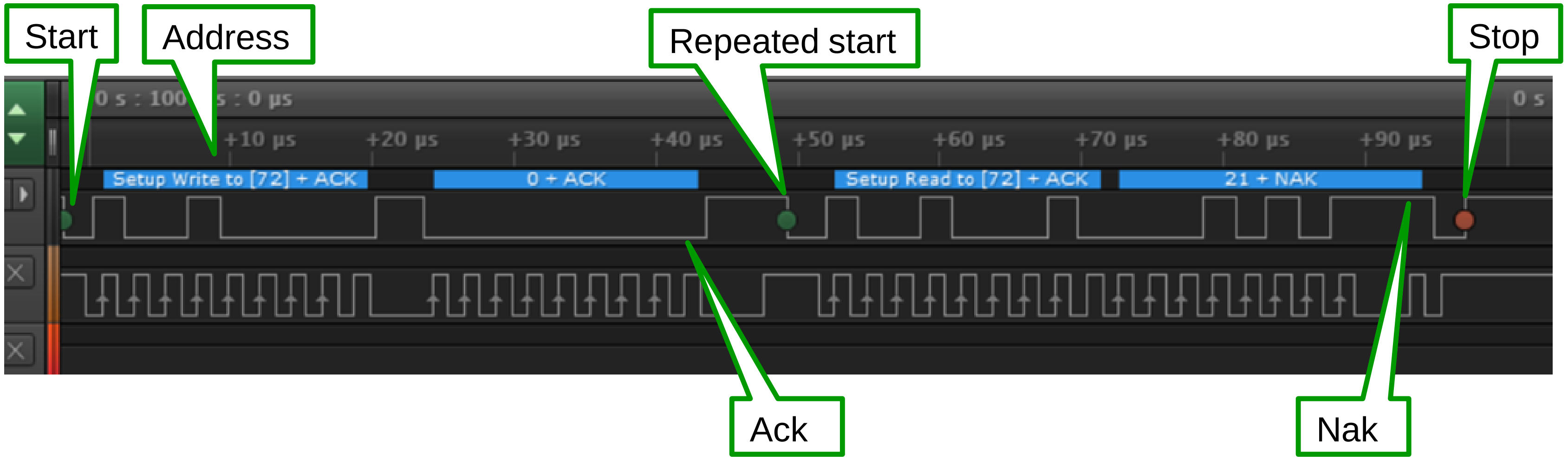
Combined I²C transfer

- I²C bus is very flexible when it comes to sending/receiving data
  - Transaction always starts with the device address – 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 to for example set a register address from which to read in the second transaction


S
SLAVE ADDRESS
R/$\overline{W}$
A
DATA
A/$\overline{A}$
Sr
SLAVE ADDRESS
R/$\overline{W}$
A
DATA
A/$\overline{A}$
P
read or write
(n bytes + ack.)*
read or write
(n bytes + ack.)*
direction of transfer may change at this point.
Sr = repeated START condition
*not shaded because transfer direction of data and acknowledge bits depends on R/$\overline{W}$ bits.
mbc607

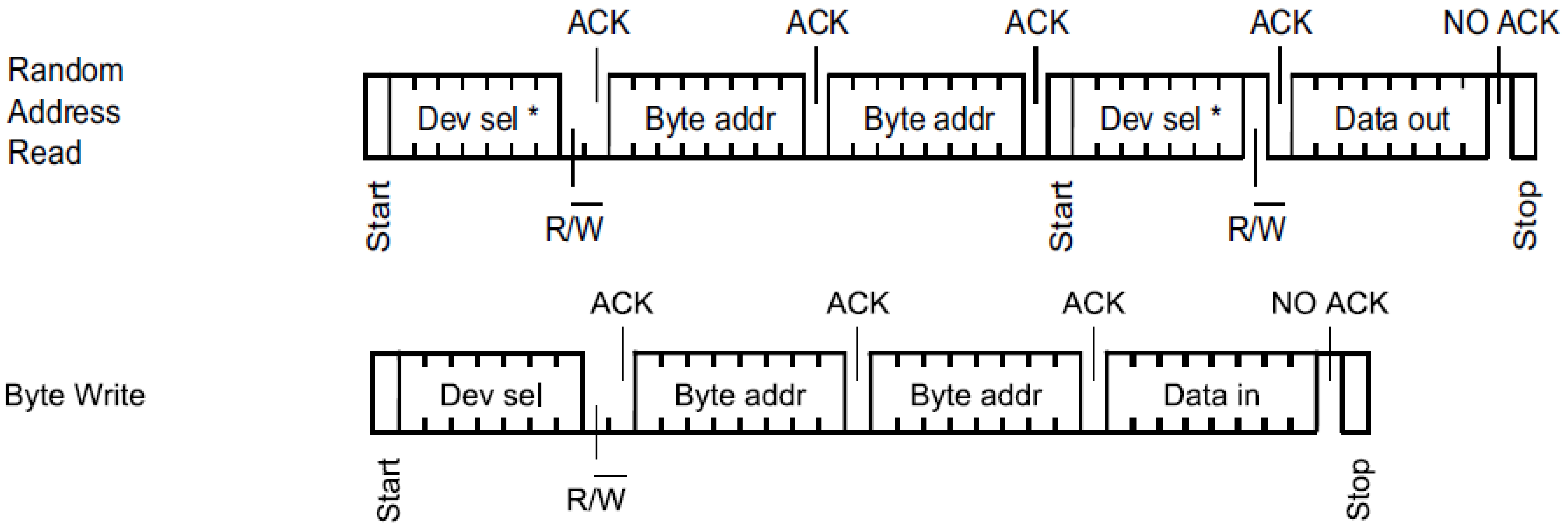
Example of Combined I²C transfer

- Below is a logic analyzer capture of I²C transaction
- The bus standard only specifies how 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



Example of device access

- The following images are taken from the data sheet of M24128-BW which is a 128 Kbit (16 Kbyte) EEPROM
- This type of communication is industry standard for I^{2}C 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



LPC1549 I^{2}C interface

- Hardware handles creating START and STOP conditions, the clocking out data, reading/asserting ACK/NACK
- Hardware does not implement a full I^{2}C state machine
  - Reacting to the status changes and deciding which action to perform next must be handled by the software
    - Interrupt driven/polled implementation
  - NXP hardware is fully compliant to the standard (since I^{2}C was developed by NXP)
  - Writing the state machine requires good understanding of the protocol (or patience to read the documents...)
- LPC Open comes with I^{2}C driver
  - There is a bug in the driver that breaks transactions if two transactions are made back to back. The driver does not check if previous transaction was completed before starting a new one. The transaction continues for a while after last write to I2C register because HW works idependently of the software.

LPC1549 I^{2}C driver

- I^{2}C driver takes a transfer descriptor and performs a transfer based on the values
- If both read and write are requested a combined transfer is made. Write takes place first and is followed by a repeated start and a read transaction
  - To read from a device register requires a combined transfer (write register address first, then switch to read)

```
/**
 * @brief Master transfer data structure definitions
 */
typedef struct {
const uint8_t *txBuff;/*!< Pointer to array of bytes to be transmitted */
uint8_t *rxBuff;/*!< Pointer memory where bytes received from I2C be stored */
uint16_t txSz;/*!< Number of bytes in transmit array,
                if 0 only receive transfer will be carried on */
uint16_t rxSz;/*!< Number of bytes to received,
                if 0 only transmission we be carried on */
uint16_t status;/*!< Status of the current I2C transfer */
uint8_t slaveAddr;/*!< 7-bit I2C Slave address */
} I2CM_XFER_T;
```