19Z604 Embedded Systems

Dr.N.Arulanand,
Professor,
Dept of CSE,
PSG College of Technology

Agenda

- I2C (I-squared-C) interface
- SPI - Serial Peripheral Interface

Drawbacks of UART Serial Communication

- UART is an asynchronous transmission protocol
- Devices using them must agree ahead of time on a data rate.
- The clocks of the devices must also have the clocks with almost same data rate.
- The data rate in UART is reduced because of the requirement of additional start and stop bits.
- Besides, the asynchronous transmission protocols are meant for data transfer between only two devices.
- Since a serial port can be connected to multiple devices, the bus contention happens often and it should be dealt with care to prevent damages to the devices.
- For asynchronous serial ports, there is a limit for the Baud rate. This is also a drawback of this communication protocol.

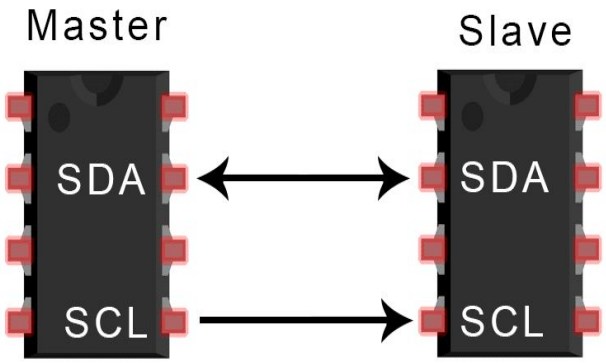
I^2C Communication

- The I^2C bus enables robust, high-speed, two-way communication between devices while using a minimal number of I/O pins to facilitate communication.
- An I2C bus is controlled by a master device (usually a microcontroller), and
- contains one or more slave devices that receive information from the master.

History of I^2C

- The I2C (Inter-Integrated Communication) protocol was invented by Phillips semiconductor (now NXP semiconductor) in the early 1980s to allow for relatively low-speed communication between various ICs.
- The protocol was standardized by the 1990s.
- The protocol is known as the "two-wire" protocol because two lines are used for communication, a clock and data line

I^2C



- Two-wire interface
- Serial Clock (SCL)
- Serial Data (SDA)

I2C reference Hardware Configuration

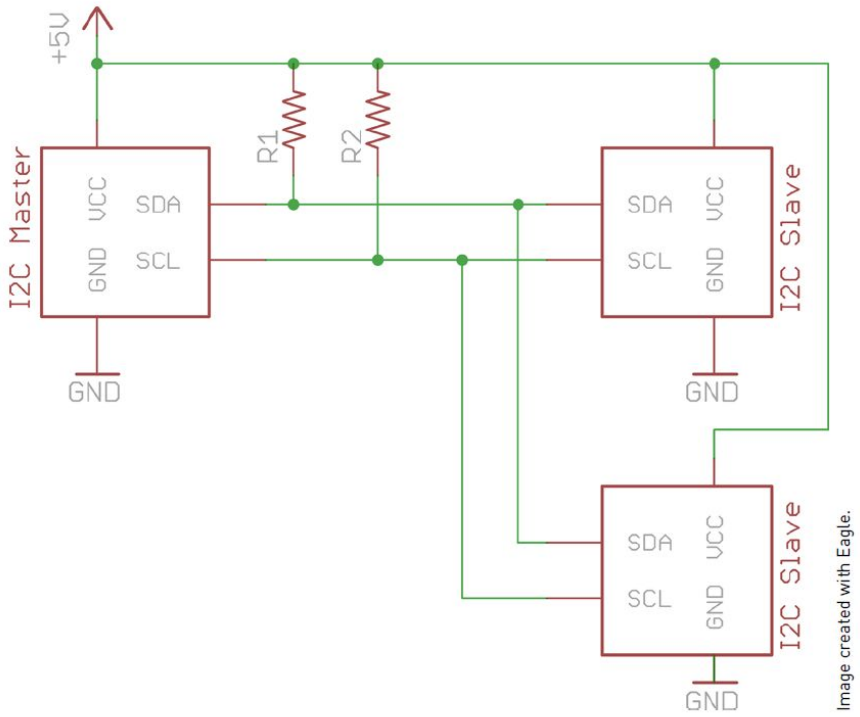


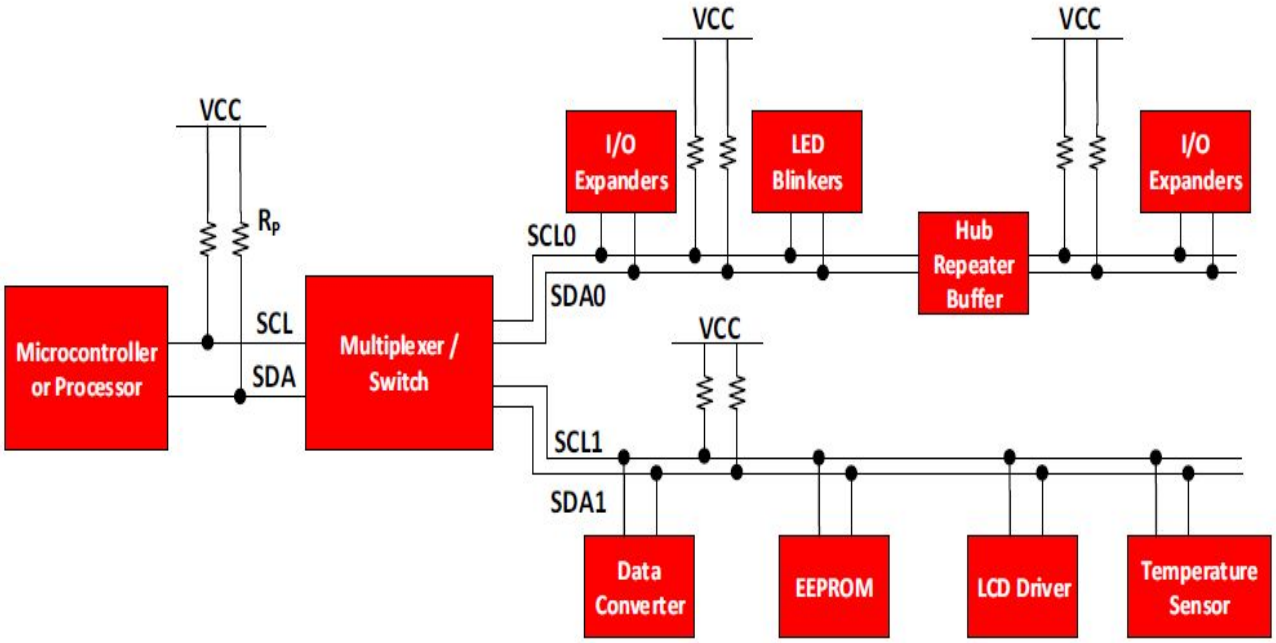
Figure 8-1: I²C reference hardware configuration

I2C is unique in that multiple devices all share the same communication lines: a clock signal (SCL), and a bidirectional data line (SDA).
Notice, Pull-up resistors on both data lines

Arduino : Master-Slave Configuration

- Arduino acts as the master device. The bus master is responsible for initiating all communications.
- Slave devices cannot initiate communications; they can only respond to requests that are sent by the master device. Because multiple slave devices share the same communication lines, it’s very important that only the master device can initiate communication. Otherwise, multiple devices may try to talk at the same time and the data would get garbled.

I^2C bus for an Embedded System

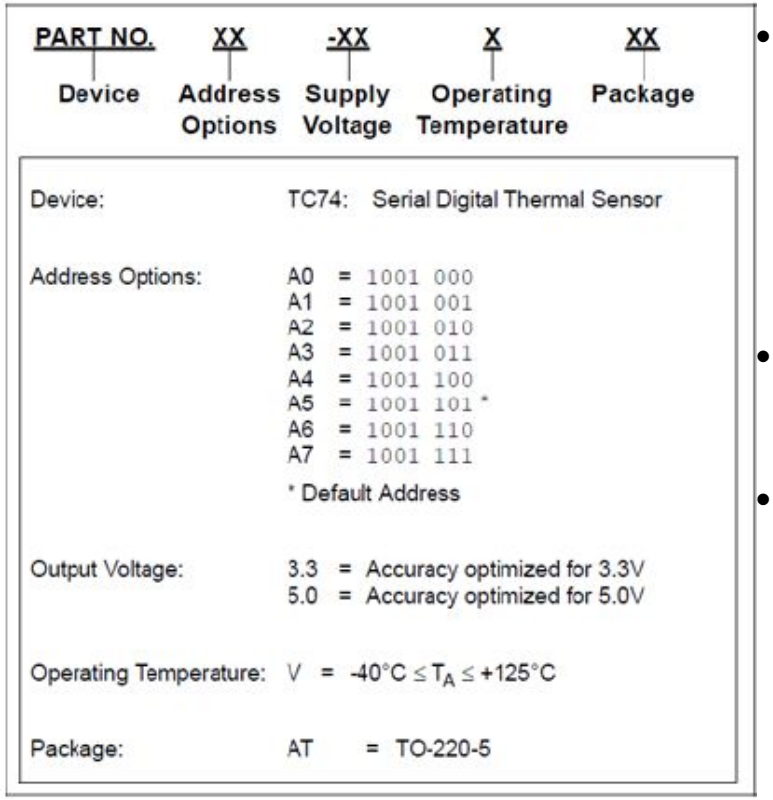


many different peripherals may
share a bus which is connected to a processor through only 2 wires

I2C Addressing

- Each I2C slave device has a unique 7-bit address, or ID number. When communication is initiated by the master device, a device ID is transmitted.
- I2C slave device react to data on the bus only when it is directed at their ID number.
- Because all the devices are receiving all the messages, each device on the I2C bus must have a unique address. Some I2C devices have selectable addresses, whereas others come from the manufacturer with a fixed address.
- If you want to have multiple numbers of the same device on one bus, you need to identify components that are available with different IDs.

TC74 digital temperature sensor



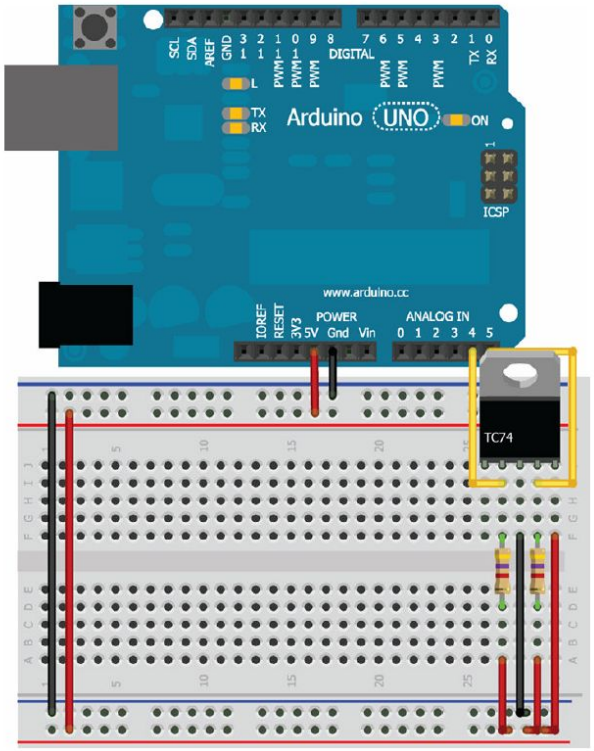
PART NO.	XX	-XX	X	XX
Device	Address Options	Supply Voltage	Operating Temperature	Package

Device:	TC74: Serial Digital Thermal Sensor
Address Options:	A0 = 1001 000 A1 = 1001 001 A2 = 1001 010 A3 = 1001 011 A4 = 1001 100 A5 = 1001 101 * A6 = 1001 110 A7 = 1001 111 * Default Address
Output Voltage:	3.3 = Accuracy optimized for 3.3V 5.0 = Accuracy optimized for 5.0V
Operating Temperature:	V = -40°C ≤ T_A ≤ +125°C
Package:	AT = TO-220-5

- Temperature sensors, for example, are commonly available with various preprogrammed I2C addresses because it is common to want more than one on a single I2C bus.
- The TC74 datasheet reveals that it is available with a variety of different addresses.
- make sure to be aware of the ID of the device you ordered so that you send the right commands

Pull-up resistor values

- Floating value – (tri-state)
- The value for these resistors depends on the slave devices and how many of them are attached.
- use 4.7kΩ resistors for both pull-ups; this is a fairly standard value that will be specified by many datasheets.
- https://www.youtube.com/watch?v=G_i1ZhadTa0



Steps for controlling any I2C device

- The basic steps are:

  1. Master sends a start bit.

  2. Master sends 7-bit slave address of device it wants to talk to.

  3. Master sends read (1) or write (0) bit depending on whether it wants to write/read data into/from an I2C device's register.

  4. Slave responds with an "acknowledge" or ACK bit (a logic low).

  5. In write mode, master sends 1 byte of information at a time, and slave responds with ACKs. In read mode, master receives 1 of byte information at a time and sends an ACK to the slave after each byte.

  6. When communication has been completed, the master sends a stop bit.

I2C Write Format

Write Byte Format

S	Address	WR	ACK	Command	ACK	Data	ACK	P
	7 Bits			8 Bits		8 Bits		

Slave Address

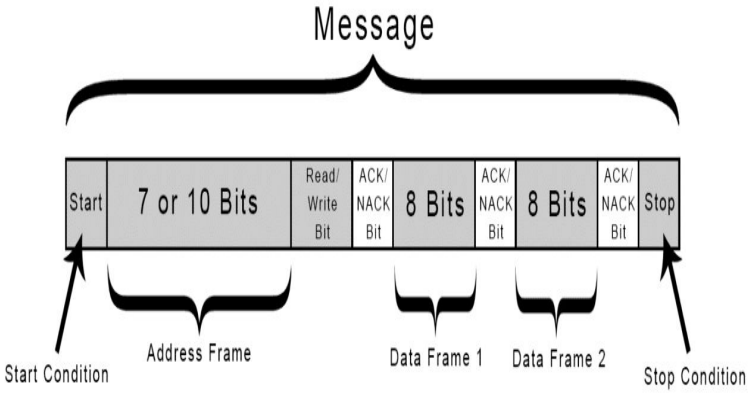
Command Byte: selects which register you are writing to.

Data Byte: data goes into the register set by the command byte.

S = START Condition
P = STOP Condition
Shaded = Slave Transmission

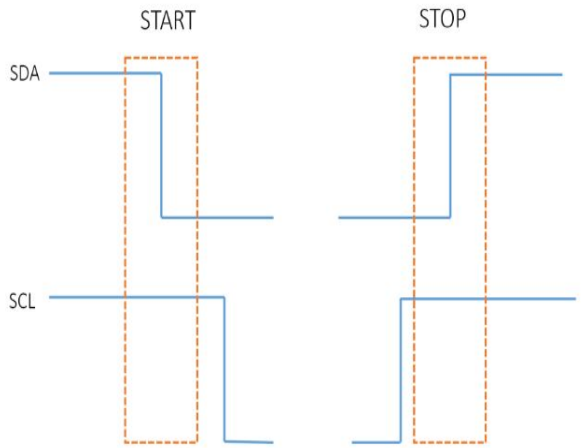
Data Byte: reads data from the register commanded by the last Read Byte or Write Byte transmission.

HOW I2C WORKS



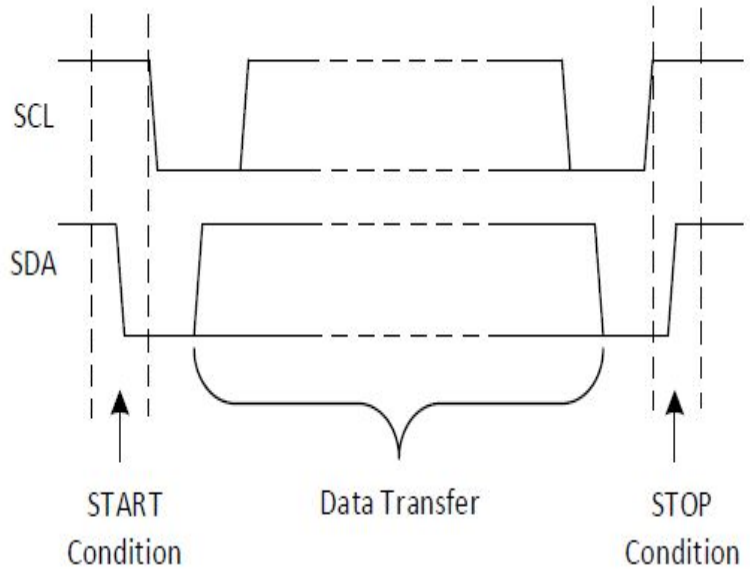
- data is transferred in *messages.*
- **Start Condition:** The SDA line switches from a high voltage level to a low voltage level *before* the SCL line switches from high to low.
- **Stop Condition:** The SDA line switches from a low voltage level to a high voltage level *after* the SCL line switches from low to high.
- **Address Frame:** A 7 or 10 bit sequence unique to each slave that identifies the slave when the master wants to talk to it.

Start and Stop Condition

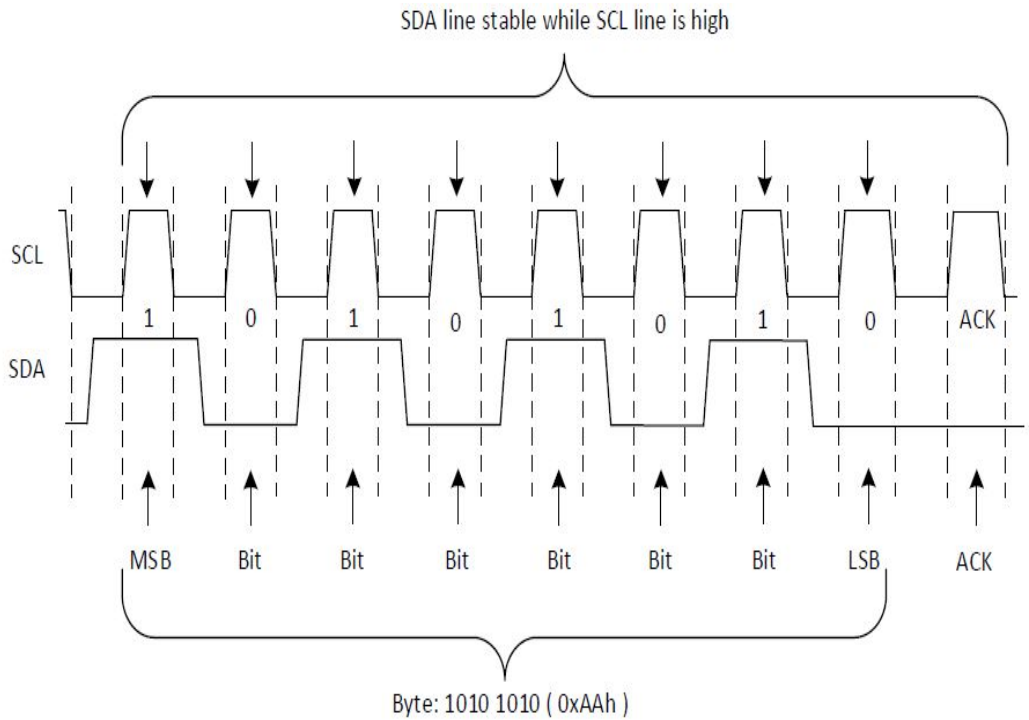


- Start Condition:
  - SCL high
  - SDA high to low
- Stop Condition
  - SCL high
  - SDA low to high

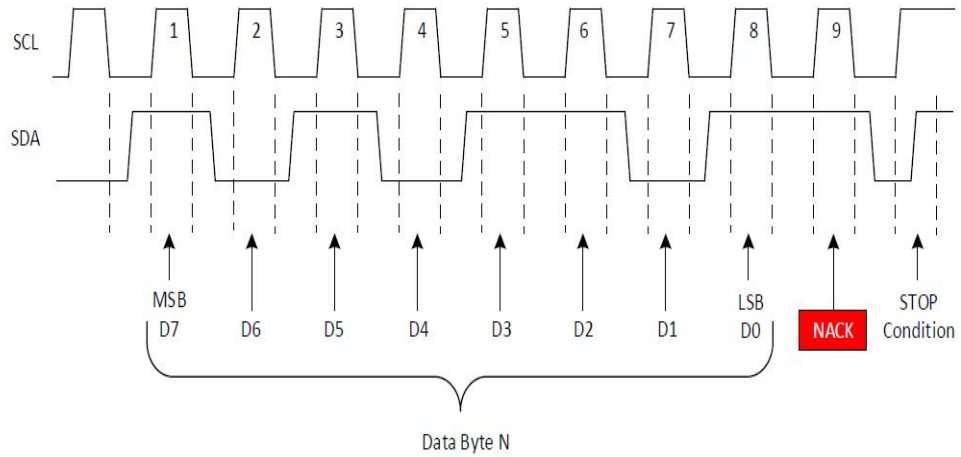
START and STOP condition



Single Byte Data Transfer



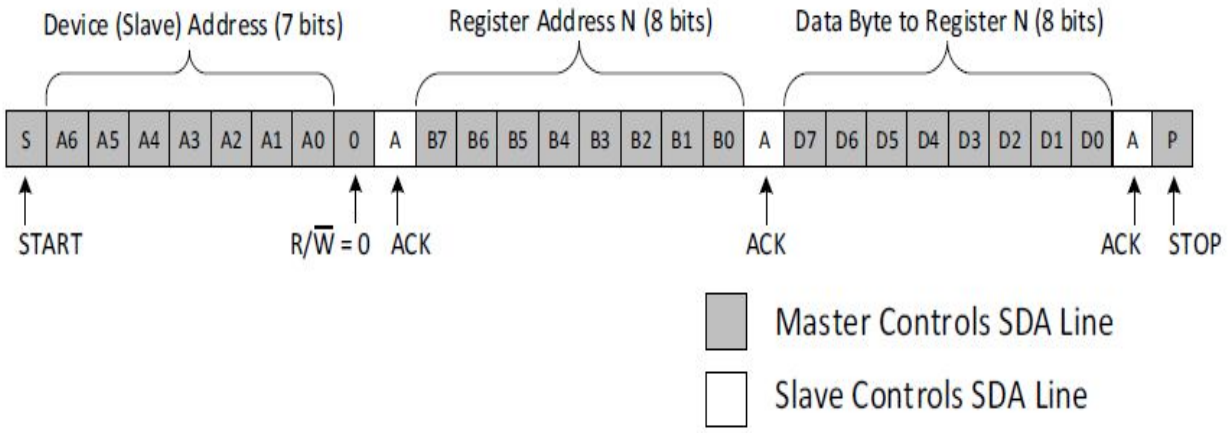
Example of NACK Waveform



There are several conditions that lead to the generation of a NACK:

1. 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.
2. During the transfer, the receiver gets data or commands that it does not understand.
3. During the transfer, the receiver cannot receive any more data bytes.
4. A master-receiver is done reading data and indicates this to the slave through a NACK

Writing to a Slave



I2C Read Format

Read Byte Format

S	Address	WR	ACK	Command	ACK	S	Address	RD	ACK	Data	NACK	P
	7 Bits			8 Bits			7 Bits			8 Bits		

Slave Address

Command Byte: selects which register you are reading from.

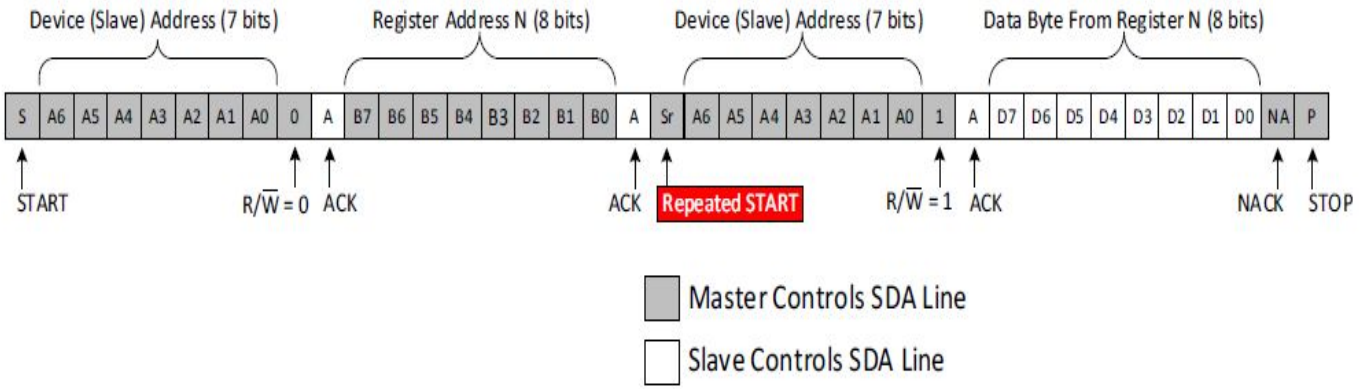
Slave Address: repeated due to change in data-flow direction.

Data Byte: reads from the register set by the command byte.

S = START Condition
P = STOP Condition
Shaded = Slave Transmission

Data Byte: reads data from the register commanded by the last Read Byte or Write Byte transmission.

Reading from a Slave



Byte-Oriented Transfer but Supports Multi-Byte

- I2C is fundamentally byte-oriented, meaning data is transferred in 8-bit (1 byte) chunks.
- However, multiple bytes can be transmitted in sequence without needing to restart communication.
- Once a START condition is sent, and the slave device is addressed, the master can send or receive multiple bytes continuously.
- The receiver sends an ACK (Acknowledge) bit after each byte, allowing uninterrupted multi-byte transfers.
- The transfer stops only when a STOP condition is issued.

Multi Byte Transfer

Examples of Multi-Byte Transfers

- Reading multiple registers from a sensor (e.g., accelerometer or EEPROM).
- Writing a block of data to memory or a display.
- Streaming data (e.g., from an audio codec or image sensor).

Buffer Size and Device Constraints

- While I2C itself allows multi-byte transfers, the number of bytes that can be transferred at once depends on the device's internal buffer size and protocol requirements.
- Some devices may require a repeated START condition after a certain number of bytes.

How a Slave Handles Buffer Full Scenario

Solution 1:

- After receiving a byte that fills its buffer, the slave can respond with a NACK (Not Acknowledge) instead of an ACK during the acknowledgment phase.
- This indicates to the master that the slave cannot accept more data.
- The master can then:
- Stop communication (generate a Stop Condition).
- Issue a Repeated Start Condition to re-initiate communication later.

Solution 2: Clock Stretching

Clock Stretching

- Clock stretching is a mechanism used in the I2C protocol where a device (usually a slave) can hold the clock line (SCL) low to pause communication temporarily.
- This allows the device to signal that it is not ready to send or receive data and needs more time to process the current byte or prepare the next one.

Example of Clock Stretching

Imagine a slave device (like a temperature sensor) that needs to process data before sending it to the master:

1. The master sends a read request to the slave.
2. The slave receives the request but needs more time to prepare the data.
3. Instead of immediately responding, the slave holds SCL low after acknowledging the request.
4. Once the slave is ready, it releases SCL, and the master continues with the next clock pulse.

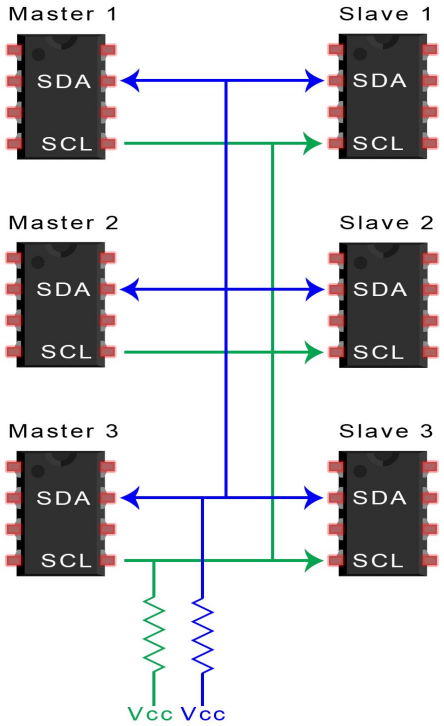
Key Use Cases for Clock Stretching

- **Slow Processing Devices:** Some slave devices (e.g., sensors or EEPROMs) may operate slower than the master's clock and need time to respond.
- **Buffer Management:** If the slave's buffer is full, it can use clock stretching until space is available for more data.
- **Data Readiness:** The slave may use clock stretching to wait until a data sample is ready or processed.

Who controls - Master or Slave ?

- The master initiates and ends communication and generates the overall clock signal.
- The master normally controls the clock in I2C.
- The electrical design (open-drain with pull-ups) allows the slave to **temporarily hold the clock line low**. This mechanism enables slaves to signal "wait" to the master, ensuring proper communication timing and preventing data corruption.

Multiple Master with Multiple Slave



Bus Contention & Arbitration Issues

- When multiple masters try to communicate at the same time, bus contention occurs.
- I2C handles this using arbitration, where masters monitor the bus while transmitting.
- If a master detects another master sending a lower-valued (dominant) bit, it loses arbitration and backs off.
- Issue: If arbitration is not handled properly, data corruption or communication failures can occur.

Arbitration

- If two masters start transmitting at the same time, they both monitor the SDA line while sending their data.
- Each master checks if the value it sent matches what is on the bus.

Example

- Master A sends 1 (recessive), but Master B sends 0 (dominant).
- The actual value on the bus is 0 (dominant overrides recessive).
- Master A detects that the bus value doesn't match what it sent (1), so it loses arbitration and stops transmitting.
- Master B continues to control the bus

Clock Synchronisation Problems

- Since multiple masters generate clock signals (SCL), they must synchronize.
- I2C ensures this using clock stretching, where a slower device can hold SCL low until it's ready.
- Issue: If a master doesn’t handle clock stretching properly, it may misinterpret data timing.

Address Conflicts

- If multiple slaves share the same 7-bit or 10-bit address, communication issues arise.
- Issue: Two slaves may respond simultaneously, causing data corruption.
- Solution: Unique addresses should be assigned, or use General Call Addressing (0x00) for broadcasting.

Arduino Wire Library

```
//Reads Temp from I2C temperature sensor and prints it on
        the serial port
//Include Wire I2C library
#include <Wire.h>
int temp_address = 72; //1001000 written as decimal
        number
void setup()  {
        //Start serial communication at 9600 baud

        Serial.begin(9600);

        //Create a Wire object
        Wire.begin();
}

 Void loop()  {
//Send a request
//Start talking to the device at the specified address
Wire.beginTransmission(temp_address);

//Send a bit asking for register zero, the data register
Wire.write(0);
//Complete Transmission
Wire.endTransmission();
//Read the temperature from the device
//Request 1 Byte from the specified
    address
Wire.requestFrom(temp_address, 1);
//Wait for response
while(Wire.available() == 0);
//Get the temp and read it into a variable
int c = Wire.read();
//Do some math to convert the Celsius to
    Fahrenheit
int f = round(c*9.0/5.0 +32.0);
//Send the temperature in degrees C and
    F to the serial monitor
Serial.print(c);
Serial.print("C ");
Serial.print(f);
Serial.println("F");
delay(500);
}
```

I^2C

- Two-wire Interface
- I^2C combines the best features of SPI and UARTs.
- With I2C,
  - multiple slaves can connect to a single master (like SPI)
  - Multiple masters controlling single, or multiple slaves.
    - Ex: More than one microcontroller logging data to a single memory card or displaying text to a single LCD.

Convert Temp to actual value

- Refer to the sensor's datasheet to understand how the raw bytes map to temperature. For example:
  - Some sensors return **2 bytes** of raw data.
  - You might need to combine these bytes (e.g., MSB and LSB) to form a 16-bit value.
- Apply scaling factors, offsets, or calculations specified in the datasheet to convert the raw data into a temperature value.

Example for a 16-bit sensor:

python Copy Edit

```python
# Assume raw_data is a 2-byte list [MSB, LSB]
raw_value = (raw_data[0] << 8) | raw_data[1]  # Combine MSB and LSB
temperature = raw_value * scale_factor + offset  # Use sensor-specific formula
```

Pros and Cons

- Advantages:
  - Only uses two wires
  - Supports multiple masters and multiple slaves
  - ACK/NACK bit gives confirmation that each frame is transferred successfully
  - Hardware is less complicated than with UARTs
  - Well known and widely used protocol
- Disadvantages:
  - Slower data transfer rate than SPI
  - The size of the data frame is limited to 8 bits
  - More complicated hardware needed to implement than SPI