PROGRAMMERING AV INBYGGDA SYSTEM

Power optimisation and comm

Dario Salvi, 2025

Materials

- ESP32 Technical Reference Manual: chapter 31 Low-Power Management
- Making embedded system: chapter 6, Communicating with Peripherals and chapter 10, Reducing Power Consumption
- Online resources:
  - https://randomnerdtutorials.com/esp32-deep-sleep-arduino-ide-wake-up-sources/
  - https://lastminuteengineers.com/esp32-sleep-modes-power-consumption/
  - https://lastminuteengineers.com/esp32-deep-sleep-wakeup-sources/
  - https://deepbluembedded.com/esp32-sleep-modes-power-consumption/
  - http://www.lucadentella.it/en/2018/01/22/esp32-29-deep-sleep/
  - https://docs.espressif.com/projects/esp-idf/en/stable/api-reference/system/sleep_modes.html
  - http://fastbitlab.com/stm32-i2c-lecture-3-i2c-protocol-explanation/
  - https://barrgroup.com/Embedded-Systems/How-To/CRC-Calculation-C-Code
  - https://docs.espressif.com/projects/esp-idf/en/stable/api-reference/peripherals/i2c.html
  - http://www.lucadentella.it/en/2017/10/09/esp32-23-i2c-basic/

Saving power

- Decreasing a system's power consumption can take an order of magnitude more time than implementing the product features.
  - Choosing all the right hardware components.
  - Processor is likely to be one of the largest consumers of power in the system.
    - -> Choose a processor with the minimum amount of resources needed.
    - -> Software strategies.
- Remember:
  - power = $(\text{current})^2$ * resistance -> the lower the current the better
  - power = voltage * current -> if current is fixed, lower the voltage
  - energy = power * time -> switch off components when not needed!

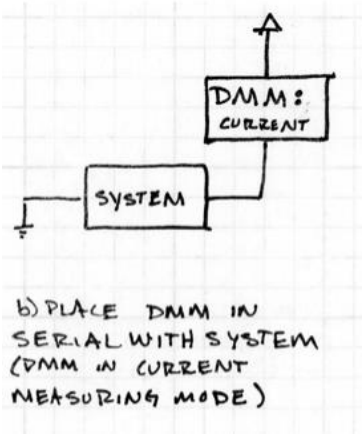
Batteries and currents

- Batteries are rated with **voltage** and **capacity**.
- Example:
  - An alkaline AA battery has a nominal voltage of 1.5V and a capacity of about 3000mAh (milliamp hours).
  - If your system takes 30mA and 1.5V when it is on, with an AA battery, it should last about 100 hours (~4 days).
  - If your system takes 300mA, it should last about 10 hours.

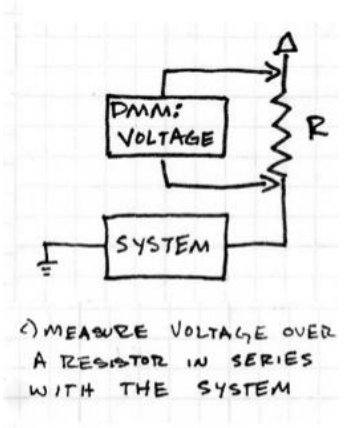
- Measuring currents during operation and sleep is fundamental!

Measuring currents

- Option 1:
  - Place an amperemeter in series with your system
  - Usually OK for currents of mA.



- Option 2:
  - Place a voltmeter in parallel with a resistor in series
  - Apply Ohm law to get current:
    - current = voltage / resistance
  - Resistor must be small (10Ω) to not consume current when heated!
  - OK for currents of 50μA

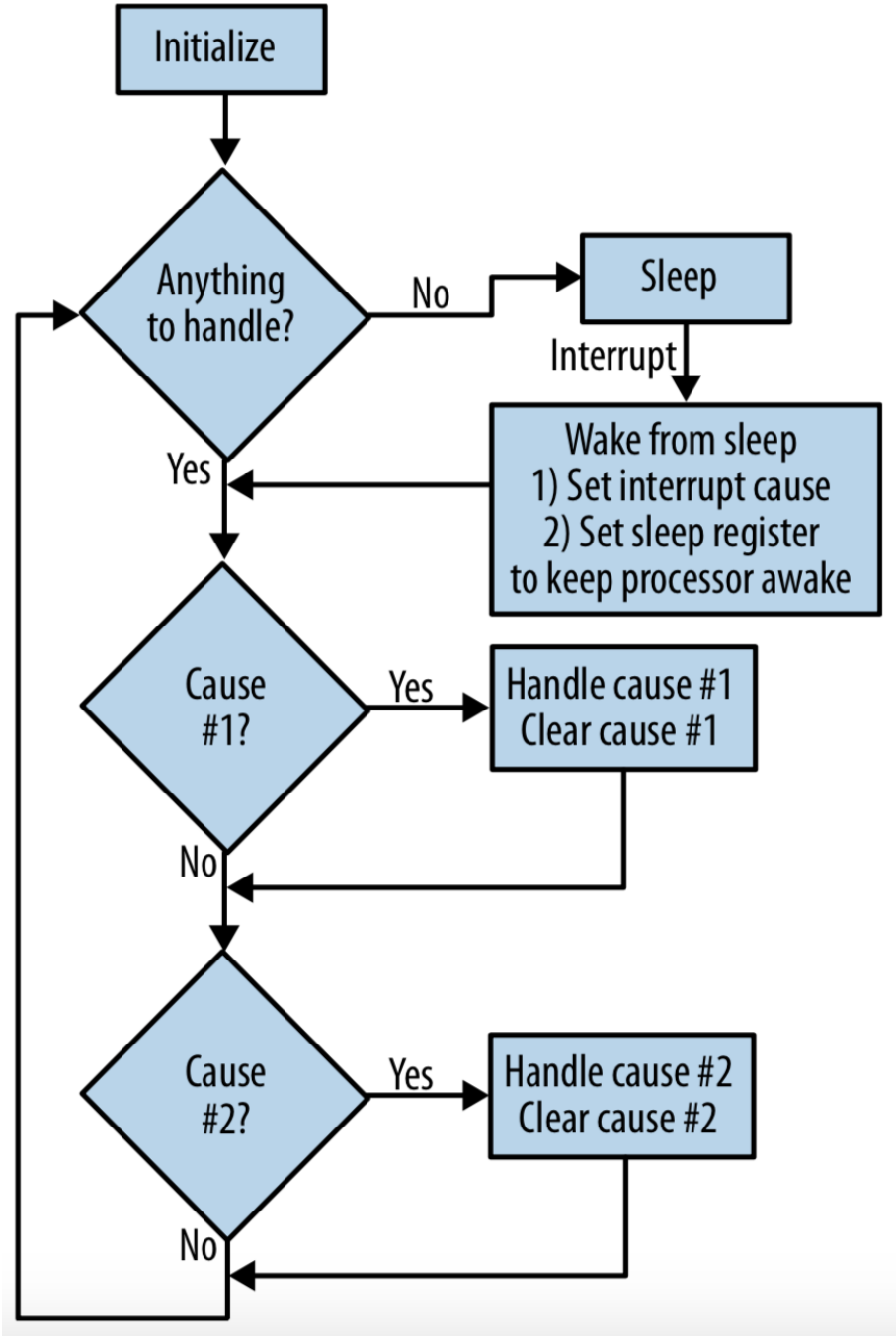


Turning off stuff

- Save power: **turn off components that are not needed**.
- Downside:
  - Those switched-off components won't be ready when you need them.
  - Bringing them back will add both some power usage and some latency in responding to events.
- Things to turn off:
  - **External chips** / **peripherals**: may need to be removed power or put into sleep mode
  - **Unused IO devices**: use your pins to remove current, set them to input with pull-downs (or output low, or input with pull-ups)
  - **Processor Subsystems**: use internal instructions to turn off parts or their clock
- Power consumption of the chip is proportional to the frequency it runs at
  - You can **slow down the clock**
  - You can slow down / accelerate clock depending on current need

Sleep

- Some MCU go into low power modes and wake up on interrupts.
- Can support different sleep modes:
  - **Idle**: processor is off, but RAM and timers are on
  - **Deep sleep**: turns off also peripherals, except those that can generate interrupt
  - **Deep hibernation**: leave RAM unstable but registers are kept
  - **Power off**: needs a complete reboot



Watchdogs and low-power coprocessors

- When using a watchdog there are two options:
  1. **Stop the watchdog** when in sleep mode (less fail tolerant!)
  2. **Keep the watchdog running** but add a timer to wakeup the processor and feed the dog regularly (consumes a bit more power)

- Some systems (like the ESP32) have **low power co-processor**
  - These can be used to deal with simple handling of events or polling data
  - They can wake up the main processor only when needed

ESP32 Power modes

- The esp32 chip offers **5 different power modes**
  - Active mode
  - Modem sleep
  - Light sleep
  - Deep sleep
  - Hibernation

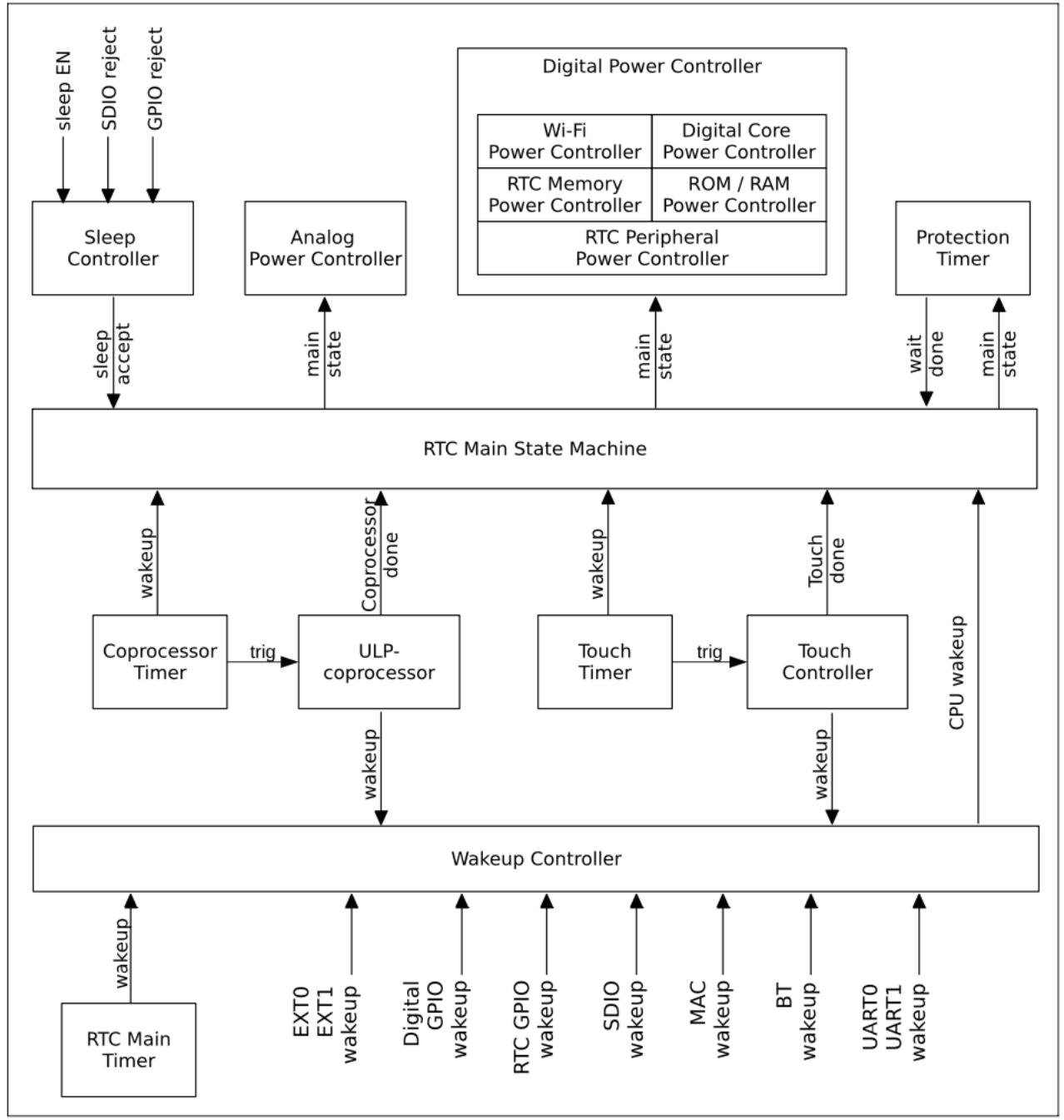
https://deepbluembedded.com/esp32-sleep-modes-power-consumption/

Hardware	Active Mode	Modem-sleep Mode	Light Sleep Mode	Deep-sleep Mode	Hibernation
CPU	ON	ON	PAUSE	OFF	OFF
WiFi/BT	ON	OFF or “Association sleep pattern”	OFF	OFF	OFF
RTC and RTC Peripherals	ON	ON	ON	ON	ON
ULP-Co Processor	ON	ON	ON	ON/OFF	OFF
Memory kept	RAM	RAM	RAM	RTC	None
Typical current	160 - 260mA	3 - 20 mA	0.8 mA	10μA- 0.15 mA	2.5μA

Some degree of configuration of the sleep modes is possible!

RTC module

- The RTC module is designed to handle the **entry into, and exit from, the low-power mode**, and control the clocks, power and gates.
  - RTC main state machine.
  - Digital & analogue power controller.
  - Sleep & wakeup controller.
  - Timers: RTC main timer, ULP co-processor timer and touch timer.
  - ULP co-processor, touch controller, SAR ADC controller.
  - Retention memory:
    - RTC slow memory: 8 KB SRAM, mostly used as retention memory for the ULP co-processor.
    - RTC fast memory: an 8 KB SRAM, mostly used as retention memory for the CPU accesses.
  - Retention registers: always-on registers of 8 x 32 bits, serving as data storage.
  - RTC IO pads: 18 always-on analogue pads, usually functioning as wake-up sources.


sleep EN
SDIO reject
GPIO reject
Digital Power Controller
Wi-Fi Power Controller
Digital Core Power Controller
RTC Memory Power Controller
ROM / RAM Power Controller
RTC Peripheral Power Controller
Sleep Controller
Analog Power Controller
Protection Timer
sleep accept
main state
main state
wait done
main state
RTC Main State Machine
wakeup
Coprocessor done
wakeup
Touch done
Coprocessor Timer
trig
ULP-coprocessor
Touch Timer
trig
Touch Controller
CPU wakeup
wakeup
wakeup
Wakeup Controller
wakeup
RTC Main Timer
EXT0 EXT1 wakeup
Digital GPIO wakeup
RTC GPIO wakeup
SDIO wakeup
MAC wakeup
BT wakeup
UART0 UART1 wakeup

Wakeup causes

- **Timer**
  - Built-in timer, RTC SLOW_CLK, can be used to wake up the chip after a predefined amount of time. Doesn't require RTC peripherals or RTC memories to be powered on during sleep.
- **Touch pad**
  - RTC IO module contains logic to trigger wakeup when a touch sensor interrupt occurs.
- **External wakeup**
  - Wakeup when one of **RTC GPIOs** is set to a predefined logic level. (Warning not all GPIOs are connected to RTC!)
  - EXT0: RTC peripherals need to be kept powered on during sleep. Internal pullup or pulldown resistors can be used.
  - EXT1: RTC peripherals and RTC memories can be powered down. Internal pullup or pulldown cannot be used.
- **GPIO**
  - Can use all GPIOs, works only in light sleep.
- **ULP coprocessor**
  - Runs the program stored in RTC slow memory. RTC slow memory and RTC peripherals will be powered on.
- **UART**
  - Wakes up the chip from light sleep when a certain number of positive edges on RX pin are seen.

https://randomnerdtutorials.com/esp32-pinout-reference-gpios/

WAKEUP_ENA	Wake-up Source	Light-sleep	Deep-sleep	Hibernation	Notes*
0x1	EXT0	Y	Y	-	1
0x2	EXT1	Y	Y	Y	2
0x4	GPIO	Y	Y	-	3
0x8	RTC timer	Y	Y	Y	-
0x10	SDIO	Y	-	-	4
0x20	Wi-Fi	Y	-	-	5
0x40	UART0	Y	-	-	6
0x80	UART1	Y	-	-	6
0x100	TOUCH	Y	Y	-	-
0x200	ULP co-proccesor	Y	Y	-	-
0x400	BT	Y	-	-	5

ESP-IDF sleep modes

1. Configure the wakeup source(s)
   - **esp_sleep_enable_timer_wakeup()** for using the RTC timer
   - **esp_sleep_enable_touchpad_wakeup(void)** for using the touchpad
   - **esp_sleep_enable_ext0/1_wakeup()** for using an RTC pin
   - **esp_sleep_enable_ulp_wakeup()** for using the ULP coprocessor
   - **gpio_wakeup_enable()** uses a (non RTC) GPIO
   - **esp_sleep_enable_uart_wakeup()** uses the UART
2. Disable pins with fixed pull-up pull-downs, e.g.with **rtc_gpio_isolate(GPIO_NUM_12)**
3. Call **esp_deep_sleep_start()** for deep sleep or **esp_light_sleep_start()** for light sleep
4. Upon wakeup, check the wakeup cause with **esp_sleep_get_wakeup_cause()**

Live coding

- Example of deep sleep in the ESP32.

FreeRTOS and sleep

- One way to reduce the power consumed by the microcontroller is using the **Idle task hook** to place the microcontroller into a low power state.
- Limitation: needs to periodically exit and then re-enter the low power state to **process tick interrupts**.
- If the tick is fast enough this strategy can consume more power than not entering low power modes.

https://www.freertos.org/low-power-tickless-rtos.html

Tickless Idle mode

- The **tickless idle mode** stops the periodic tick interrupt during idle periods, then makes a correcting adjustment to the RTOS tick count value when the tick interrupt is restarted.
- Stopping the tick interrupt allows the microcontroller to remain in a low power state until either an interrupt occurs, or it is time for the RTOS kernel to transition a task into the Ready state.
- When the tickless idle functionality is enabled the kernel will call the portSUPPRESS_TICKS_AND_SLEEP() macro when the following two conditions are both true:
    1. The **Idle task is the only task able to run** because all the application tasks are either in the Blocked state or in the Suspended state.
    2. **At least *n* further complete tick periods will pass** before the kernel is due to transition an application task out of the Blocked state, where *n* is set by the configEXPECTED_IDLE_TIME_BEFORE_SLEEP definition in FreeRTOSConfig.h.

ESP-IDF automatic power management

- Power management algorithm can :
  - Adjust the Advanced peripheral bus (APB) frequency.
  - Adjust CPU frequency.
  - Put the chip into light sleep mode.
- Cost: increased interrupt latency (up to 10µs).
- Must be activated at compile time with a flag (see https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/kconfig.html) + using **esp_pm_configure()**
- Don’t forget to include wakeup sources if needed!
- Automatic light sleep is based on FreeRTOS Tickless Idle functionality.

So many peripherals!

- A peripheral is **anything outside your processor that it communicates with**. Peripherals come in all shapes and flavours!
- More complex peripherals need a **communication protocol** to interact with the CPU:
  - External Memory (usually ROM)
  - Keyboards and buttons matrixes
  - Sensors (sound, light, accelerometers, gyroscopes, magnetometers, wind, humidity, temperature, cameras, infrared detectors…)
  - Actuators (motors, solenoids…)
  - Displays
  - Lights

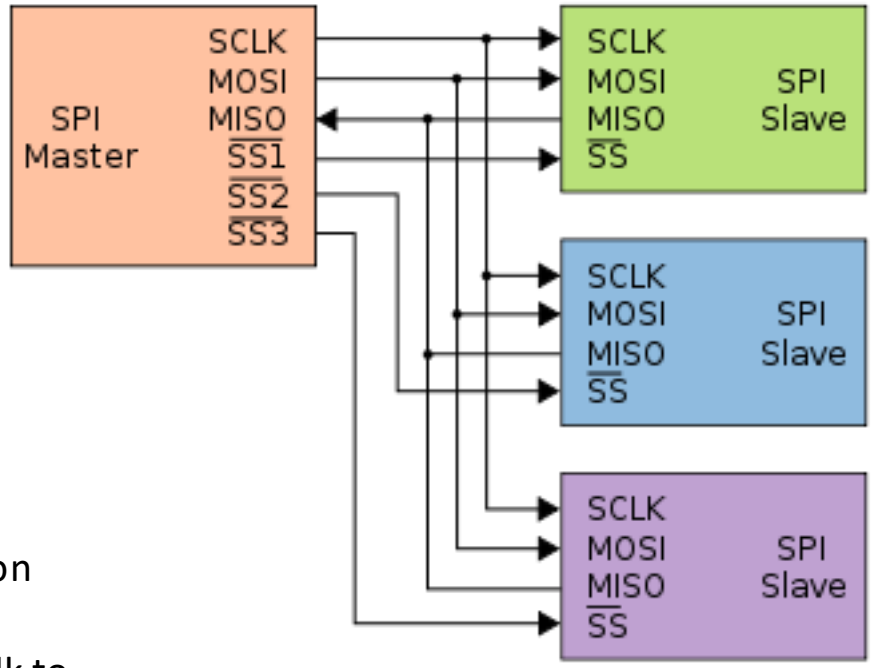
So many communication protocols!

- Integrating digital systems needs a protocol
- Communication between host and peripheral usually needs a **clock** to synchronise the two parties
- Possible technologies:
  - Ethernet and WiFi
    - Complex and costly, but reliable and flexible, can work on long cables
  - USB
    - Complex, can work on medium long cables (5m)
  - RS-232
    - Very simple, uses 12V, can be sent over 15m, an evergreen protocol
  - TTL
    - Simple, uses 3V, usually for computer to boards communication
  - 1-Wire
    - Simple, uses 1 wire only, can support up to 10m distance, used for authentication of hardware (e.g. a printer cartridge)

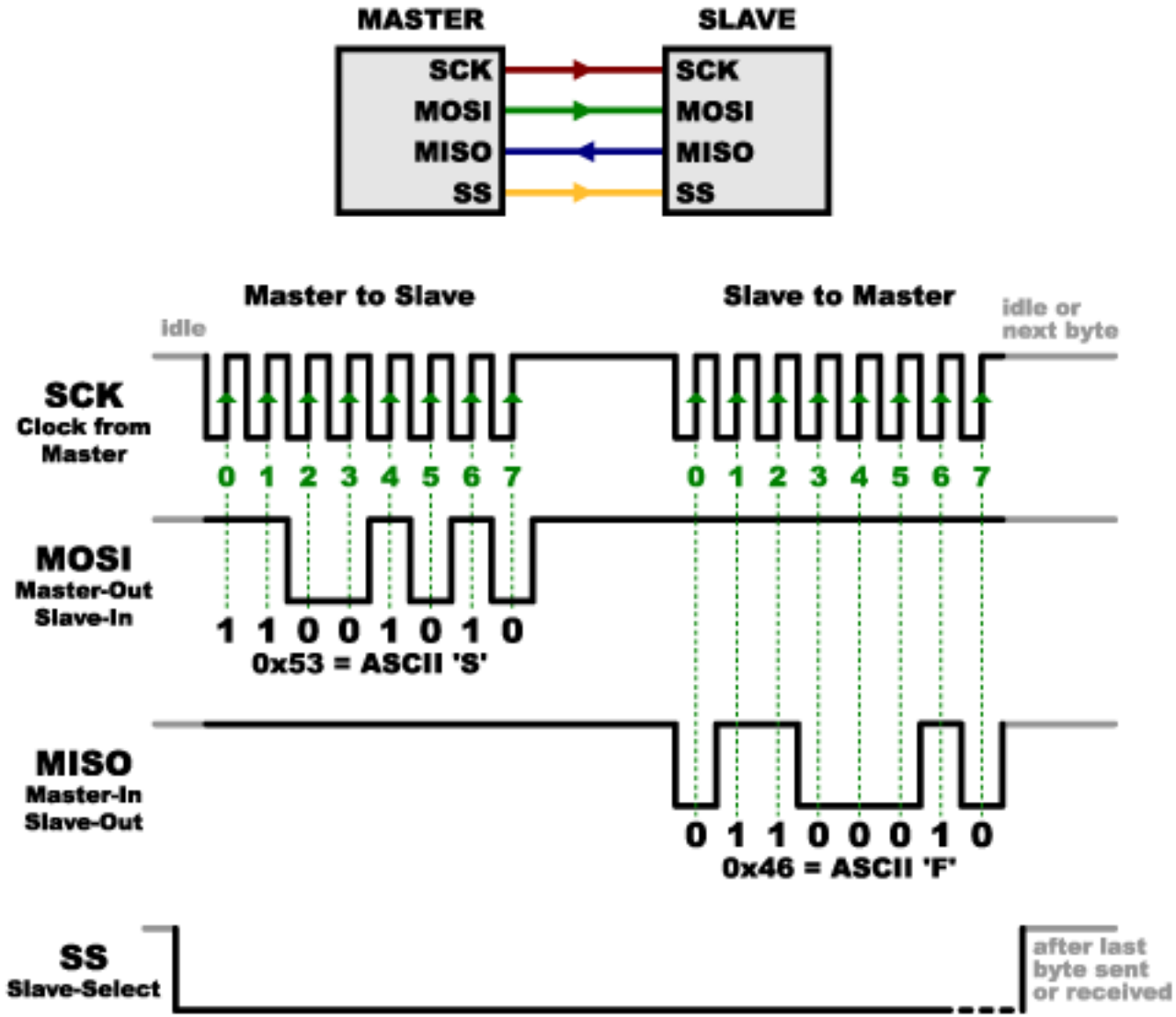
Characteristics of communication protocols

- How is the clock generated?
  - **Implicit:** all parties agree upon it ahead of time.
  - **Explicit**: there is a “clock generator” or “master”.
- How many hardware lines does it need?
- Synchronous or asynchronous?
  - **Synchronous**: all parties must send at the same time.
  - **Asynchronous**: parties send when they want to.
- Full or half duplex?
  - **Full duplex**: parties can transmit at the same time.
  - **Half duplex**: one party at the time can transmit.
- Is it point-to-point or is there addressing?
  - **Point to point**: only 2 parties.
  - **Networked**: several parties connected (with addresses or dedicated lines to identify sender/receiver).
- What is the maximum throughput of the system?
- How far can the signals travel?

Serial Peripheral Interface (SPI)

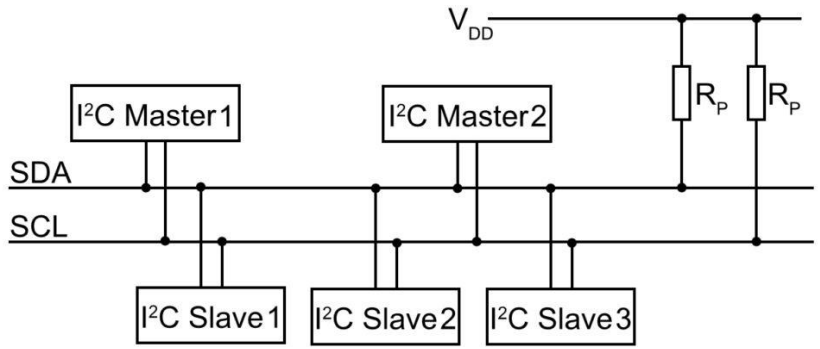


- Uses 4 wires (+ GND):
  - *Master-In Slave-Out (MISO)* or *Slave Data Out (SDO)*
    - Receive for the master
  - *Master-Out Slave-In (MOSI)* or *Slave Data In (SDI)*
    - Transmit for the master
  - *Clock (CLK or SCLK)*
    - Generated by the master, it is the clock for both sides of communication
  - *Chip Select (CS)* or *Slave Select (SS)*
    - Generated by the master, specifies which slave the master wants to talk to
- Synchronous protocol: master and slave both have to send data when the clock is going.
- Clock can be whatever (also very fast!), but high frequency limits distance between chips.
- Very simple! Usually chips have dedicated hardware for it to offload CPU.


MASTER
SLAVE
SCK
MOSI
MISO
SS
SCK
MOSI
MISO
SS
Master to Slave
Slave to Master
idle
idle or
next byte
SCK
Clock from
Master
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
MOSI
Master-Out
Slave-In
1 1 0 0 1 0 1 0
0x53 = ASCII 'S'
MISO
Master-In
Slave-Out
0 1 1 0 0 0 1 0
0x46 = ASCII 'F'
SS
Slave-Select
after last
byte sent
or received

Inter-integrated circuit (I^2C)

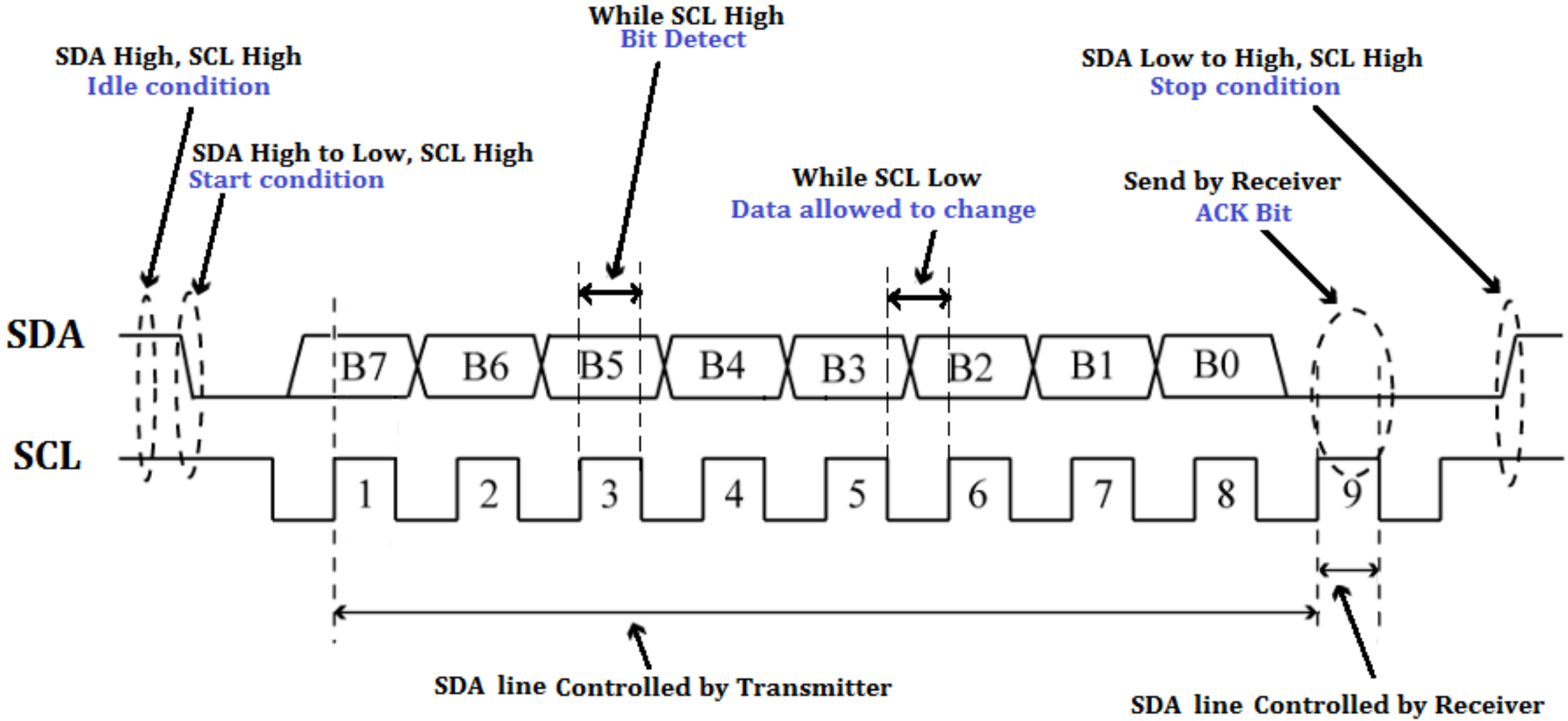
- **Bus** has a **master** which provides the clock and starts the interaction.
- However, the I2C bus master can change, allowing a peripheral to control the interaction.
- 3 wires:
  - SCL provides the clock
  - SDA which provides the data (half-duplex)
  - Ground



- Protocol allows multiple peripherals (and multiple masters) -> needs an **address scheme** on 7bits

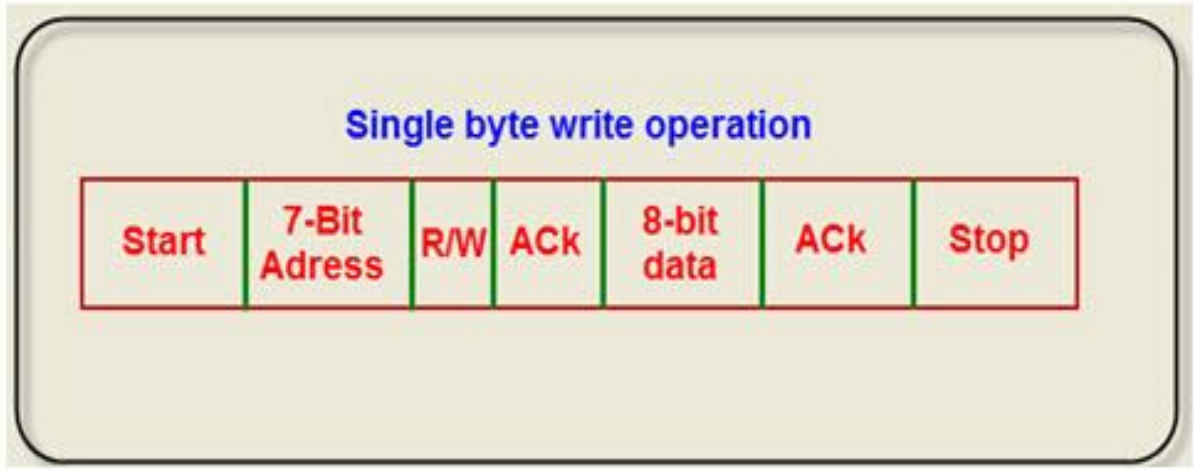
I2C protocol

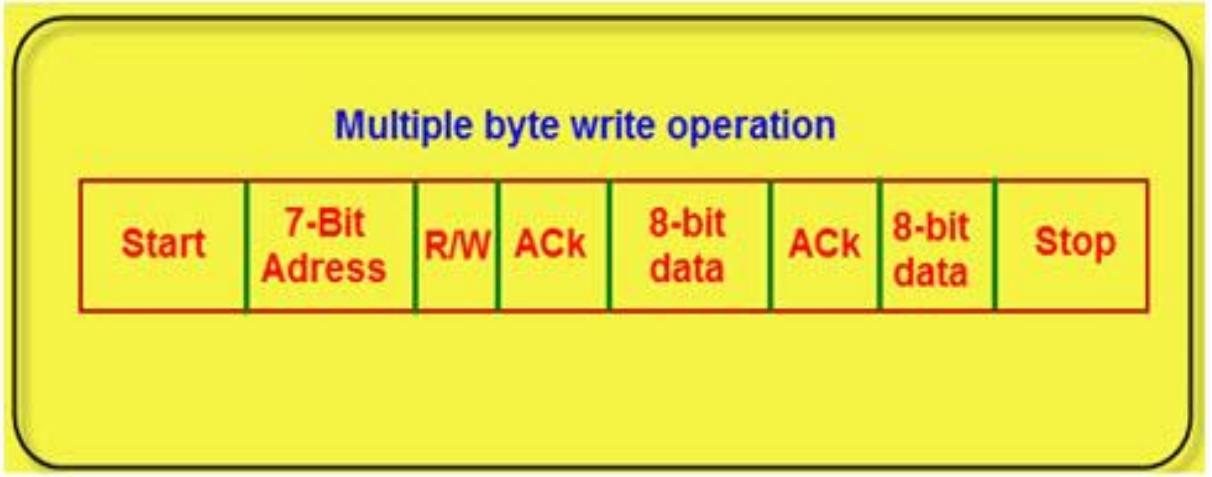
1. The master starts communication by sending an address and whether it wants to read or write to the slave.
2. The slave with that address then sends an acknowledgement (ACK).
3. Next, the master sends a command to (or reads from) the slave and the interaction proceeds.
4. When the communication is complete, the master sends a stop bit.


SDA High, SCL High
Idle condition
SDA High to Low, SCL High
Start condition
While SCL High
Bit Detect
While SCL Low
Data allowed to change
Send by Receiver
ACK Bit
SDA Low to High, SCL High
Stop condition
SDA
SCL
B7
B6
B5
B4
B3
B2
B1
B0
1
2
3
4
5
6
7
8
9
SDA line Controlled by Transmitter
SDA line Controlled by Receiver

Messages

- The first 7 bits are used for the **address**
- The 8th bit specifies if it's **read or write**
  - Read means read from the slave and write means write to the slave
- At least **1 byte of data** then follows
- Each byte is **acknowledged** by receiver
- Each message is started with a **start condition** and ended with a **stop condition**
- Only master is in charge of the clock
- After read operation, master waits for data to be transmitted by the slave





Robustness

- Noise from other circuits or cosmic rays can get in the middle.
  - You need to know that the data you received is the data that was transmitted.
- Some **redundancy** is needed to verify correctness of the data
- **Checksum**: a sum of “chunks” of the data (like 1 byte)
  - Example: if your buffer is {10, 20, 40, 60, 80, 90} then your checksum should be 300. If you are only using an 8-bit checksum, that becomes 44 (300 mod 256).
  - There are many other combinations that could sum to 44 value but those aren't likely to happen by chance.
- **Cyclic Redundancy Check** (CRC)
  - Based on the remainder of a polynomial division of their contents.

ESP32 communication means

- The ESP32 comes with hardware facilities for the following:
  - DMA
  - SPI
  - SDIO
  - SD cards
  - MMC cards
  - Ethernet
  - I2C
  - I2S
  - UART
  - RMT (infrared)
  - WiFi
  - Bluetooth

ESP32 I2C dedicated hardware

- ESP32 has two I2C controllers which are responsible for handling communications on two I2C buses.
- **RAM** (32 bytes)
  - Directly mapped onto the address space of the CPU cores.
  - Each byte of I2C data is stored in a 32-bit word of memory.
- **CMD_Controller** and 16 command registers
  - Used by the I2 C Master to control data transmission.
- **SCL_FSM**
  - A state machine that controls the SCL clock.
- **SDA_FSM**
  - A state machine that controls the SDA data line.
- **DATA_Shifter**
  - Converts the byte data to an outgoing bitstream, or converts an incoming bitstream to byte data.
- **SCL_Filter** and **SDA_Filter**
  - Input noise filter for the I2C_Slave.

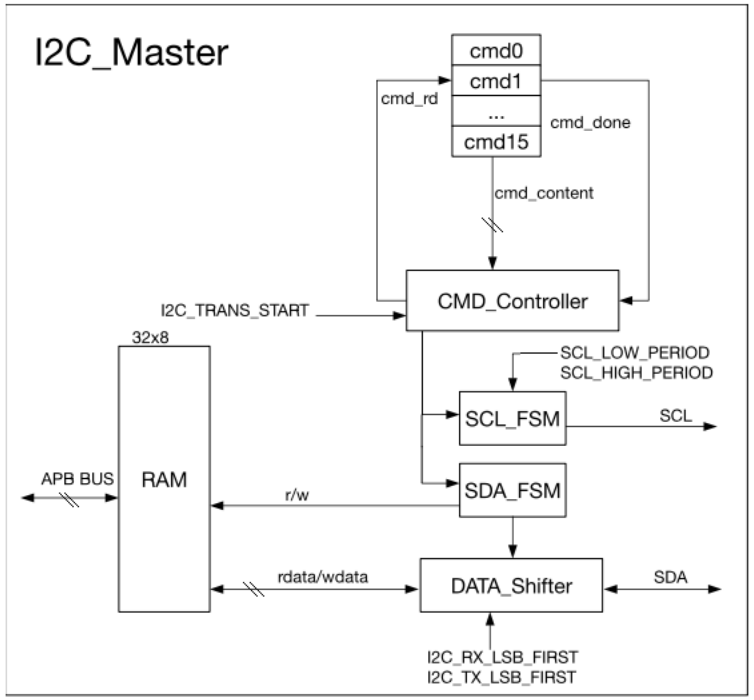
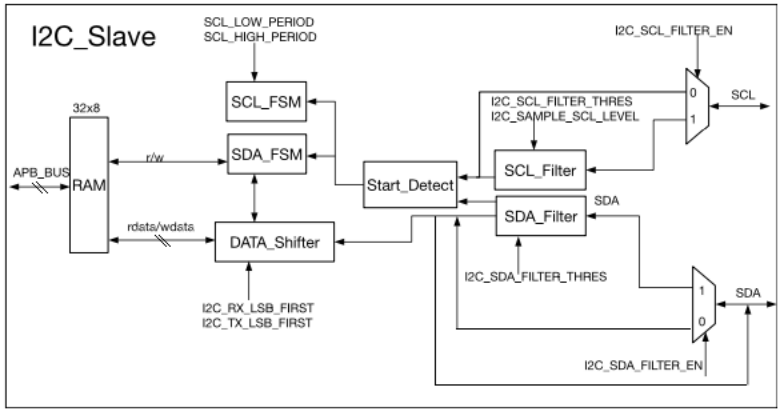


Figure 49: I²C Master Architecture

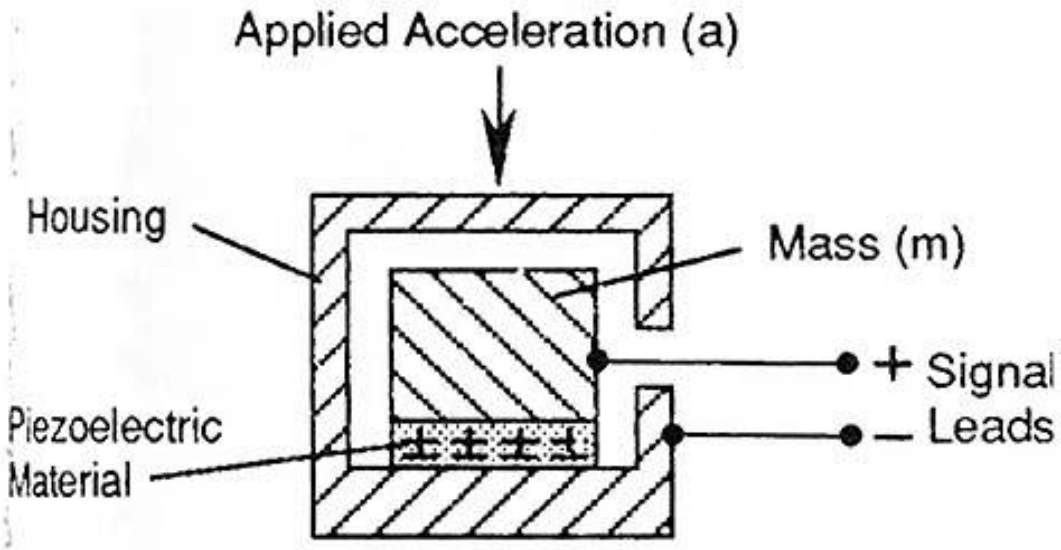


ESP-IDF and I2C

1. **Configure the driver**
   - Use i2c_param_config() and pass a i2c_config_t.
   - Specifies peripheral address, master/slave mode, clock speed, pins used.
2. **Install driver**
   - Use i2c_driver_install() and specify controller, buffers, and flags.
3. **Communicate (as master)**
   - i2c_cmd_link_create() creates a command
   - i2c_master_start() start bit
   - i2c_master_write_byte() writes address + specifies if read or write
   - i2c_master_write() writes data (1 or more bytes) or i2c_master_read() reads data
   - i2c_master_stop() stop bit
   - calling i2c_master_cmd_begin() send command
   - i2c_cmd_link_delete() free resources

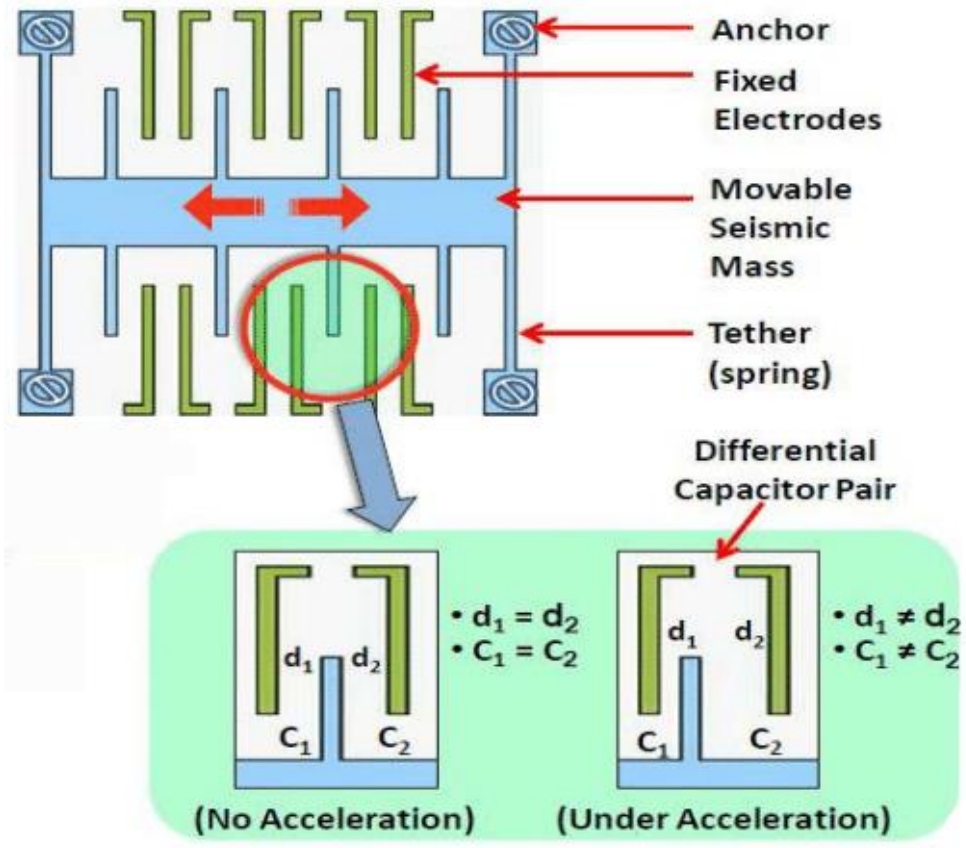
Industrial accelerometer

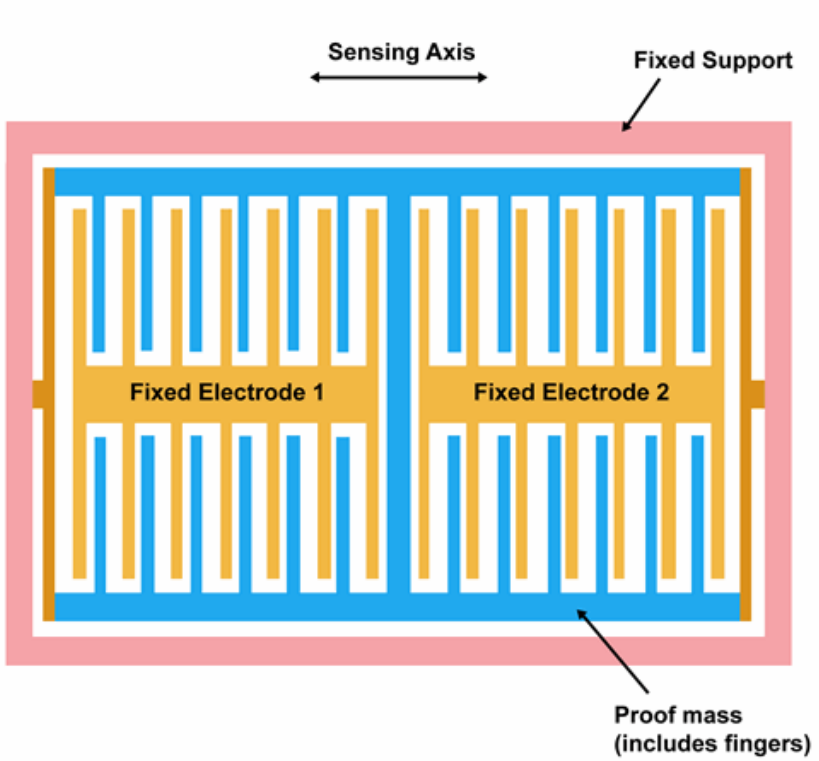
- Accelerometers are sensors that convert acceleration (motion) into an electrical signal.
- These devices are typically used to **measure vibration** on machines or structures in mines, highways, and bridges or buildings located in areas subject to earthquakes.
- It's possible to determine acceleration by **measuring a force and dividing it by the mass**. Most accelerometers use quartz or ceramic crystals to generate a piezoelectric effect that is converted into an electrical output.



Basic Accelerometer

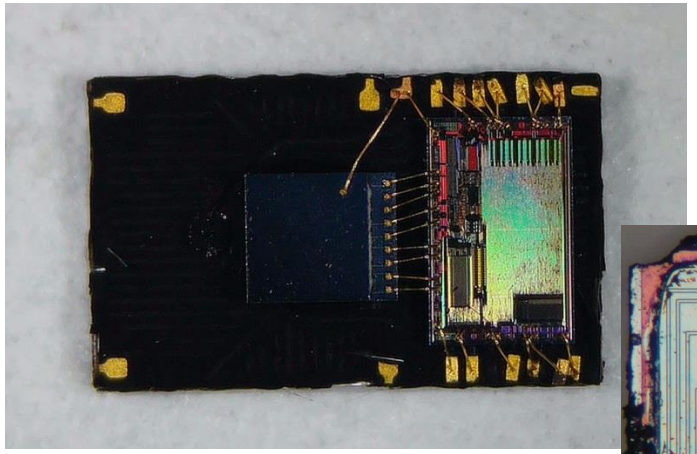
MEMS Accelerometer

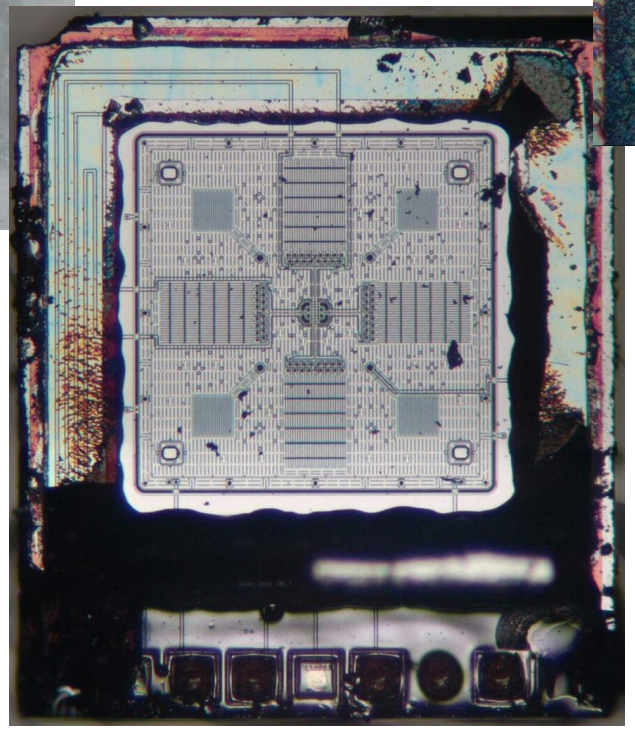


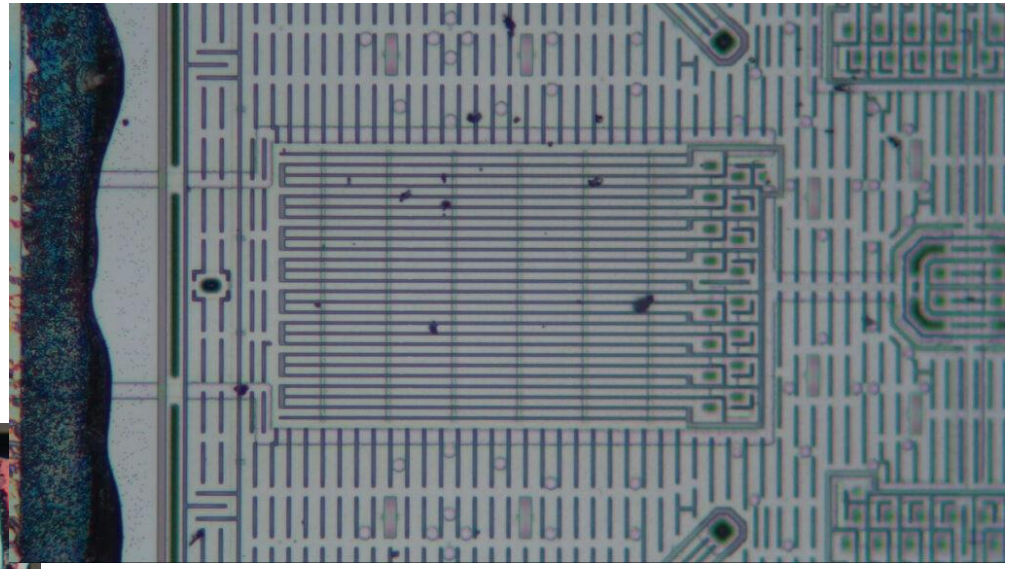


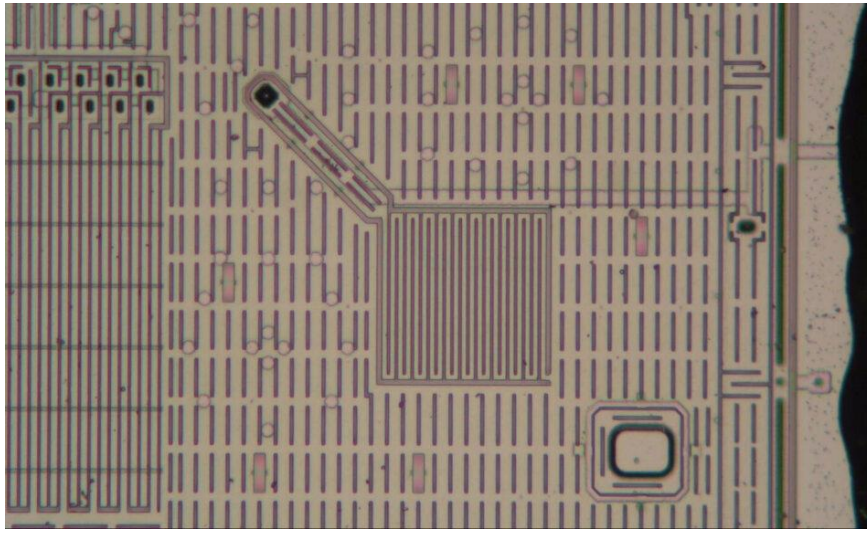
See how it works: https://www.youtube.com/watch?v=9X4frIQo7x0

Adxl345





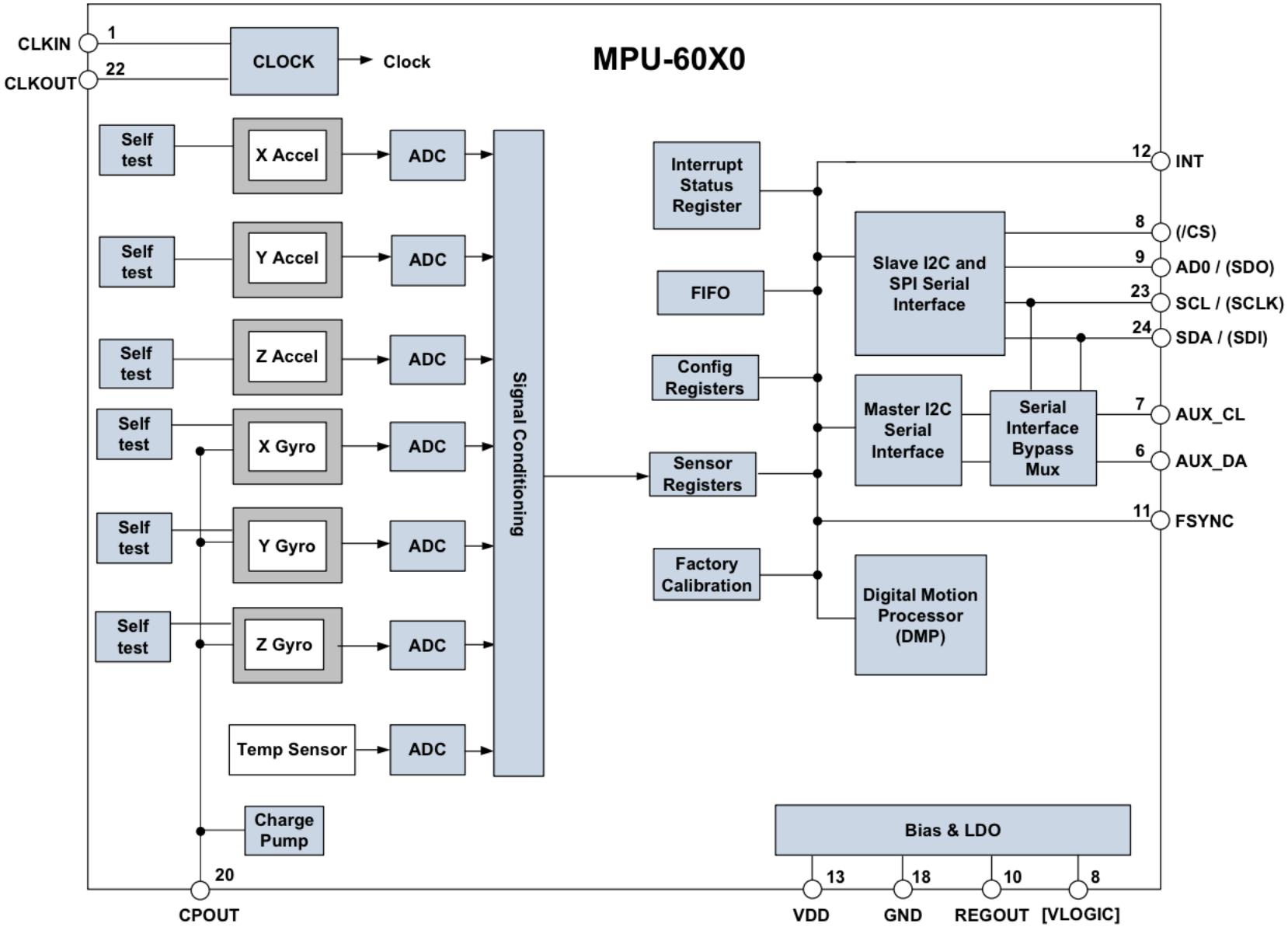




https://www.tinytransistors.net/2024/08/25/adxl345/

MPU6050

- Tri-Axis angular rate sensor (gyro)
- Tri-Axis accelerometer with a programmable full scale range of ±2*g*, ±4*g*, ±8*g* and ±16*g*
- Digital Motion Processing™ (DMP™) engine offloads complex MotionFusion, sensor timing synchronization and gesture detection
- Embedded algorithms for run-time bias and compass calibration in library.
- Digital-output temperature sensor
- Programmable interrupt supports gesture recognition, panning, zooming, scrolling, and shake detection
- VDD Supply voltage range of 2.375V–3.46V
- Full Chip Idle Mode Supply Current: 5µA
- 400kHz Fast Mode I²C or up to 20MHz SPI (MPU-6000 only) serial host interfaces


MPU-60X0
CLKIN
1
CLKOUT
22
CLOCK
Clock
Self test
X Accel
ADC
Self test
Y Accel
ADC
Self test
Z Accel
ADC
Self test
X Gyro
ADC
Self test
Y Gyro
ADC
Self test
Z Gyro
ADC
Temp Sensor
ADC
Signal Conditioning
Charge Pump
20
CPOUT
Interrupt Status Register
FIFO
Config Registers
Sensor Registers
Factory Calibration
Slave I2C and SPI Serial Interface
Master I2C Serial Interface
Serial Interface Bypass Mux
Digital Motion Processor (DMP)
12
INT
8
(/CS)
9
AD0 / (SDO)
23
SCL / (SCLK)
24
SDA / (SDI)
7
AUX_CL
6
AUX_DA
11
FSYNC
Bias & LDO
13
VDD
18
GND
10
REGOUT
8
[VLOGIC]

Sample Rate Divider register

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
19	25	SMPLRT_DIV[7:0]							

Description:

This register specifies the divider from the gyroscope output rate used to generate the Sample Rate for the MPU-60X0.

The sensor register output, FIFO output, and DMP sampling are all based on the Sample Rate.

The Sample Rate is generated by dividing the gyroscope output rate by *SMPLRT_DIV*:

Sample Rate = Gyroscope Output Rate / (1 + *SMPLRT_DIV*)

where Gyroscope Output Rate = 8kHz when the DLPF is disabled (*DLPF_CFG* = 0 or 7), and 1kHz when the DLPF is enabled (see Register 26).

Note: The accelerometer output rate is 1kHz. This means that for a Sample Rate greater than 1kHz, the same accelerometer sample may be output to the FIFO, DMP, and sensor registers more than once.

For a diagram of the gyroscope and accelerometer signal paths, see Section 8 of the MPU-6000/MPU-6050 Product Specification document.

Parameters:

SMPLRT_DIV — 8-bit unsigned value. The Sample Rate is determined by dividing the gyroscope output rate by this value.

TEMP_OUT registers

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
41	65	TEMP_OUT[15:8]							
42	66	TEMP_OUT[7:0]							

Description:

These registers store the most recent temperature sensor measurement.

Temperature measurements are written to these registers at the Sample Rate as defined in Register 25.

These temperature measurement registers, along with the accelerometer measurement registers, gyroscope measurement registers, and external sensor data registers, are composed of two sets of registers: an internal register set and a user-facing read register set.

The data within the temperature sensor's internal register set is always updated at the Sample Rate. Meanwhile, the user-facing read register set duplicates the internal register set's data values whenever the serial interface is idle. This guarantees that a burst read of sensor registers will read measurements from the same sampling instant. Note that if burst reads are not used, the user is responsible for ensuring a set of single byte reads correspond to a single sampling instant by checking the Data Ready interrupt.

The scale factor and offset for the temperature sensor are found in the Electrical Specifications table (Section 6.4 of the MPU-6000/MPU-6050 Product Specification document).

The temperature in degrees C for a given register value may be computed as:

Temperature in degrees C = (TEMP_OUT Register Value as a signed quantity)/340 + 36.53

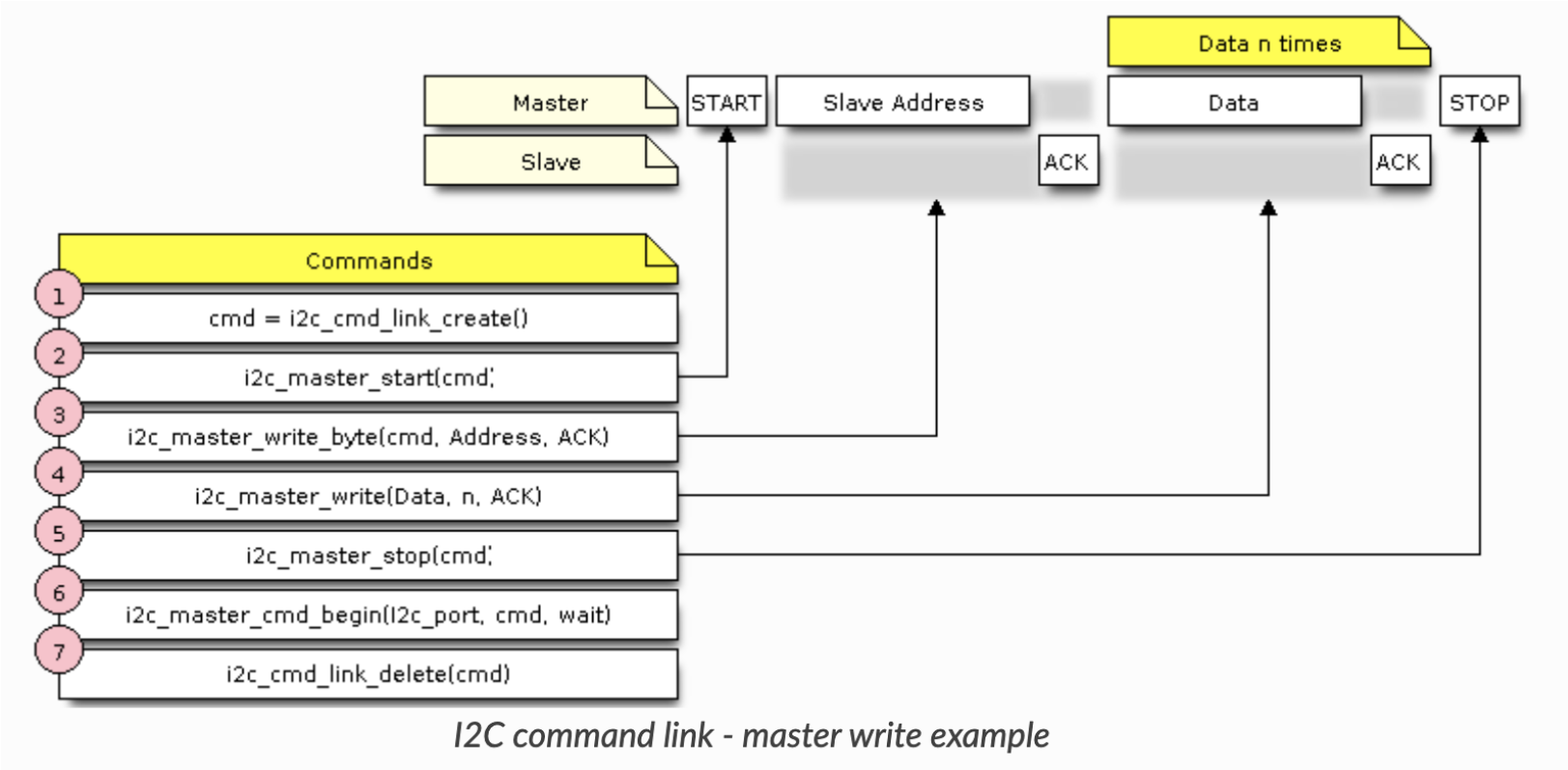
Please note that the math in the above equation is in decimal.

Parameters:

TEMP_OUT 16-bit signed value.

Stores the most recent temperature sensor measurement.

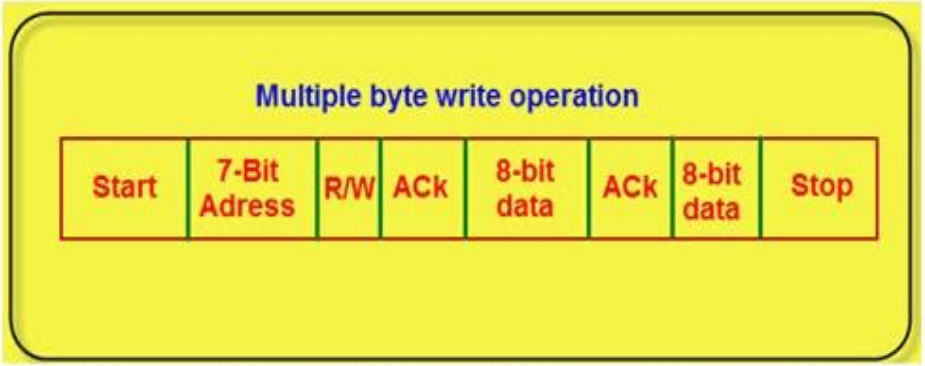
ESP-IDF: write on I2C

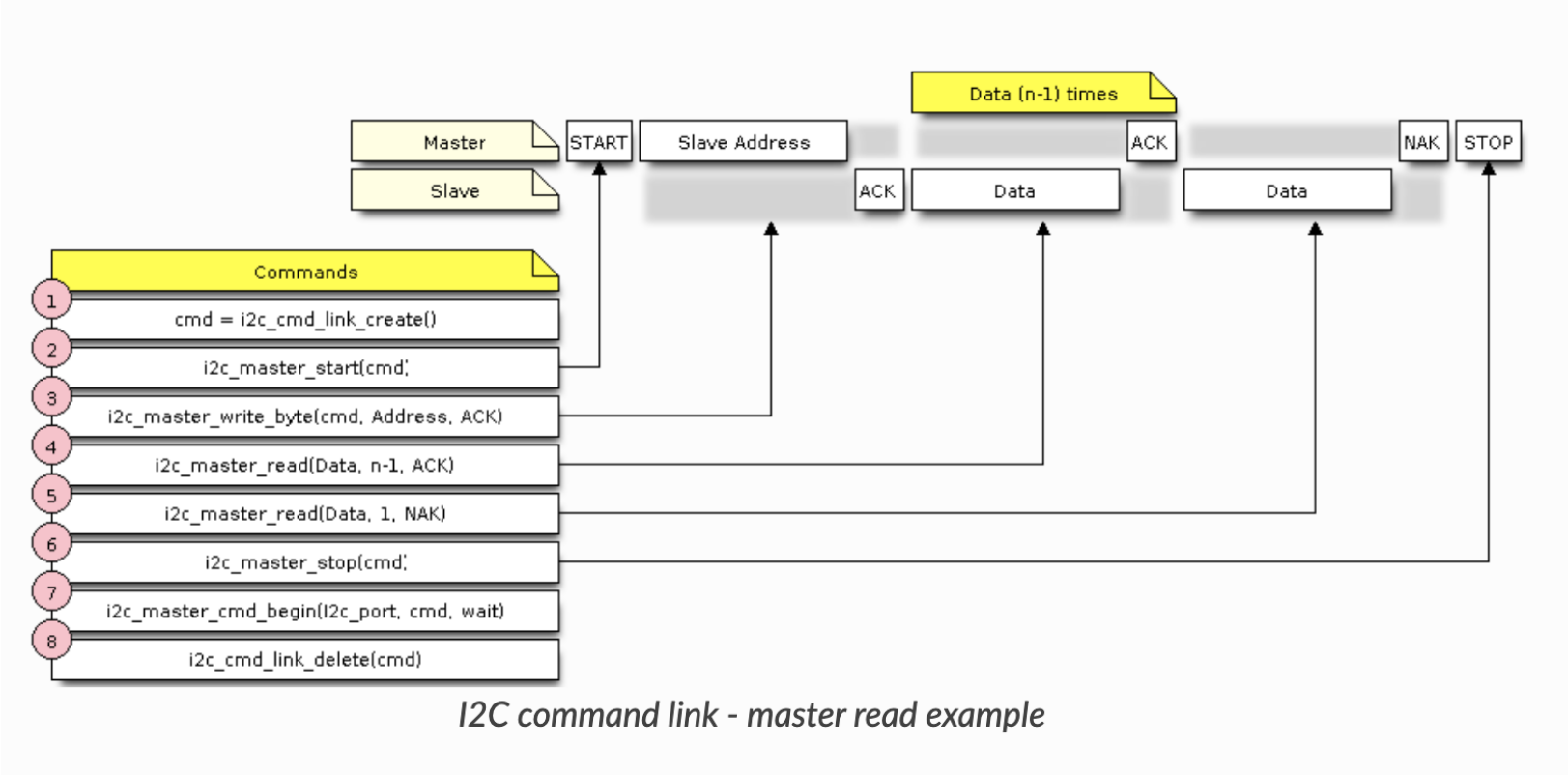


I2C command link - master write example

https://docs.espressif.com/projects/esp-idf/en/v5.0.2/esp32/api-reference/peripherals/i2c.html

ESP-IDF: read on I2C





I2C command link - master read example

https://docs.espressif.com/projects/esp-idf/en/v5.0.2/esp32/api-reference/peripherals/i2c.html

Live coding

- Communicating with the MPU6050 accelerometer.

Example exam questions

- What strategies exist to reduce power consumption in an embedded system?
- Provide examples of sleep modes on a microcontroller.
- What are the 5 power modes of the ESP32?
- How can FreeRTOS allow low power modes?
- What types of peripherals use digital communication protocols?
- Describe the Inter-integrated circuit (I^2C) protocol
- If you have to connect two system located in a production line of a factory, which would you use among: Bluetooth, RS232 and SPI?
- Compute the checksum of the following sequence of bytes, using a 1 byte checksum: {01, 10, 05, 04, 25, 15}