GPIO, I2C, and SPI on Embedded Devices

Sommerakademie in Leysin
AG 2 – Effizientes Rechnen

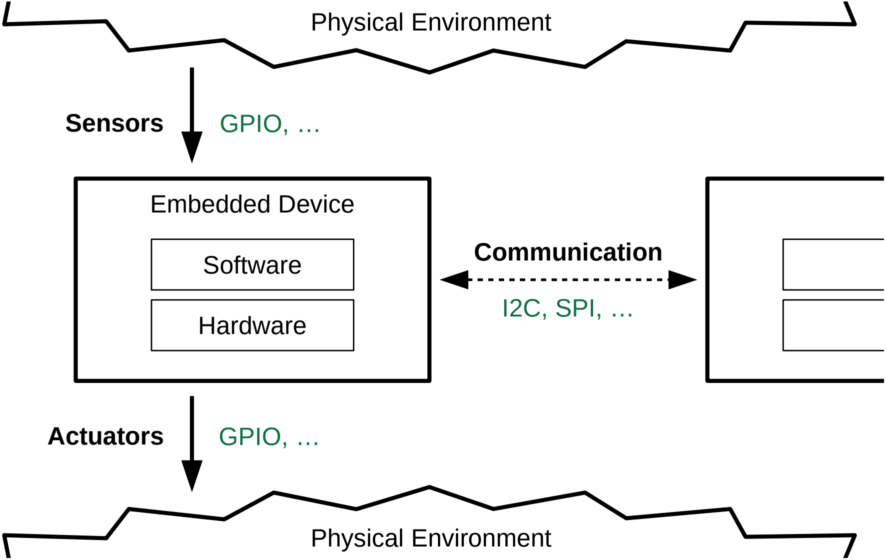
Maximilian Köhl

Universität des Saarlandes

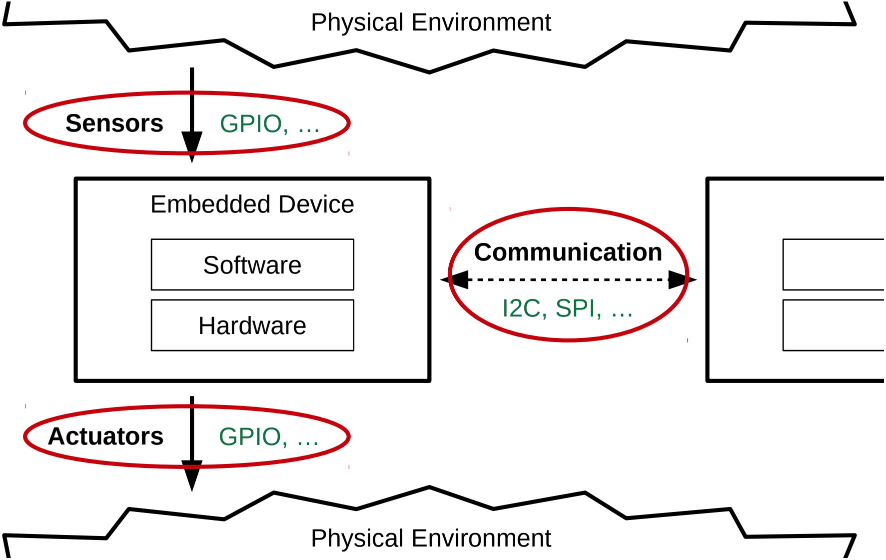
August 2016

Structure

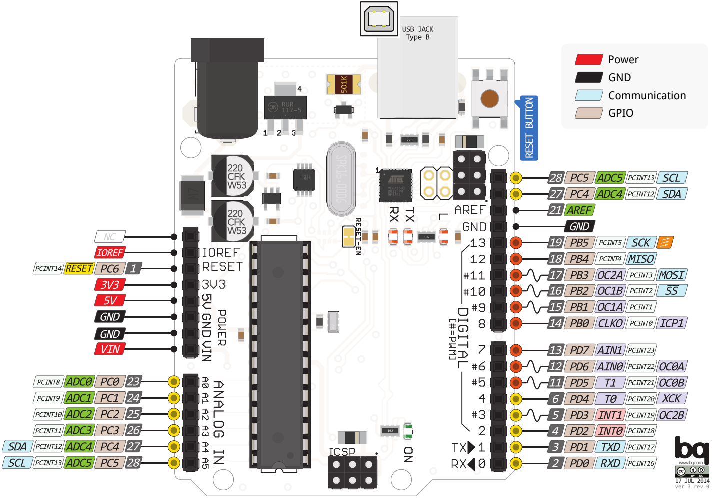
Embedded Systems



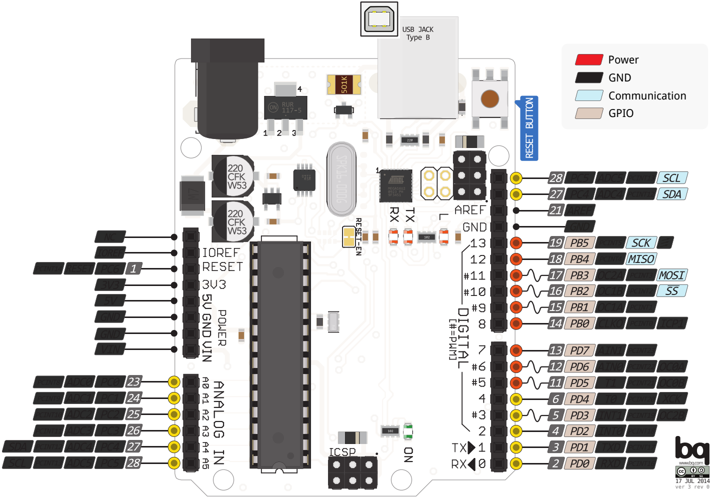
Embedded Systems



Arduino's Pins



Arduino's Pins



GPIO Pin

General **P**urpose [digital] **I**nput/**O**utput Pin:

- pin behavior is **user specified**
- digital: logical **high** or **low** voltage level

- **input mode**: does not drain or provide current
- **output mode**: drains and provides current

- may be used as interrupt source
- with or without **pull-up resistor**

Memory-Mapped IO

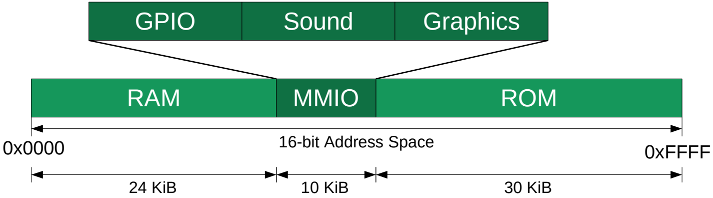


Figure: Typical Memory Layout

- special address space for IO
- **one bus** for both, memory and peripheral devices

Program GPIO Pins (Arduino)

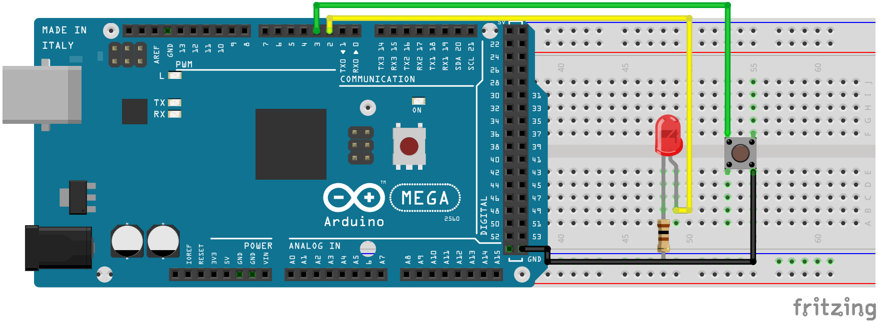
```
pinMode(4, OUTPUT);

pinMode(5, INPUT);
pinMode(6, INPUT_PULLUP);

digitalWrite(4, HIGH);
digitalWrite(4, LOW);

digitalRead(5);
```

Example using an Arduino



Program GPIO Pins (Raspberry)

```
var raspi = require('raspi-io');
var five = require('johnny-five');
var board = new five.Board({ io: new raspi() });

board.on("ready", function() {
  var led = new five.Led("P1-2");
  var button = new five.Button("P1-3");

  led.on();

  button.on("up", function () {
    led.on();
  });
  button.on("down", function () {
    led.off();
  });
});
```

Introduction

Distribution: separation into components which communicate

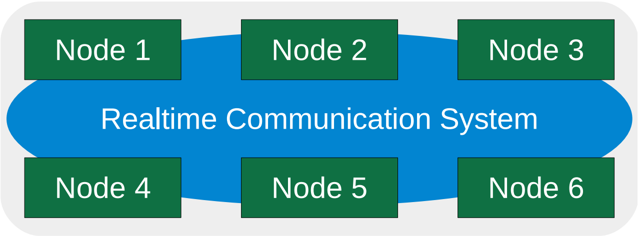


Figure: Distributed Architecture

Why a distributed solution?

- **composability** allows modular system development
- **reliability** is improved by introducing redundancy

Communication Protocols

- **protocol** = contract about how to communicate
- **stream** and **message** based protocols
- **message** = basic entity of information (bit-string)
- message types: **event messages**, **state messages**

When to send a message?

external control: decision by host computer
event-triggered, explicit send command, receiving host is interrupted

autonomous control: decision by communication system
time-triggered, regularly exchange of state information

Protocol Requirements

- meet safety critical **real-time** constraints
- cost and **energy efficient**
- appropriate bandwidth and communication delay
- robustness and **fault tolerance**
- maintainability and diagnosability

Error Detection

- detection of **transmission errors** using:
  - **checksums**
  - **acknowledgments**
- detection of **node errors** using:
  - **acknowledgments**

Communication Protocols

Latency and Jitter

- **small latency** = rapid communication, low distribution overhead
- **low jitter** = low variation in latency, reliable temporal properties

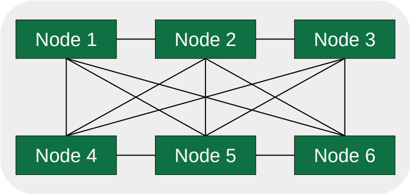
Full- and Half-Duplex Communication

- **full duplex** = simultaneous communication in both directions
- **half duplex** = only unidirectional communication at every instant

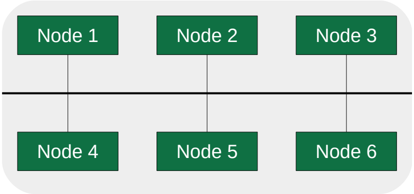
Flow Control

- **implicit flow control** = pace is pre-determined
- **explicit flow control** = receiver controls pace

How to connect the nodes?



(a) P2P Topology



(b) Bus Topology

Figure: Network Topologies

Peer-to-Peer vs. Bus Topology:

- **economic efficiency**: reduced number and length of cables
- **modular** system development, support, and evolution
- **efficient diagnosis** by bus snooping

Coordination of Bus Access

How to coordinate access to the communication bus?

Fundamental Tension

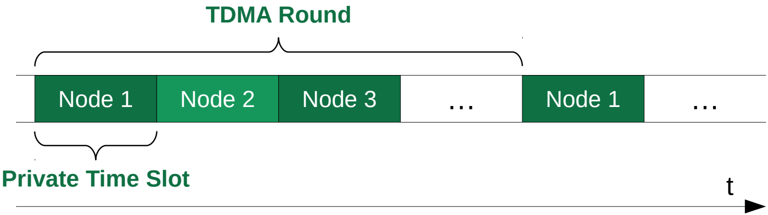
- small latency for important nodes/messages vs.
- sufficient and consistent service for less important nodes/messages

In practice: Inverse relation between volume and urgency.

Bus Arbitration

- Time Division Multiplexed Access
- Bus Master Approach
- Carrier Sense Multiple Access

TDMA – Time Division Multiplexed Access



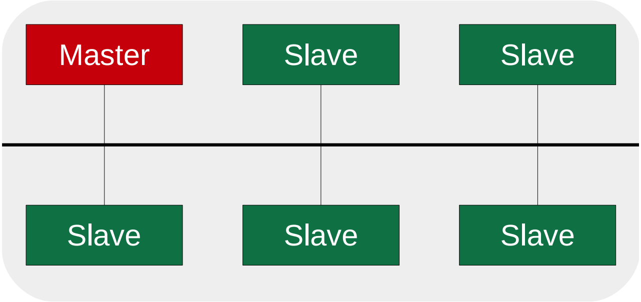
Advantages

- guaranteed worst-case latency

Disadvantages

- private time slots waste bandwidth
- number of nodes fixed during installation

Bus Master Approach to Coordination



Error Detection

- unresponsive or slow slave: use timeouts
- faulty master: use heartbeats

Bus Master Approach to Coordination

Slave-to-Slave Communication

- master needs to poll a willing-to-send slave
- slave transmits message only when polled
- potential recipients listen to bus traffic

Advantages

- simple to implement
- bounded latency

Disadvantages

- centralized master $\Rightarrow$ single point of failure
- polling consumes bandwidth
- number of nodes fixed during installation

CSMA – Carrier Sense Multiple Access

If communication medium is idle then send a message.

Problem

- two nodes might start sending almost simultaneously $\Rightarrow$ collision
- messages may be crippled or overwritten

CSMA/CD – CSMA with Collision Detection

- detect collisions
- back of for a random time

CSMA/CA – CSMA with Collision Resolution

- resolve collisions
- e.g. using bit-arbitration

Node Structure

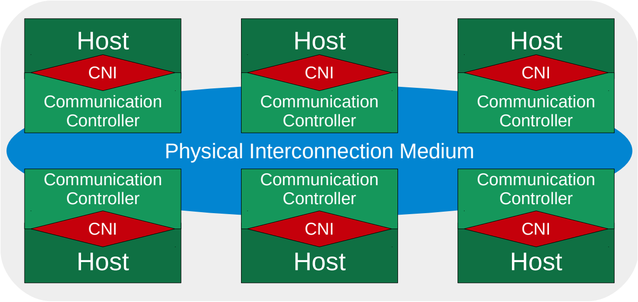
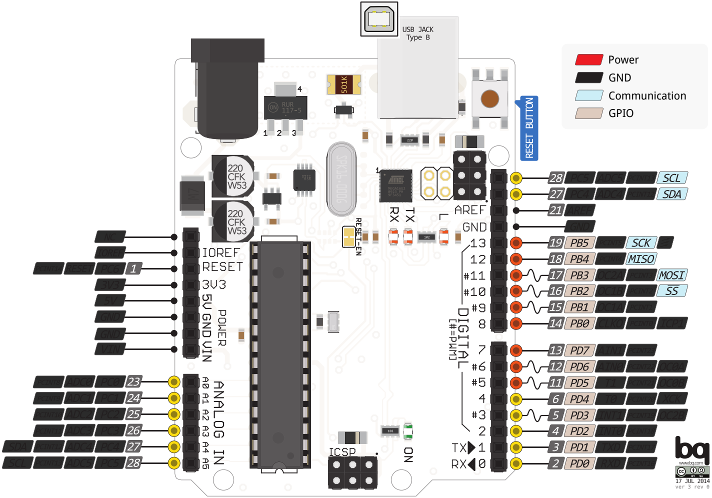


Figure: Host and Communication Controller

- **CNI**: Communication Network Interface
- hides details of communication protocol
- special purpose pins for communication

Arduino's Pins



Introduction

Inter-**I**ntegrated **C**ircuit [Protocol]

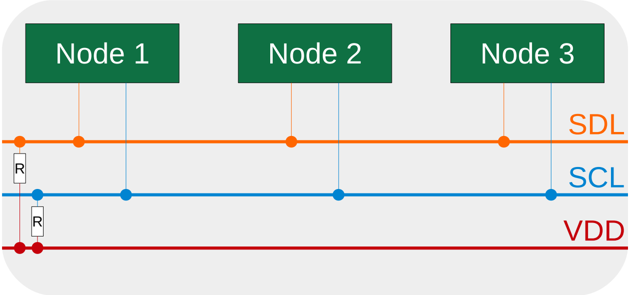


Figure: I2C Bus Signals

SDL Serial Data Line
SCL Serial Clock Line

I2C Physical Layer

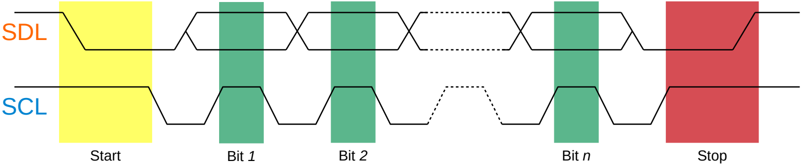


Figure: I2C Timing Diagram

1. Start Condition
2. Message
3. Stop Condition

I2C Data Layer

- **Master** Role
- **Slave** Role

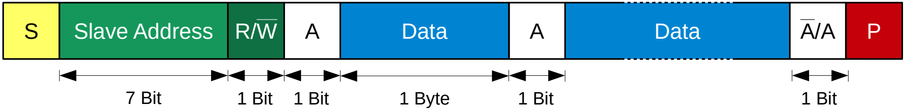


Figure: I2C Message Format

- flow control: clock stretching
- various clock speeds
- roles may be changed between messages
- multi-master bus

I2C Physical Layer

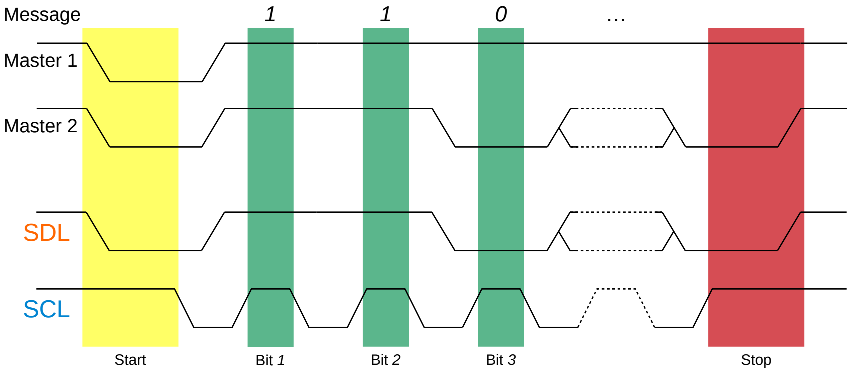


Figure: I2C Timing Diagram with Collision

How to use I2C to talk to a sensor?

- example: TMP100-Q1 digital temperature sensor

```
#include <Wire.h>

#define ADDRESS 0x94

void setup()
{
  // initialize I2C communication as master
  Wire.begin();

  // initialize serial communication
  Serial.begin(9600);

  // start I2C transmission
  Wire.beginTransmission(ADDRESS);
  // select configuration register
  Wire.write(0x01);
  // set continuous conversion, comparator mode, 12-bit resolution
  Wire.write(0x60);
  // stop I2C transmission
  Wire.endTransmission();

  delay(300);
}
```

How to use I2C to talk to a sensor?

```
void loop() {
  unsigned int data[2];

  // start I2C transmission
  Wire.beginTransmission(ADDRESS);
  // select data register
  Wire.write(0x00);
  // stop I2C Transmission
  Wire.endTransmission();

  // request 2 bytes of data
  Wire.requestFrom(ADDRESS, 2);

  // read 2 bytes of data
  if(Wire.available() == 2) {
    data[0] = Wire.read();
    data[1] = Wire.read();
  }

  // convert the data
  float temp = (((data[0] * 256) + (data[1] & 0xF0)) / 16) * 0.0625;

  // output data to serial monitor
  Serial.print("Temperature in Celsius : ");
  Serial.println(temp);

  delay(500);
}
```

Introduction

Serial **P**eripheral **I**nterface

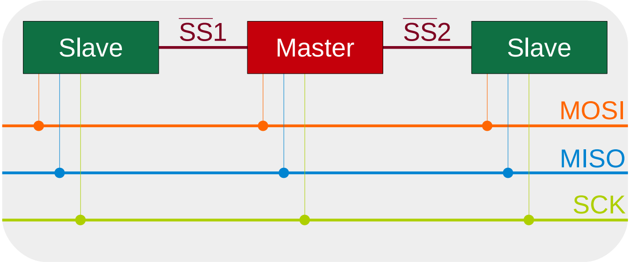


Figure: SPI Bus Signals

$\overline{SS}$	Slave Select
MOSI	Master Out Slave In
MISO	Master In Slave Out
SCK	Serial Clock

SPI Physical Layer

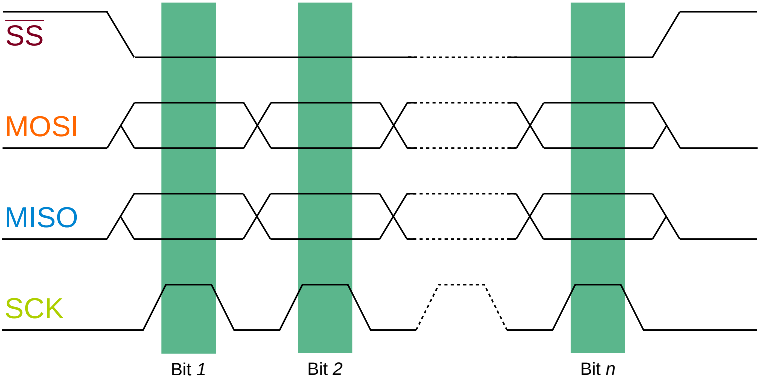


Figure: SPI Timing Diagram

Conclusion

I2C	SPI
2-wires	$3 + n$ wires
multi-master	single-master
half-duplex	full-duplex
5 Mbit/s	over 10 Mbit/s

- **high speed**: use SPI
- **modularity**: use I2C

Thank you for your Attention!