# Microcontroller Basics

## Signal Groups

### Data lines

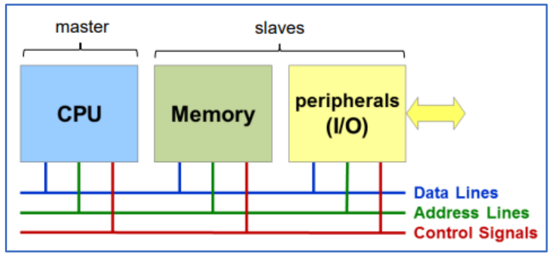
- 8, 16, 32, 64 parallel lines of data
- Bidirectional (read / write)

### Address lines

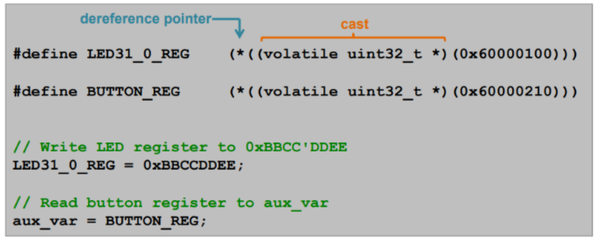
- Unidirectional: From master to slave
- Number of lines → size of address space

### Control signals

- Control read / write direction
- Provide timing information



## Accessing Control Registers in C

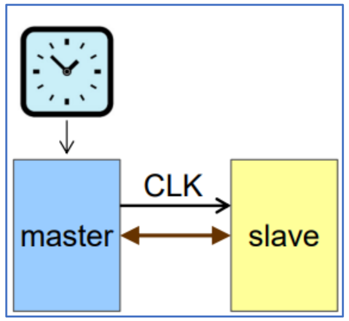


```
#define LED31_0_REG     (*((volatile uint32_t *)(0x60000100)))

#define BUTTON_REG      (*((volatile uint32_t *)(0x60000210)))

// Write LED register to 0xBBCC'DDEE
LED31_0_REG = 0xBBCCDDEE;

// Read button register to aux_var
aux_var = BUTTON_REG;
```

## Timing Options

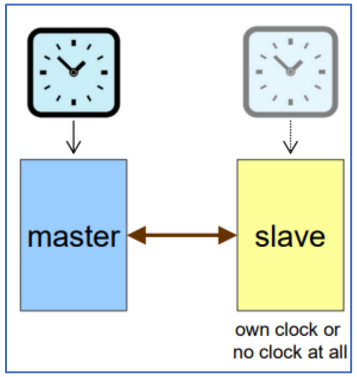
Synchronous

- *Master* and *slave* use a common clock
- Clock edges control bus transfer on both sides
- Used by most on-chip busses
- Off-chip: DDR and synchronous RAM



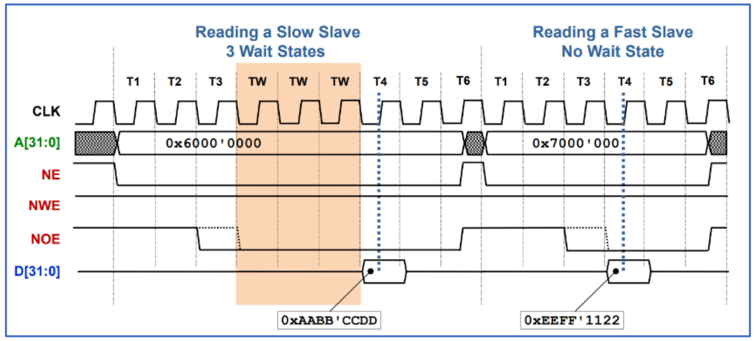
Asynchronous

- *Slaves* have no access to the clock of the master
- Control signals carry timing information to allow synchronization
- Widely used for low data-rate off-chip memories



## Slow Slaves

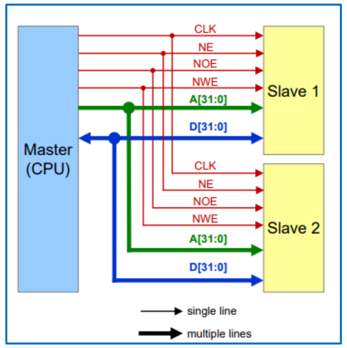
- Wait states are inserted depending on the address of an access



# Microcontroller Basics

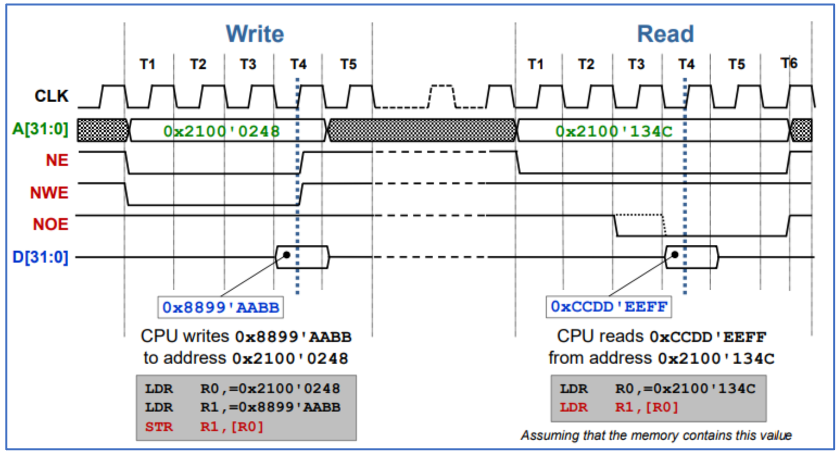
## Block Diagramm

- Address lines A[31:0]
- Data lines D[31:0]
- Control
  - CLK Clock
  - NE **N**ot **E**nable
  - NWE **N**ot **W**rite **E**nable
  - NOE **N**ot **O**utput **E**nable



## Timing Diagram

- *Write* D[:] to A[:]: NE = 0, NWE = 0
- *Read* D[:] from A[:]: NE = 0, NOE = 0



CPU writes 0x8899'AABB to address 0x2100'0248

```
LDR   R0,=0x2100'0248
LDR   R1,=0x8899'AABB
STR   R1,[R0]
```

CPU reads 0xCCDD'EEFF from address 0x2100'134C

```
LDR   R0,=0x2100'134C
LDR   R1,[R0]
```

*Assuming that the memory contains this value*

**Bus Access Size** is determined by the NBL (0 - 3) (**N**o **B**yte **L**ine) signals.

- NBL = 1 → Byte used for Read / Write
- NBL[0:3] = 0011 → Read Half-Word
- NBL[0:3] = 0010 → Read Byte

Control Bits

- Allow CPU to <u>configure</u> *slave*
- CPU writes to register bit
- *Slave* uses output of register bit
- Usually read/write

Status Bits

- Allow CPU to <u>monitor</u> *slave*
- *Slave* write into register bit
- CPU reads register bit
- Usually read-only

## Address Decoding

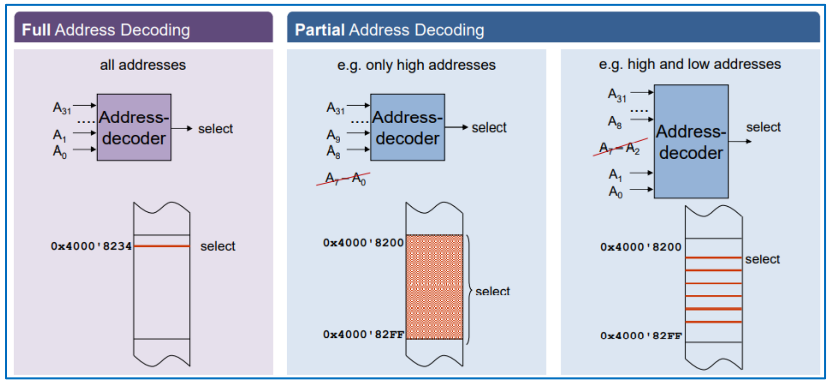
Interpretation of address line values. See wheather bus access targets a particular address or address range.

Full Address Decoding

- **All** address lines are decoded
- A control register can be accessed at **exactly one location**
- **1:1** mapping – **A** unique address maps to a single hardware register

Partial Address Decoding

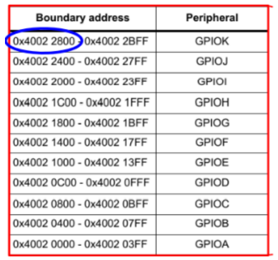
- **Only a sub-set** of the address lines is decoded
- Detects an address range or a **set of addresses**
- **N:1** mapping – **N** unique addresses map to the same hardware register
- Map a hardware register to several addresses



# GPIO - General Purpose Input / Output

Register address = Base address + Offset

- Offset is given for each register in reference manual
- Base address defined in memory map (*reference manual*)



| Boundary address | Peripheral |
|---|---|
| 0x4002 2800 - 0x4002 2BFF | GPIOK |
| 0x4002 2400 - 0x4002 27FF | GPIOJ |
| 0x4002 2000 - 0x4002 23FF | GPIOI |
| 0x4002 1C00 - 0x4002 1FFF | GPIOH |
| 0x4002 1800 - 0x4002 1BFF | GPIOG |
| 0x4002 1400 - 0x4002 17FF | GPIOF |
| 0x4002 1000 - 0x4002 13FF | GPIOE |
| 0x4002 0C00 - 0x4002 0FFF | GPIOD |
| 0x4002 0800 - 0x4002 0BFF | GPIOC |
| 0x4002 0400 - 0x4002 07FF | GPIOB |
| 0x4002 0000 - 0x4002 03FF | GPIOA |

## GPIO

Situation

- Microcontroller as general-purpose device
- Many functional blocks included

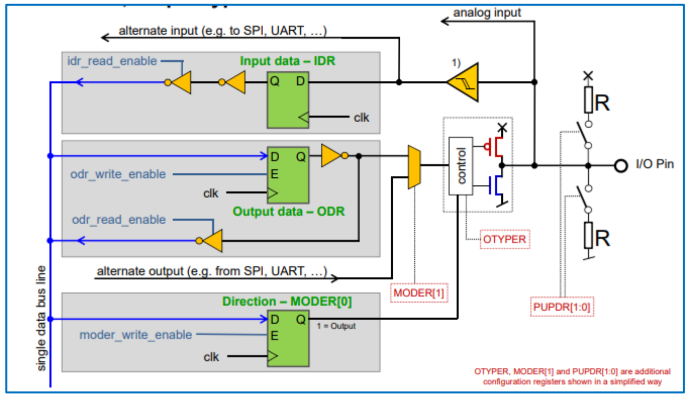
Problem

- Limited number of pins
- For a specific configuration, not all functions can be routed to I/O pins

Solution

- Many (all) pings are configurable
- Select the needed I/O pins / functions
- «pin sharing»
- Output multiplexer needs to be configured

## Structure



## Configuration Registers

| | | |
|---|---|---|
| GPIOx_MODER[1:0] | Direction | e.g. Input, Analog mode, … |
| GPIOx_OTYPER[0:0] | Output type | Push-Pull / Open-Drain (Low) |
| GPIOx_PUPDR[1:0] | Pull-Up / Pull-Down | |
| GPIOx_OSPEEDR[1:0] | Speed | Low, Medium, … |

## Setting and Clearing Bits - GPIOx_BSRR

| | |
|---|---|
| 0-15 Set Bits | Set port bit by writing a '1' to BSRR[bit] |
| 16-31 Clear Bits | Clear port bit by writing a '1' to BSRR[bit+16] |

- Ensures atomic access in software (no interruption possible)

## Data operations

| | |
|---|---|
| Input | Read register GPIOx_IDR |
| Output | Write register GPIOx_ODR or GPIOx_BSRR |

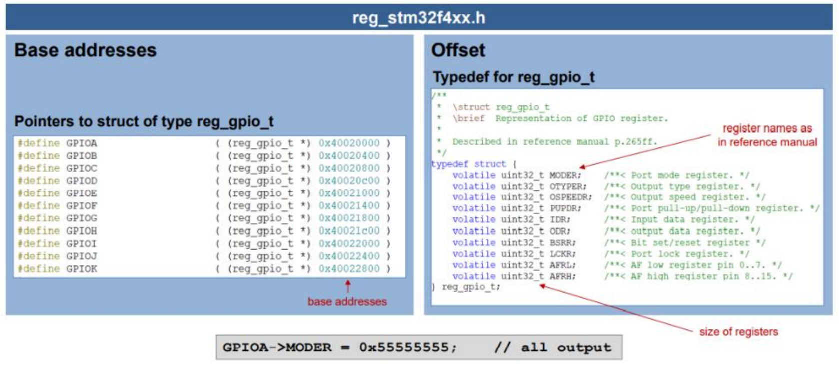
## Hardware Abstraction Layer (HAL)

```
#define ADDR (* ((volatile uintXX_t *) (0x40020000)))
```

```
#define GPIOA_MODER     (*((volatile uint32_t *)(0x40020000)))
```

Accessing a register

- Each GPIO port has the same 10 registers
- There are 11 GPIO ports → GPIOA – GPIOK

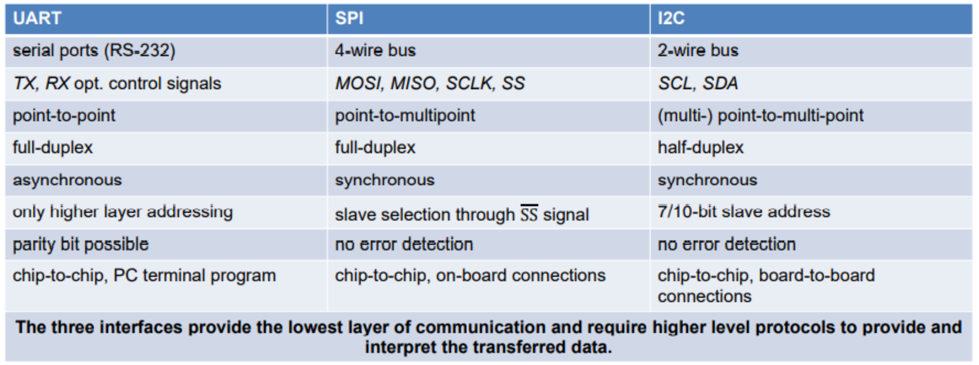


```
#define GPIOA                  ( (reg_gpio_t *) 0x40020000 )
#define GPIOB                  ( (reg_gpio_t *) 0x40020400 )
#define GPIOC                  ( (reg_gpio_t *) 0x40020800 )
#define GPIOD                  ( (reg_gpio_t *) 0x40020c00 )
#define GPIOE                  ( (reg_gpio_t *) 0x40021000 )
#define GPIOF                  ( (reg_gpio_t *) 0x40021400 )
#define GPIOG                  ( (reg_gpio_t *) 0x40021800 )
#define GPIOH                  ( (reg_gpio_t *) 0x40021c00 )
#define GPIOI                  ( (reg_gpio_t *) 0x40022000 )
#define GPIOJ                  ( (reg_gpio_t *) 0x40022400 )
#define GPIOK                  ( (reg_gpio_t *) 0x40022800 )
```

```
/**
 *  \struct reg_gpio_t
 *  \brief  Representation of GPIO register.
 *
 *  Described in reference manual p.265ff.
 */
typedef struct {
    volatile uint32_t MODER;     /**< Port mode register. */
    volatile uint32_t OTYPER;    /**< Output type register. */
    volatile uint32_t OSPEEDR;   /**< Output speed register. */
    volatile uint32_t PUPDR;     /**< Port pull-up/pull-down register. */
    volatile uint32_t IDR;       /**< Input data register. */
    volatile uint32_t ODR;       /**< output data register. */
    volatile uint32_t BSRR;      /**< Bit set/reset register */
    volatile uint32_t LCKR;      /**< Port lock register. */
    volatile uint32_t AFRL;      /**< AF low register pin 0..7. */
    volatile uint32_t AFRH;      /**< AF high register pin 8..15. */
} reg_gpio_t;
```

```
GPIOA->MODER = 0x55555555;    // all output
```

# Serial Connection – Overview / SPI



| UART | SPI | I2C |
|---|---|---|
| serial ports (RS-232) | 4-wire bus | 2-wire bus |
| *TX, RX* opt. control signals | *MOSI, MISO, SCLK, SS* | *SCL, SDA* |
| point-to-point | point-to-multipoint | (multi-) point-to-multi-point |
| full-duplex | full-duplex | half-duplex |
| asynchronous | synchronous | synchronous |
| only higher layer addressing | slave selection through $\overline{SS}$ signal | 7/10-bit slave address |
| parity bit possible | no error detection | no error detection |
| chip-to-chip, PC terminal program | chip-to-chip, on-board connections | chip-to-chip, board-to-board connections |

**The three interfaces provide the lowest layer of communication and require higher level protocols to provide and interpret the transferred data.**

<u>I2C – Inter-Integrated Circuit</u>

- Synchronous half-duplex transmission (SCL, SDA)
- 7-bit slave addresses

<u>UART - Asynchronous Serial Interface</u>

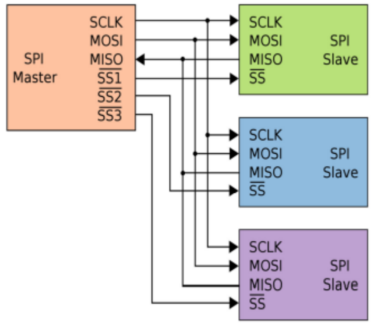
- Transmitter and receiver use diverging clocks
- Synchronization using start/stop bits → overhead
- Longer connections require line drivers → RS-232/RS-485

<u>SPI – Serial Peripheral Interface</u>

- Master / Slave
- Synchronous full-duplex transmission (MOSI, MISO)
- Selection of device through Slave Select ($\overline{SS}$)
- No acknowledge, no error detection
- Four mode → clock polarity and clock phase

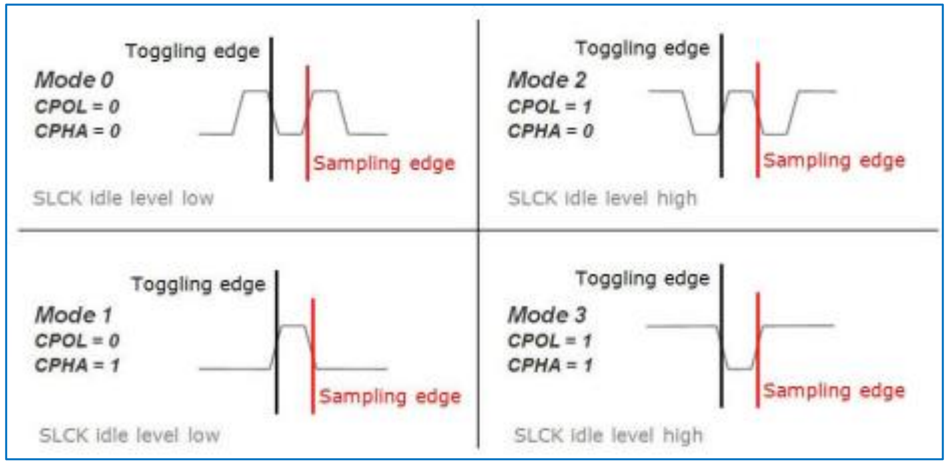
**Single Master – Multiple Slaves**

- Master generates a common clock signal for all slaves
- MOSI  From **M**aster **O**utput to all **S**lave **I**nputs
- MISO  All slave outputs connected to single master input
- Slaves
  - Individual select $\overline{SS1}, \overline{SS2}, \overline{SS3}$
  - $\overline{SSx} = '1'$ → Slave output MISOx is tri-state



**Clock Polarity (CPOL) and Clock Phase (CPHA)**

- TX provides data on 'Toggling Edge'
- RX takes over data with 'Sampling Edge'



## Serial Connection - SPI

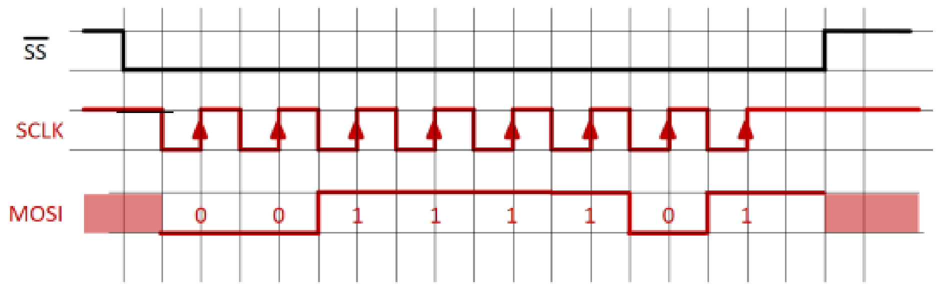
### Properties

- No defined addressing scheme
  - Use of $\overline{SS}$ instead → KISS
- Transmission without receive acknowledge and error detection
  - Has to be implemented in higher level protocols
- Originally used only for transmission of single bytes
  - $\overline{SS}$ deactivated after each byte
  - Today also used for streams
- Data rate
  - Highly flexible as clock signal is transmitted
- No flow-control available
  - Master can delay the next clock edge
  - Slave can't influence the data rate
- Susceptible to spikes on clock line

### Example

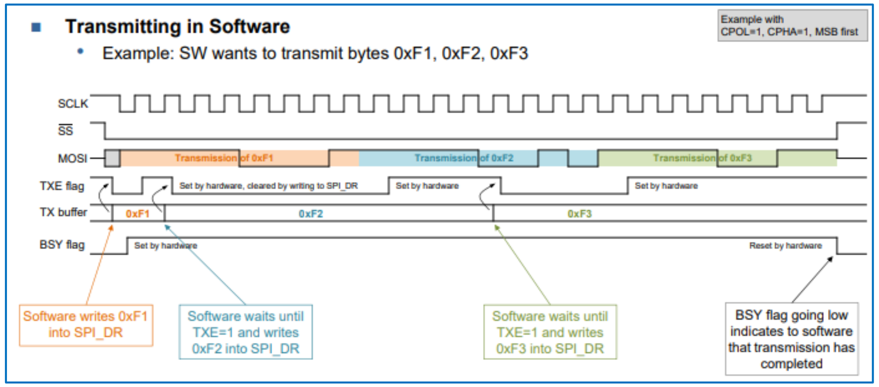
Ein Prozessor (SPI Master) sendet das Byte 0x3D $= 0011\ 1101$. Die Schnittstelle ist wie folgt konfiguriert:

$$Mode = 3, \quad CPOL = 1, \quad CPHA = 1, \quad MSB - First$$

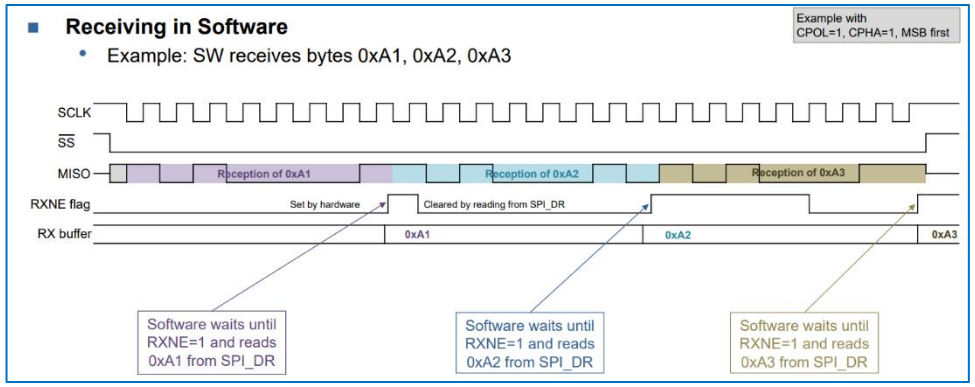


### Synchronizing Hardware and Software

| | | |
|---|---|---|
| TXE | TX Buffer Empty | Software can write next TX Byte to register SPI_DR |
| RXNE | RX Buffer Not Empty | A byte has been received. Software can read it from SPI_DR |



Transmitting in Software
Example: SW wants to transmit bytes 0xF1, 0xF2, 0xF3
Example with CPOL=1, CPHA=1, MSB first
SCLK
SS
MOSI
Transmission of 0xF1
Transmission of 0xF2
Transmission of 0xF3
TXE flag
Set by hardware, cleared by writing to SPI_DR
Set by hardware
Set by hardware
TX buffer
0xF1
0xF2
0xF3
BSY flag
Set by hardware
Reset by hardware
Software writes 0xF1 into SPI_DR
Software waits until TXE=1 and writes 0xF2 into SPI_DR
Software waits until TXE=1 and writes 0xF3 into SPI_DR
BSY flag going low indicates to software that transmission has completed

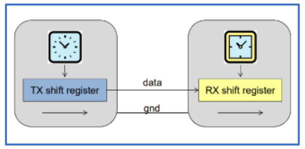


Receiving in Software
Example: SW receives bytes 0xA1, 0xA2, 0xA3
Example with CPOL=1, CPHA=1, MSB first
SCLK
SS
MISO
Reception of 0xA1
Reception of 0xA2
Reception of 0xA3
RXNE flag
Set by hardware
Cleared by reading from SPI_DR
RX buffer
0xA1
0xA2
0xA3
Software waits until RXNE=1 and reads 0xA1 from SPI_DR
Software waits until RXNE=1 and reads 0xA2 from SPI_DR
Software waits until RXNE=1 and reads 0xA3 from SPI_DR

## UART / I2C – Universal Synchronous Receiver Transmitter / Inter-Integrated Circuit

### Universal Asynchronous Receiver Transmitter – UART

Connecting shift registers with diverging clock sources

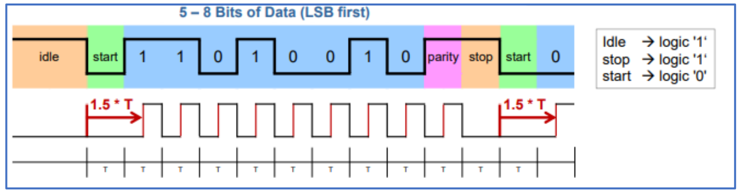


- Same target frequency
- Different tolerances and divider ratios
- Requires synchronization at start of each data item in receiver

### UART Timing

Transition stop (‘1’) → start (‘0’)

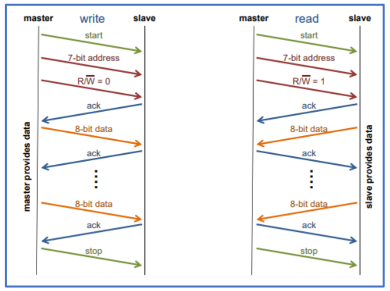
- Receiver detects edge at the start of each data block (5 to 8 bits)
- Allows receiver to sample data «in middle of bits» → red edges
- Clocks have to be accurate enough to allow sampling up to parity bit



### UART Characteristics

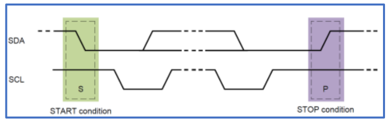
- Synchronization
  - Each data item (5-8 bits) requires synchronization
- Asynchronous data transfer
  - Mismatch of clock frequencies in TX and RX
  - Requires overhead for synchronization → additional bits
  - Requires effort for synchronization → additional hardware
- Advantage
  - Clock does not have to be transmitted
  - Transmission delays are automatically compensated
- On-board connections
  - Signal levels are 3V or 5V with reference to ground
  - Off-board connections require strong output drivers

### I2C Bus - I2C – Inter-Integrated Circuit



- Bidirectional 2-wire
  Clock → SCL  Data → SDA
- Synchronous, half-duplex
- Each device on bus addressable
- 8-bit oriented data transfer
- Different bit rates up to 5 Mbit/s
- Suited for connection of multiple boards
- Multi-master possible

### I2C Bus – Operation



- Master drives clock line (SCL)
- Master initiates / terminates transaction through START / STOP condition

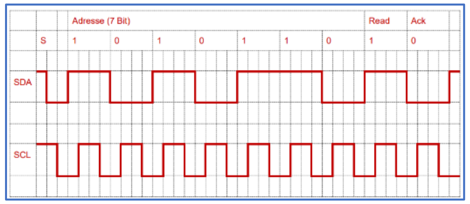
### I2C Bus – Driving Data on SDA

- Data driven onto SDA by master or by addressed slave
  - Depending on transaction (read/write) and point in time
  - Change of data only allowed when SCL is low
  - Allow detection of START and STOP condition

### I2C Bus – Data Transfer on I2C

- 8-bit oriented transfers
- Bit 9: Receiver acknowledges by driving SDA low
- Master defines number of 8-bit transfers (STOP)

<u>Example:</u> Ein Baustein soll zum lesen adressiert werden (7-Bit: 0x56)



## Timer / Counter

Binary up-counter or down-counter

- Counts events / clock pulses or external signals
- Output after a defined number of events
- Timer: counting clock cycles or processor cycles
- Counter: counting events

Use

- Count events Measure of time, frequencies, phases, periods
- Generate intervals, row of pulses, interrupts

Function

- Configure in up- or down-counting mode
- Select source
- 16-bit / 32-bit counter register
- Set interrupt flag → trigger interrupt

Prescaler

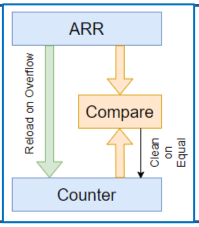
- Increase counting range
- Count only every n-th event

Insert-Capture

- Measuring intervals → puls lengths and periods
  → Counts ticks between timer start and an event

Up-counting mode

- From 0 to ARR
- Restarts from 0
- Generates overlow



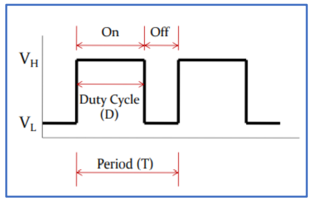
Down-counting mode

- From ARR to 0
- Restart from ARR
- Generates underflow

**Pulse-Width-Modulation PWM**

Duty Cycle – Definition

$$Duty\ Cycle = \frac{On\ Time}{Period}$$

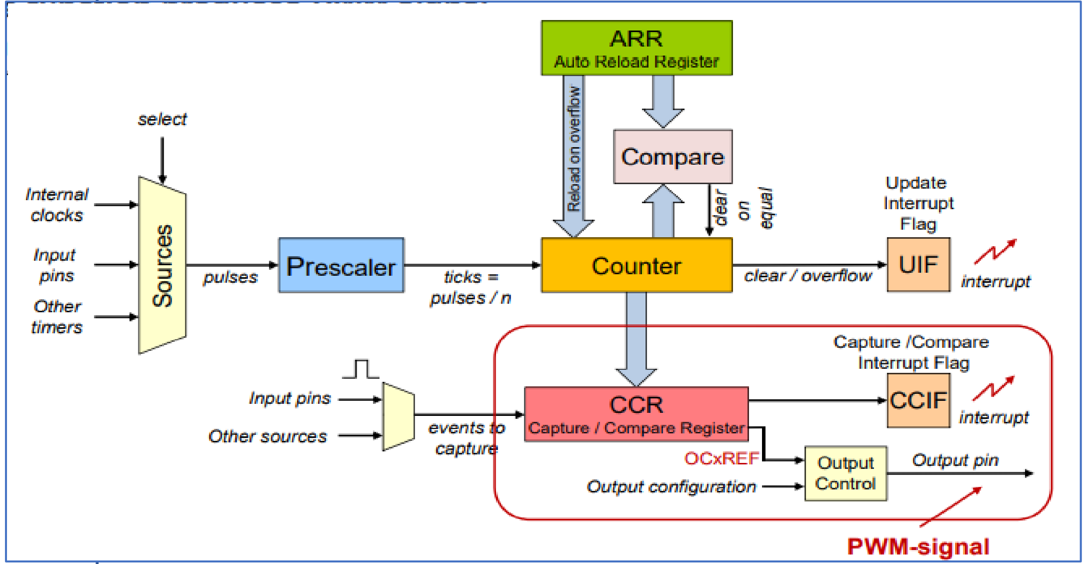


Average signal

$$V_{avg} = D \cdot V_H + (1 - D) \cdot V_L$$

Compare function produces PWM signal

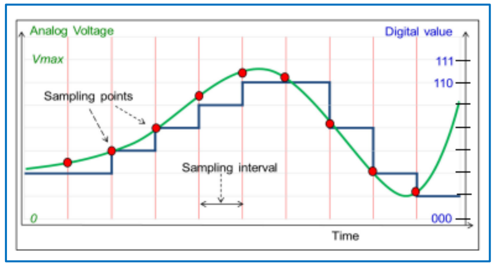
- Toggle output pin when counter reaches CCR
- Mode = 1: UP: $CNT < CCR$ DOWN: $CNT \leq CCR$
- Mode = 2: UP: $CNT \geq CCR$ DOWN: $CNT > CCR$



# ADC / DAC

**ADC – Analog to Digital Converter**

- Converts input signal (voltage) to a digital value (N-bit)
- Conversion results in one of $2^N$ possible numerical levels
- Raw input signal can be dynamic or static
    - Dynamic signal (green) sampled at specific time intervals
    - Samples transformed into series of discrete values (blue)



**Input signals**

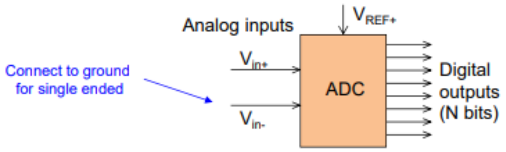
- Differential inputs

$V_{in} = V_{in+} - V_{in-}$

- $V_{in+}$ signal to convert (non-inverting input)
- $V_{in-}$ signal to convert (inverting input)

**Signal ended mode**

- Only $V_{in+}$ used
- $V_{in-}$ is grounded



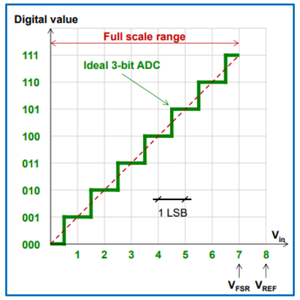
**Reference voltage $V_{REF+}$**

- Internal or external stable voltage
- Needed to weight input voltage

$V_{in} = (\textit{digital value}) * V_{REF+} \ / \ (2^N)$

**Resolution**

- Number of bits N
- Size of digital word



**LSB** → 1 LSB $\triangleq V_{REF} \ / \ 2^N$

**Full Sale Range (FSR)**

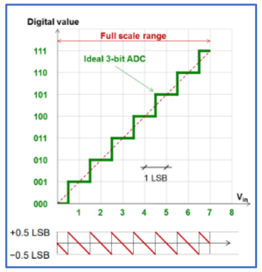
- Range between analog levels of min. and max. digital codes
- $V_{FSR}$ is one LSB less than $V_{REF}$

Quantization error

- Continuous → Discrete
- Introduces an error between -0.5 and +0.5 LSB

Quantization error can be reduced by reducing LSB, e.g. either by increasing number of bits (resolution) or by reducing $V_{REF}$

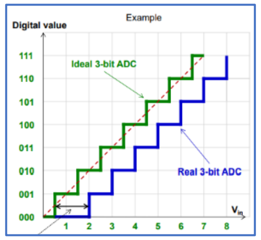
Reducing $V_{REF}$ also reduces full scale range



Offset error (zero-scaler error)

- Deviation of real and ideal N-bit ADC at input point zero
- Ideal: First transition at 0.5 LSB above zero
- Can be corrected using the microcontroller

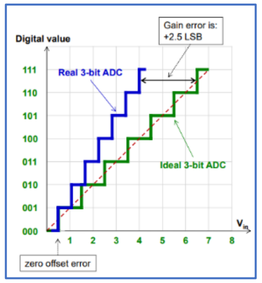
Measuring the offset error:
Zero-scale voltage is applied to analog input and is increased until first transition occurs



Gain error

- Indicates how well the slop of an actual transfer function matches the slope of the ideal transfer function
- Expressed in LSB or as a percent of full-scale range
- Calibration with hardware or software possible

full-scale error = offset error + gain error



## ADC / DAC

**DAC – Digital to Analog Converter**

- Converts N-bit digital input to analog voltage level
- Music from your MP3 player is read and converted back to sound
    - A series of different values in the digital domain leads to a series of steps in the analog domain.
    - «Play-back» time depends on time between conversions

**Output signal $V_{out}$**

- Analog output
    - Unipolar (only positive)
    - Bipolar (positive or negative)
- Conversion yields approximation of digital signal

**Reference voltage $V_{REF}$**

- Accurate reference voltage
- Needed to relate digital value to a voltage

ADC Example 1

Ein externes analoges Signal soll digital abgetastet werden.

Bei einer maximalen Referenzspannung von $V_{Ref} = 4.5\text{V}$ soll eine Abtast-Auflösung von mindestens $5mV$ erreicht werden.

Wie viele Bits werden mindestens für die Analog-Digital Wandlung benötigt.

$$\frac{4.5V}{0.005V} = 900, \qquad \log_2(900) = 9.8 \rightarrow 10\ Bit$$

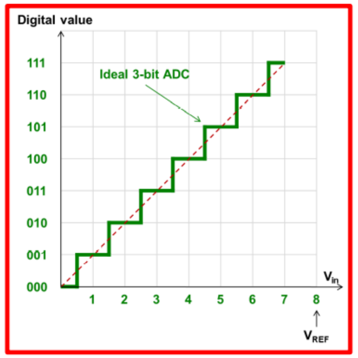
ADC Example 2

Gegeben ist ein 3-bit ADC. Die Referenzspannung $V_{REF} = 8V$ festgelegt.

In welchem Spannungsbereich bewegt sich der Quantisierungsfehler?

$$1\ LSB = \frac{V_{REF}}{2^N} = \frac{8V}{2^3} = 1V$$

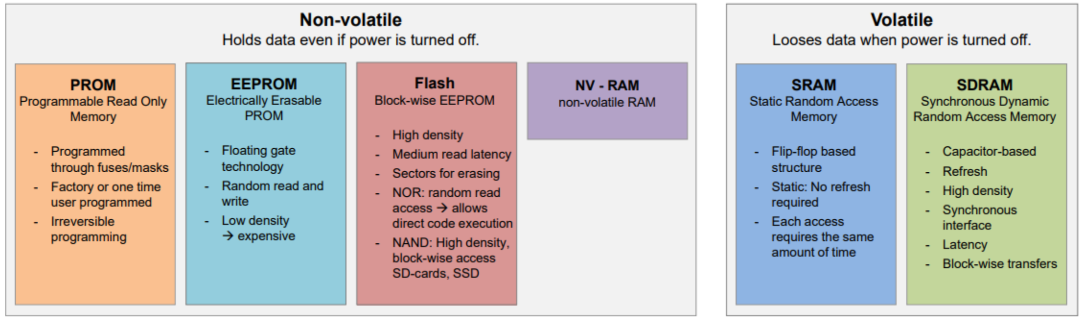
Der Quantisierungsfehler beträgt $\pm\, 0.5 \cdot LSB \rightarrow \pm\, 0.5V$



## Memory

### Übersicht

- PROM – Programmable Read-Only Memory
- EEPROM – Electrical Eraseable PROM
- NOR / NAN Flash
- SRAM – Static Random Access Memory
- DRAM – Dynamic Random Access Memory



**Non-volatile**
Holds data even if power is turned off.

**PROM**
Programmable Read Only Memory
- Programmed through fuses/masks
- Factory or one time user programmed
- Irreversible programming

**EEPROM**
Electrically Erasable PROM
- Floating gate technology
- Random read and write
- Low density → expensive

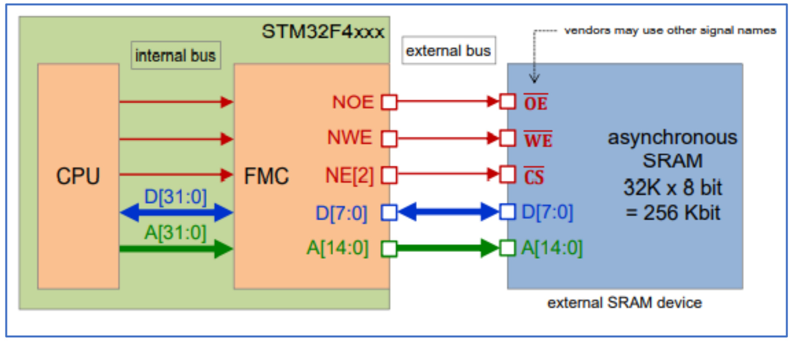
**Flash**
Block-wise EEPROM
- High density
- Medium read latency
- Sectors for erasing
- NOR: random read access → allows direct code execution
- NAND: High density, block-wise access SD-cards, SSD

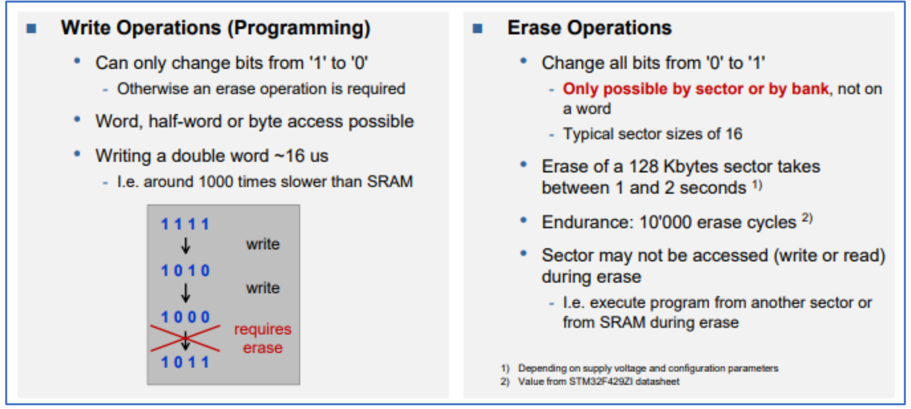
**NV - RAM**
non-volatile RAM

**Volatile**
Looses data when power is turned off.

**SRAM**
Static Random Access Memory
- Flip-flop based structure
- Static: No refresh required
- Each access requires the same amount of time

**SDRAM**
Synchronous Dynamic Random Access Memory
- Capacitor-based
- Refresh
- High density
- Synchronous interface
- Latency
- Block-wise transfers



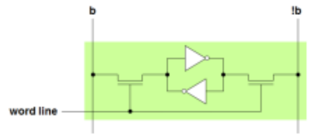
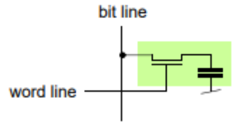


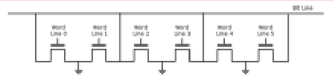
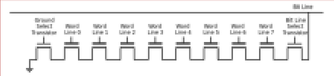
**Write Operations (Programming)**
- Can only change bits from '1' to '0'
  - Otherwise an erase operation is required
- Word, half-word or byte access possible
- Writing a double word ~16 us
  - I.e. around 1000 times slower than SRAM

1 1 1 1 ↓ write
1 0 1 0 ↓ write
1 0 0 0 ↓ requires erase
1 0 1 1

**Erase Operations**
- Change all bits from '0' to '1'
  - **Only possible by sector or by bank**, not on a word
  - Typical sector sizes of 16
- Erase of a 128 Kbytes sector takes between 1 and 2 seconds [1)]
- Endurance: 10'000 erase cycles [2)]
- Sector may not be accessed (write or read) during erase
  - I.e. execute program from another sector or from SRAM during erase

1) Depending on supply voltage and configuration parameters
2) Value from STM32F429ZI datasheet

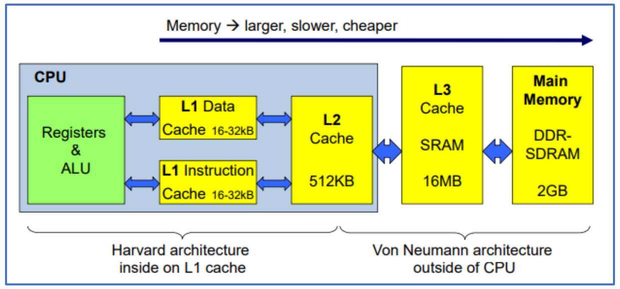
| Static RAM (SRAM) | Synchronous Dynamic RAM (SDRAM) |
|---|---|
| Flip-flop/latch → 4 Transistors / 2 resistors  | Transistor and capacitor  |
| Large cell<br>• Low density, high cost<br>• Up to 64 Mb per device | Small cell<br>• High density, low cost<br>• Up to 4 Gb per device |
| Almost no static power consumption<br>• Static i.e. no accesses taking place | Leakage currents<br>• Requires periodic refresh |
| Asynchronous interface (no clock)<br>• Simple connection to bus | Synchronous interface (clocked)<br>• Requires dedicated SDRAM Controller |
| All accesses take roughly the same time<br>• ~5ns per access → 200 MHz<br>• Suitable for distributed accesses | Long latency for first access of a block<br>• Fast access for blocks of data (bursts)<br>• Large overhead for single byte |

| | NOR Flash | NAND Flash |
|---|---|---|
| Topology |  |  |
| Applications | • Execute code directly from memory<br>• Persistent device configurations (replacement of EEPROM) | • File-based IO, disks<br>• Large amounts of sequential data (images, SD cards, SSD)<br>• Load programs into RAM before executing |
| Density | • Medium Up to 2 GBit = 256 MByte | • High Up to 1 Tbit |
| Interface | • Read same as asynchronous SRAM<br>• Types with serial interface available | • Special NAND flash interface<br>• Error correction for defective blocks |
| Access | • Random access<br>read ~0.12 µs<br>• Writing individual bytes possible<br>• Slow writes<br>~180 µs / 32 Byte | • Slow random access<br>read: 1. Byte 25 µs, then 0.03 µs each<br>• Writing of individual bytes difficult<br>• Fast block write<br>~300 µs / 2'112 Bytes |

## Cache

### Definition

- Computer memory with short access time
- Storage of frequently / recently used instructions / data

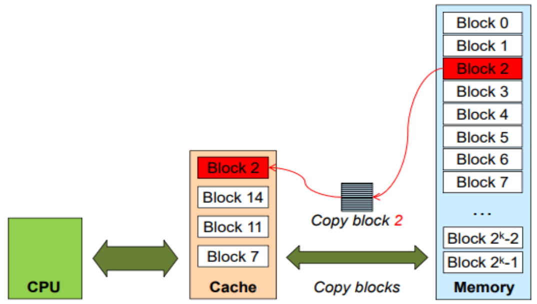

Memory → larger, slower, cheaper
CPU
Registers & ALU
L1 Data Cache 16-32kB
L1 Instruction Cache 16-32kB
L2 Cache 512KB
L3 Cache SRAM 16MB
Main Memory DDR-SDRAM 2GB
Harvard architecture inside on L1 cache
Von Neumann architecture outside of CPU

### Principle of locality

| | |
|---|---|
| Spatial locality | likely close to next accessed location |
| Temporal locality | likely being accessed again in near future |

### Memory blocks

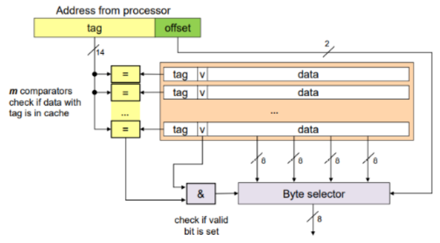
- Address range is partitioned into memory blocks
- Cache is guessing which blocks the CPU will need next
- Selected blocks *copied to faster cache memory*

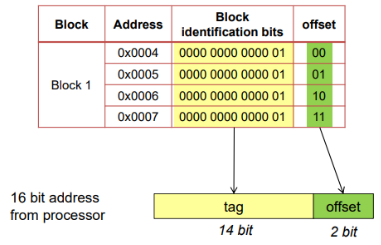


| | |
|---|---|
| Cache hit | Requested block is in cache |
| Cache miss | Requested block is not in cache |

### Organization - Fully associative

- Tag contains complete block identification
- Any cache line can load any block

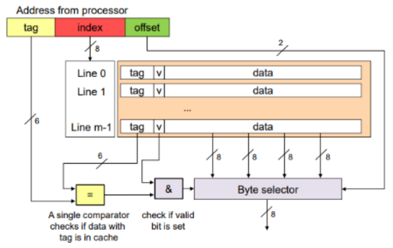

Address from processor
tag
offset
14
2
m comparators check if data with tag is in cache
tag v data
&
Byte selector
check if valid bit is set
6
8

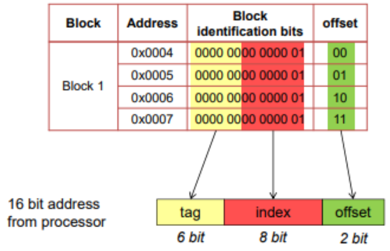


| Block | Address | Block identification bits | offset |
|---|---|---|---|
| Block 1 | 0x0004 | 0000 0000 0000 01 | 00 |
| | 0x0005 | 0000 0000 0000 01 | 01 |
| | 0x0006 | 0000 0000 0000 01 | 10 |
| | 0x0007 | 0000 0000 0000 01 | 11 |

### Organization - Direct mapped

- Block identification split into tag and index
- Each block is mapped to exactly one cache line
- Multiple memory blocks mapped to the same line

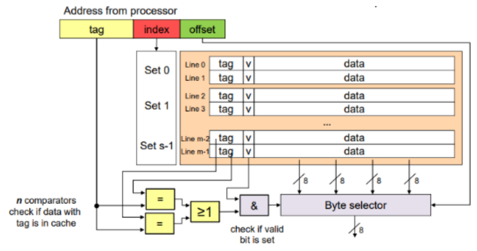


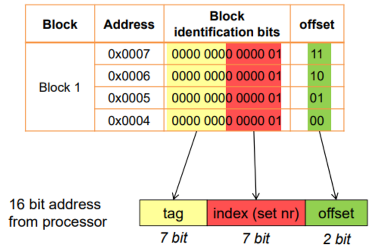


| Block | Address | Block identification bits | offset |
|---|---|---|---|
| Block 1 | 0x0004 | 0000 0000 0000 01 | 00 |
| | 0x0005 | 0000 0000 0000 01 | 01 |
| | 0x0006 | 0000 0000 0000 01 | 10 |
| | 0x0007 | 0000 0000 0000 01 | 11 |

### Organization - N-way set associative

- Partition into sets





| Block | Address | Block identification bits | offset |
|---|---|---|---|
| Block 1 | 0x0007 | 0000 0000 0000 01 | 11 |
| | 0x0006 | 0000 0000 0000 01 | 10 |
| | 0x0005 | 0000 0000 0000 01 | 01 |
| | 0x0004 | 0000 0000 0000 01 | 00 |

# Cache

**Cache miss**

| | |
|---|---|
| Cold miss | first access to a block |
| Capacity miss | Working set larger than cache |
| Conflict miss | Multiple data objects map to same slot |

**Performance**

| | |
|---|---|
| Hit rate | $hits\ /\ accesses$ |
| Miss rate | $misses\ /\ accesses = 1 - hit\ rate$ |
| Hit time | Time to deliver a block from cache to processor |
| Miss penalty | Additional time to fetch from memory (miss) |

**Replacement Strategies**

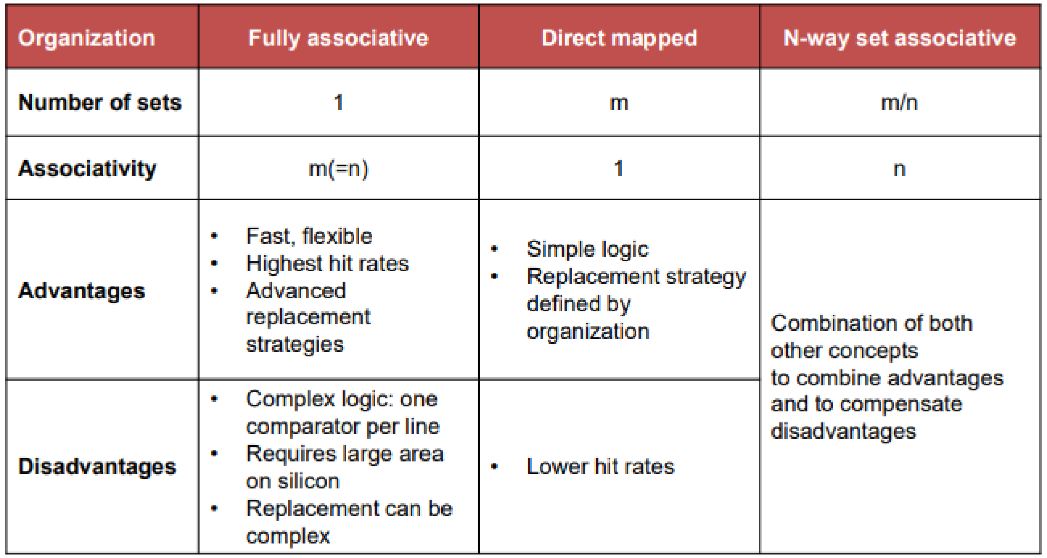
| | |
|---|---|
| LRU | Least recently used |
| LFU | Least frequently used |
| FIFO | First In-First-Out → oldest |
| Random | randomly chosen |

**Write Strategies** (Hit)

| | |
|---|---|
| Write through | immediately write to memory |
| Write back | write on before replacement (valid bit needed) |

**Write Strategies** (Miss)

| | |
|---|---|
| Write-allocate | load line into cache and update line in cache |
| No-write-allocate | Write immediately to memory |

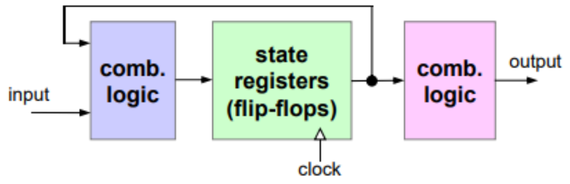


| Organization | Fully associative | Direct mapped | N-way set associative |
|---|---|---|---|
| **Number of sets** | 1 | m | m/n |
| **Associativity** | m(=n) | 1 | n |
| **Advantages** | • Fast, flexible<br>• Highest hit rates<br>• Advanced replacement strategies | • Simple logic<br>• Replacement strategy defined by organization | Combination of both other concepts to combine advantages and to compensate disadvantages |
| **Disadvantages** | • Complex logic: one comparator per line<br>• Requires large area on silicon<br>• Replacement can be complex | • Lower hit rates | |

## State Machines

### FSM in Hardware

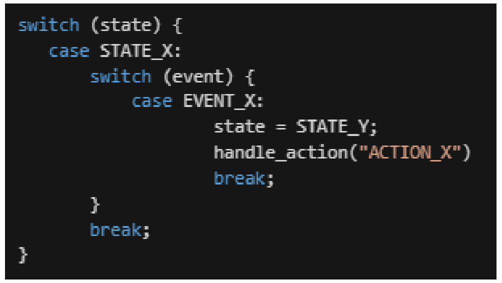
- Flip-flops store internal state
- Clock-driven
  - Inputs are evaluated at each clock edge
  - State can only change on a clock



### FSM in Software

- Responds to external events
  - Event driven
  - Only evaluate the FSM if an input changes
- Internal state
  - Memory of what happened before
- Actions
  - Influence the outside world
- Each event may or may not
  - change the internal state
  - trigger actions

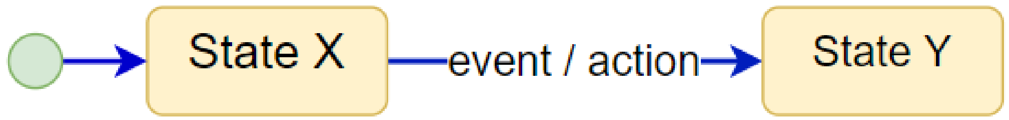
### Implementation



```
switch (state) {
   case STATE_X:
       switch (event) {
           case EVENT_X:
                   state = STATE_Y;
                   handle_action("ACTION_X")
                   break;
       }
       break;
}
```

### Modeling State Machines

| | |
|---|---|
| State | Internal state of the system in which it is awaiting the next event |
| Transition | Reaction to an event: May change state and/or trigger an action |
| Event | Asynchronous input that may cause a transition |
| Action | Output associated with transition |



### Rules for UML

- Every state-diagram must have an *initial state*
- Each state must be *reachable* through a transition
- State diagram must be *deterministic*

### Semantics

- Only reacts to events from the outside
- Always has a defined state
- Reaction to an event depends on the current state
- Once started a transition cannot be interrupted

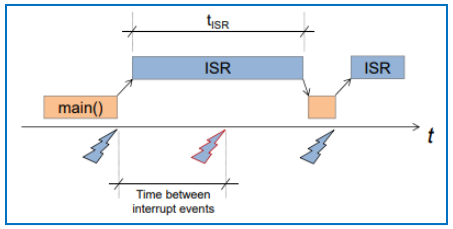
### Events queues

- Collect events generated by different objects
- FSM processes one event after another

# Interrupt Performance

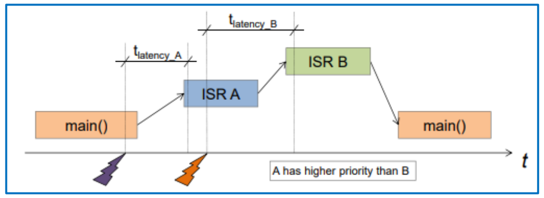
## Interrupt Performance

- $f_{INT}$ Interrupt frequency How often does an interrupt occur
- $t_{ISR}$ Interrupt service time Required time to process an interrupt
  - $t_{ISR} >$ time between two interrupt events → Some interrupts will be lost
- $Imp = f_{INT} \cdot t_{ISR}$ Percentage of CPU time used to service interrupts



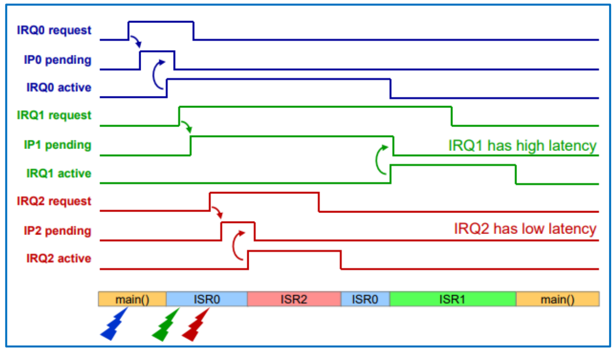
## Interrupt Latency

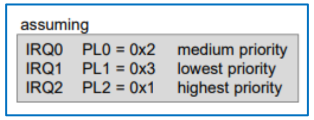
- $t_{latency}$ Time between interrupt event and start of servicing by ISR
- Influenced by HW Instructions have different execution times
- Influenced by SW
  - Saving additional registers on stack…
  - Interrupts with higher priority



- $f_{INT}$ too high → Too many interrupts
  - No CPU cycles left for data processing

## Example – Interrupt Priorities




assuming

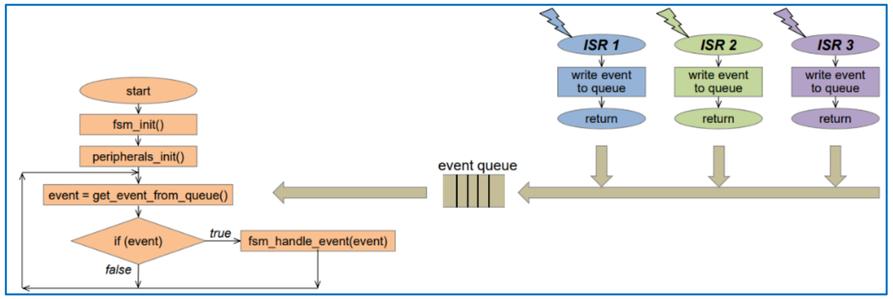
| | | |
|---|---|---|
| IRQ0 | PL0 = 0x2 | medium priority |
| IRQ1 | PL1 = 0x3 | lowest priority |
| IRQ2 | PL2 = 0x1 | highest priority |

## Fast response times – low latency

- Fast CPU High clock rate / low number of cycles per instruction
- Extremely short polling loops can be fast
- Pre-emption with appropriate priorities

## Managing Latency

- High prio interrupts may cause high latencies for low prio interrupts
- Remedy: Move "waiting loop" to main program
- Move non-time-critical work from ISRs to main loop



## Polling

| | |
|---|---|
| Advantages | Simple and straightforward, Implicit synchronization, Deterministic, No additional interrupt logic required |
| Disadvantages | Busy wait → wastes CPU time, Reduced throughput, Long reaction times |