

## Microcontroller Basics

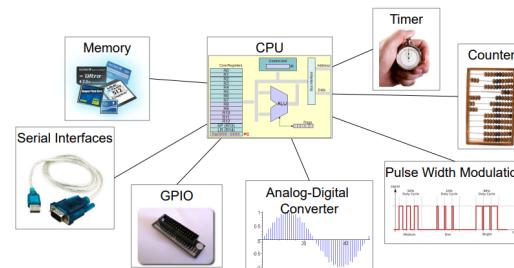
### Microcontroller Architecture

#### Embedded Systems

- low cost (usb sticks, consumer electronics)
- low power (sensor networks, mobile devices)
- small size (smart cards, wearables)
- real time (anti-lock brakes, process control)
- reliability (medical devices, automotive)
- extreme environment (space, automotive)

#### Single Chip Solution

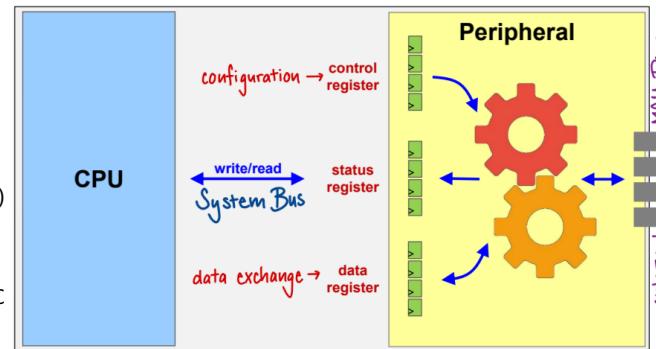
⇒ CPU with integrated memory and peripherals



### Peripherals and Registers

#### Peripherals

- configurable hardware blocks of a microcontroller
- accepts specific task from CPU, executes task and returns result (status, e.g. task completion, error)
- often interfaces to outside world many (not all) interact with external MCU pins
- examples: GPIO, UART, SPI, ADC



#### Registers

- Registers are arrays of flip-flops (storage elements with two states, i.e. 0 or 1)
- Each flip-flop stores one bit of information
- CPU writes to and reads from registers

#### CPU read/write to peripheral registers

How does the CPU write to and read from peripheral registers?

- CPU reads/writes to peripheral registers
  - CPU uses memory-mapped I/O to access peripheral registers
  - CPU uses load/store instructions to access peripheral registers
- ⇒ System Buses

#### Control and Status Registers

Peripherals interact with the CPU through registers:

- **Control Registers:** CPU configures peripherals
  - CPU writes to register bit
  - Slave hardware uses the output of this bit
  - Usually read/write
- **Status Registers:** CPU monitors peripheral state
  - Slave writes status into register bit
  - CPU reads register bit
  - Usually read-only
- Both control & status bits can be in same register.
- **Data Registers** enable CPU to exchange data with the peripheral

#### CPU access to individual Peripheral Registers

- ARM & STM map the peripheral registers into the memory address range
- Reference Manual shows the defined addresses

#### Example SPI (Serial Peripheral Interface)

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x00	SPI_CR1																BIDIMODE	BIDIOE	CRCEN	CRCNEXT	DFF	RXONLY	SSM	SSI	SPE	BR [2:0]	MSTR	CPOL	CPHA					
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
0x04	SPI_CR2																TXEIE	RXNEIE	ERRIE	FRF	Reserved	SSOE	CHSIDE	TXE	RXNE									
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
0x08	SPI_SR																FRE	BSY	OVR	MODF	CRCERR	UDR												
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
0x0C	SPI_DR																DR[15:0]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			

Configuration

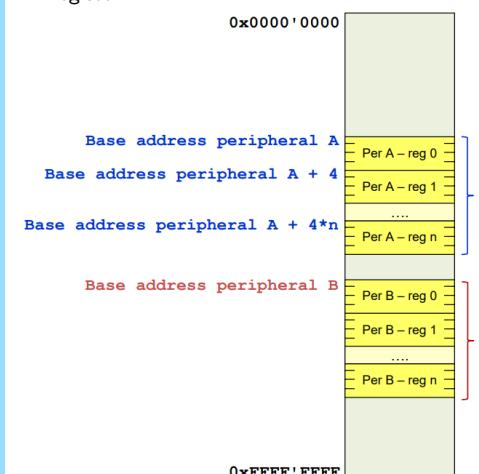
Status

Transmit / Receive

source: STM32F42xxx Reference Manual

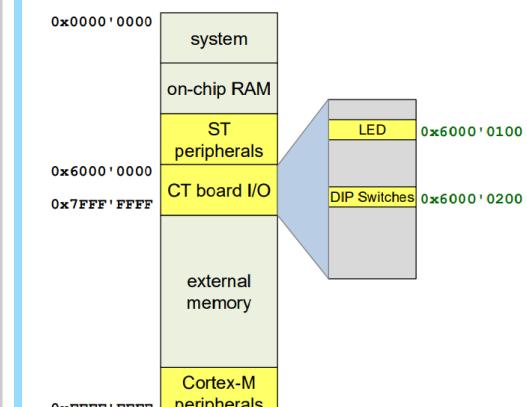
#### Memory mapping of Peripheral Registers

- Each peripheral register has a unique address
- CPU uses the address to access the register
- CPU uses load/store instructions to access the register



#### Memory-Mapped Peripheral Registers

- Control register: controls states of LEDs
- Status register: monitors states of DIP switches



#### Memory-Mapped Registers

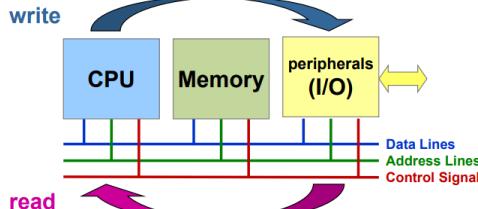
Registers are mapped into the memory address range - each has a specific address:

```

1 // Define memory-mapped register addresses
2 #define ADDR_LED_31_0      0x60000100
3 #define ADDR_DIP_SWITCH_31_0 0x60000200
4
5 // Read from DIP switches and write to LEDs
6 uint32_t value = read_word(ADDR_DIP_SWITCH_31_0);
7 write_word(ADDR_LED_31_0, value);
  
```

## System Bus

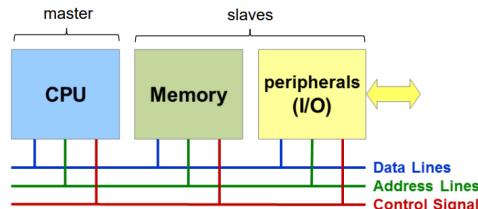
- Interconnects CPU with memory and peripherals, allowing data transfer between components.
- CPU acts as master: initiating and controlling all transfers
- Peripherals and memory act as slaves: responding to requests from the CPU
- System bus is a shared resource



The CPU acts as a master, initiating and controlling all transfers, while peripherals and memory act as slaves, responding to requests.

## Signal Groups

- **Data lines**
  - Bidirectional (read/write)
  - Number of lines → data bus width (8, 16, 32, 64 parallel lines of data)
  - Example: Cortex-M has 32 address lines → 4GB address space  
→ 0x00000000 to 0xFFFFFFFF
- **Address lines**
  - Unidirectional: from Master to slaves
  - Number of lines → size of address space (e.g., 32 lines allow  $2^{32}$  addresses)
- **Control signals**
  - Control read/write direction
  - Provide timing information
  - Chip select, read/write, etc.



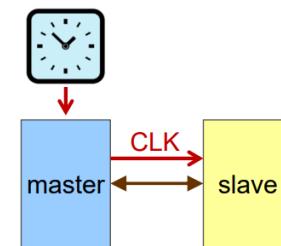
## Bus Specification

- Protocol and operations
- **Signals**
  - Number of Signals
  - Signal descriptions
- **Timing**
  - Frequency
  - Setup and hold times
- **Electrical properties** (not in exam)
  - Drive strength and Load
- **Mechanical requirements** (not in exam)

## Bus Timing Options

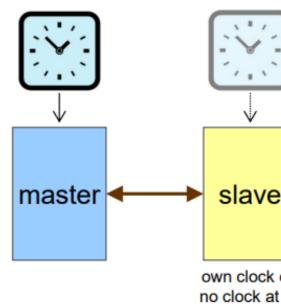
### Synchronous

- Master and slaves use a common clock
  - Often dedicated CLK signal from master to slave, but clock can also be encoded in a data signal
- Clock edges control bus transfer on both sides
- Used by most on-chip buses
- Off-chip: DDR and synchronous RAM



### Asynchronous

- Slaves have no access to clock of the master
  - slave has their own clock or no clock at all
- Control signals carry timing information to allow synchronization
- Widely used for low data-rate off-chip memories
  - parallel flash memories and asynchronous RAM



## But how can a driver be disconnected electrically?

### Multiple devices driving the same data line

What if one device drives a logic 1 (Vcc) and another device drives a logic 0 (Gnd)?  
→ Electrical short circuit!  
→ bus contention ('Streitigkeit')

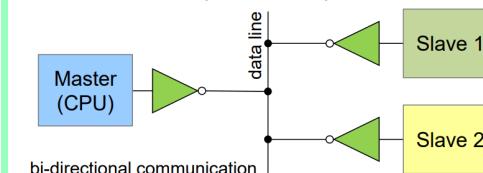


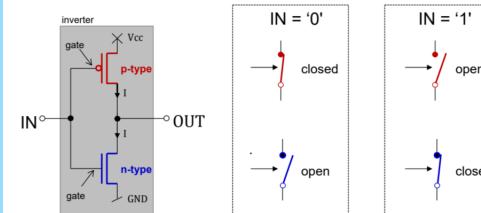
Figure only shows output paths, input paths are not shown.

## Digital Logic Basics

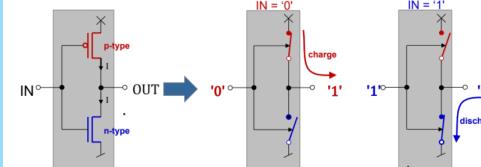
### CMOS Inverter

Complementary switches (transistors)

→ p-type and n-type have opposite open-close behaviour



e.g. Vcc = 3V = '1', Gnd = 0V = '0'  
Vcc is the supply voltage of the circuit/chip

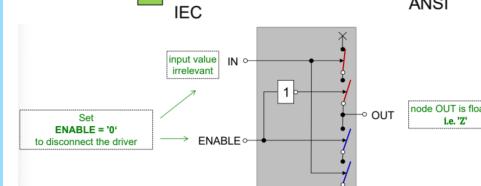


A buffer is built by connecting two inverters in series

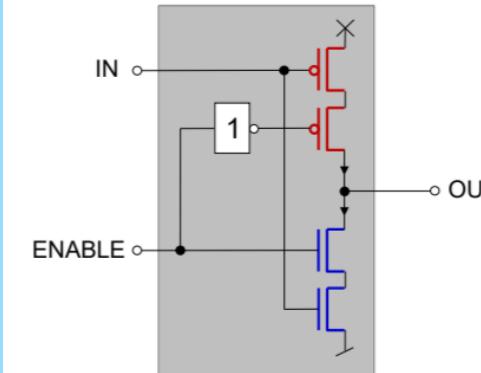
### CMOS Tri-State Inverter

ENABLE → IN → OUT

ENABLE → IN → OUT



### Implementation:

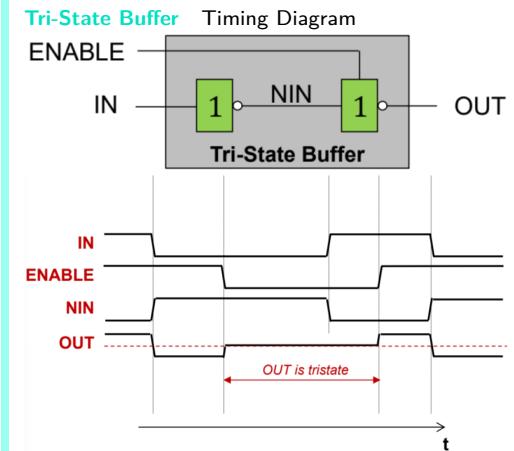


## Tri-State Logic

Multiple devices can drive the same data line thanks to tri-state capability:

- Logic '1': Voltage level (e.g., 3.3V)
  - Logic '0': Ground (0V)
  - Third state 'Z': High impedance (disconnected, floating)
- The CPU defines which device drives the bus:
- **Write**: CPU drives bus, all slave drivers disconnected
  - **Read**: CPU driver disconnected, selected slave drives bus, other slave drivers disconnected

### Tri-State Buffer



When a signal like OUT is in tristate, we often say that it is 'floating'. The term expresses that such a signal can easily be moved by parasitic electrical effects to either one of the reference levels, i.e. '0' or '1'.

## Bus Contention

CPU defines who drives the data bus at which moment in time:

- write CPU drives bus → all slave drivers disconnected
- read CPU releases bus → one slave drives bus (selected through values on address lines, other slave drivers disconnected)

Electrically disconnecting a driver is called **tri-state** or **high-impedance** (Hi-Z) state. (switch)

ENABLE	OUT
'1'	! IN
'0'	'Z'

'Z' = high impedance

## Synchronous Bus

### Synchronous Bus

Example Uses External Bus from ST Microelectronics

- Reason: Internal workings of the system bus are not disclosed by STM
- Signal names, bus protocol and timing based on external synchronous STM32F429xx mode instead
- For details see Chapter 37, Flexible memory controller (FMC) in ST Reference Manual RM0090
- Datasheet STM32F429xx
- Figure 60 Synchronous non-multiplexed NOR/PSRAM read timings
- Figure 61 Synchronous non-multiplexed PSRAM write timings

### Naming Convention

- Letter 'N' prefix in signal name ( $N_{xxx}$ ) means active-low signal
- E.g. NOE means 'NOT OUTPUT ENABLE'  
NOE = '0' → output enabled  
NOE = '1' → output disabled

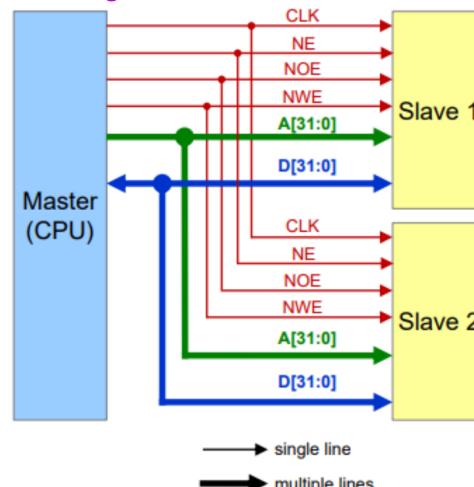
### Synchronous Bus Timing

Key signals for read/write operation:

- CLK: System clock
- A[31:0]: Address lines
- D[31:0]: Data lines
- NWE: Not Write Enable (active low)
- NOE: Not Output Enable (active low)
- NE: Not Enable (active low)

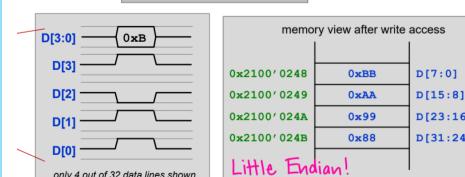
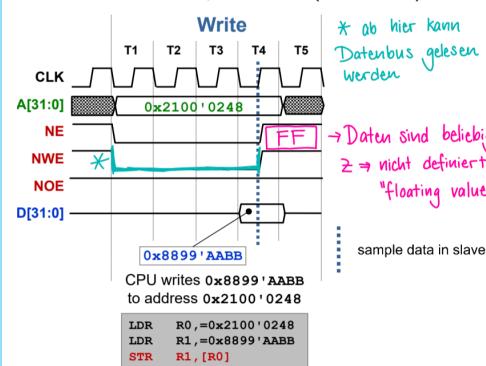
Note that 'N' prefix indicates active-low signals.

### Block Diagram



### Bus Timing Diagram

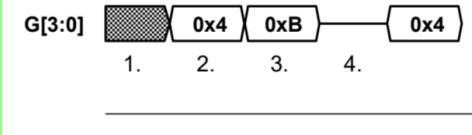
- CLK → clock signal (rising edge)
- NE → Not enable (active low)
- NWE → Not write enable (active low)
- NOE → Not output enable (active low)



WICHTIG: nicht genau mit Flanke schreiben, Daten brauchen eine gewisse Zeit um stabil zu werden (keine genaue Definition, muss einfach 'genug' sein)  
READ dauert länger als WRITE, da die Daten erst stabil werden müssen bevor sie gelesen werden können

### Bus Timing Diagrams

Notation for Groups of Signals



Group G of 4 signals

1. unknown values

The values on each of the 4 signals are either '1' or '0', but unknown.  
2. The bus holds the value  $0 \times B$

i.e.  $G[3] = '0'$ ,  $G[2] = '1'$ ,  $G[1] = '0'$ ,  $G[0] = '0'$

3. The bus holds the value  $0 \times B$

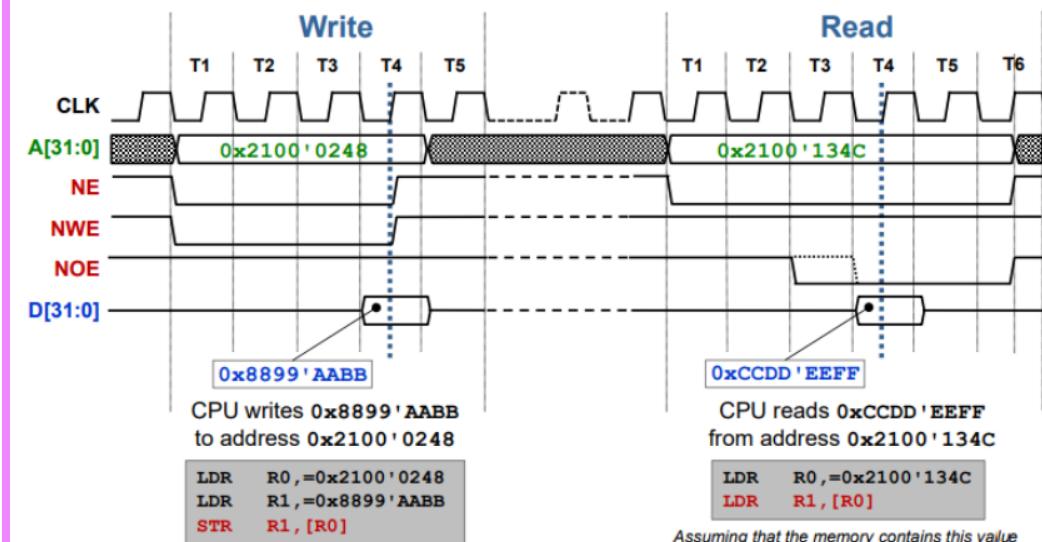
i.e.  $G[3] = '1'$ ,  $G[2] = '0'$ ,  $G[1] = '1'$ ,  $G[0] = '1'$

4. Tri-state

All signals  $G[3 : 0]$  are tri-state (i.e. 'Z' or high-impedance). "No one is driving the bus"

### Timing Diagram

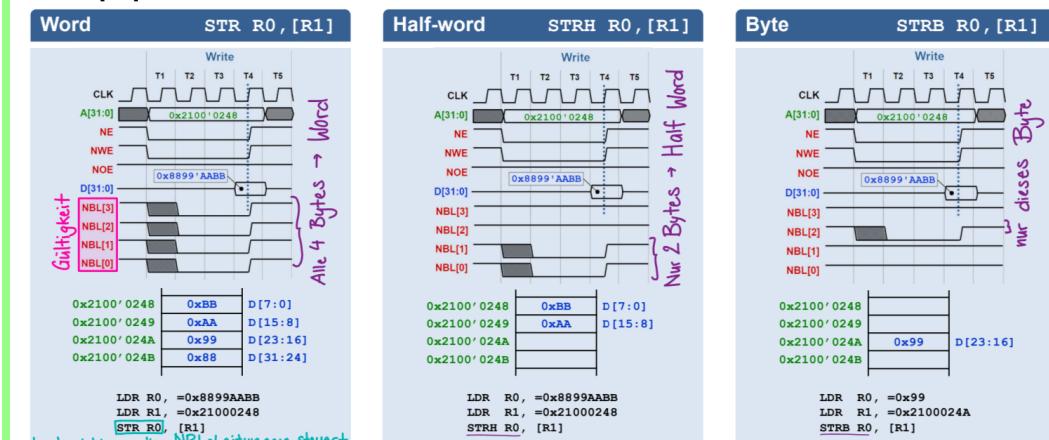
- write  $D[:]$  to  $A[:]$  → NE, NWE = 0
- read  $D[:]$  from  $A[:]$  → NE, NOE = 0



Assuming that the memory contains this value

Bus Access Size is determined by the NBL (0-3) (No Byte Line) signals

- NBL = 1 → Byte used for Read/Write
- NBL = 00
- NBL[0:3] = 0011 → Read Halfword
- NBL[0:3] = 1111 → Read Word



Gültigkeit: damit zeigt die CPU an, welche der 4 Bytes übertragen werden sollen (gültig = 0 (unten))

- Exact Position of falling edge on NBL varies with chip version
- Value on unused data lines are unknown, figures show assumptions

## Bus Access Analysis

### Analyzing Bus Cycle Diagrams

Read access analysis

Timing-Diagramm interpretieren

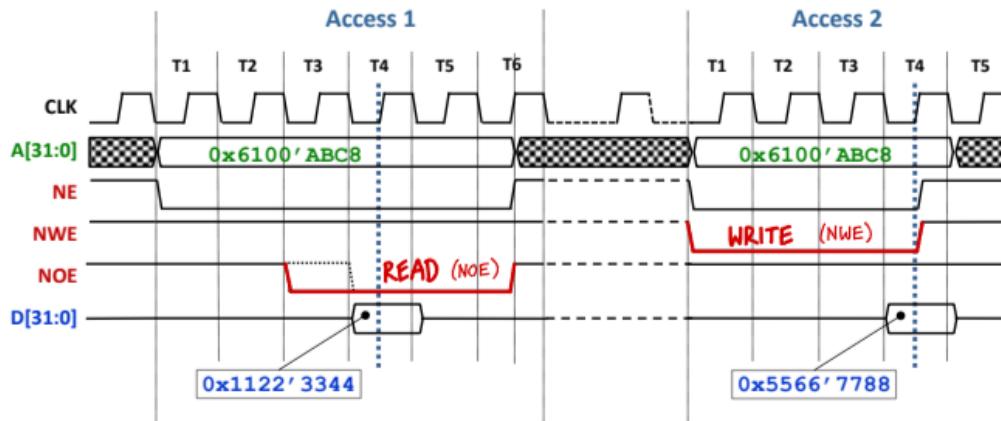
- NWE (Not Write Enable) = '0' → Write-Zugriff
- NOE (Not Output Enable) = '0' → Read-Zugriff
- Adresse steht auf A[31:0] während des Zugriffs
- Daten stehen auf D[31:0] während des Zugriffs

Memory mapping

- Remember that processors like ARM use little-endian ordering
- Lowest memory address holds the least significant byte (LSB)
- Highest memory address in the word holds the most significant byte (MSB)
- Map the bytes according to this ordering in memory

### Bus Cycle Analysis Example

Given the following bus cycle diagram, determine the operation type, address, and data value:



Access 1: T1 T2 T3 T4 T5 T6 with A[31:0] = 0x6100ABC8, NOE active, D[31:0] = 0x11223344

Access 2: T1 T2 T3 T4 T5 with A[31:0] = 0x6100ABC8, NWE active, D[31:0] = 0x55667788

Solution:

Access 1: This is a read operation because NOE is active.

- Address: 0x6100ABC8
- Data read: 0x11223344

Access 2: This is a write operation because NWE is active.

- Address: 0x6100ABC8
- Data written: 0x55667788

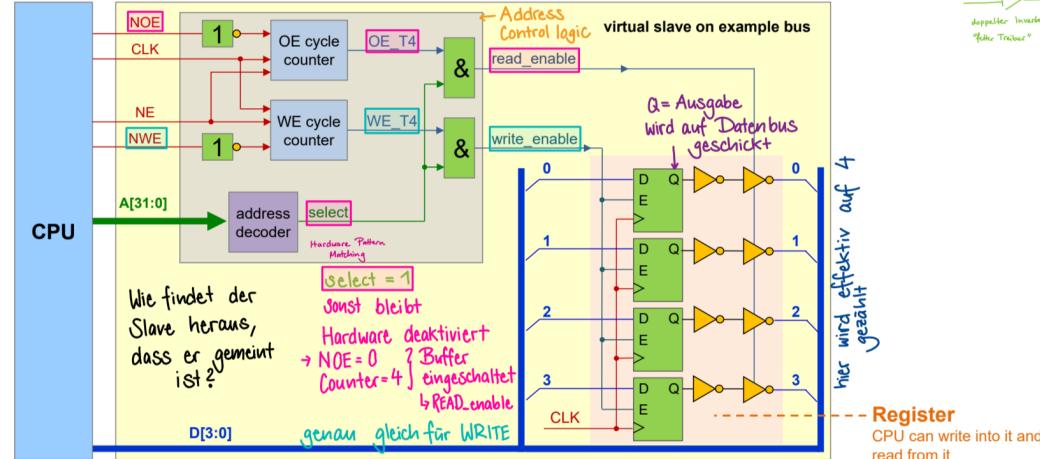
Memory contents (little-endian):

Address	Content before Access 2
0x6100ABC8	0x44 (LSB)
0x6100ABC9	0x33
0x6100ABCA	0x22
0x6100ABCB	0x11 (MSB)

Address	Content after Access 2
0x6100ABC8	0x88 (LSB)
0x6100ABC9	0x77
0x6100ABCA	0x66
0x6100ABCB	0x55 (MSB)

## Control and Status Registers

### Hardware Slave (Peripheral)



### Control Bits

- Allow CPU to configure Slaves
- CPU writes to register bit to configure Slave
- Slave uses output of register bit to configure itself
- Example: SPI Slave Select (SS) bit
- Usually read/write access to control bits

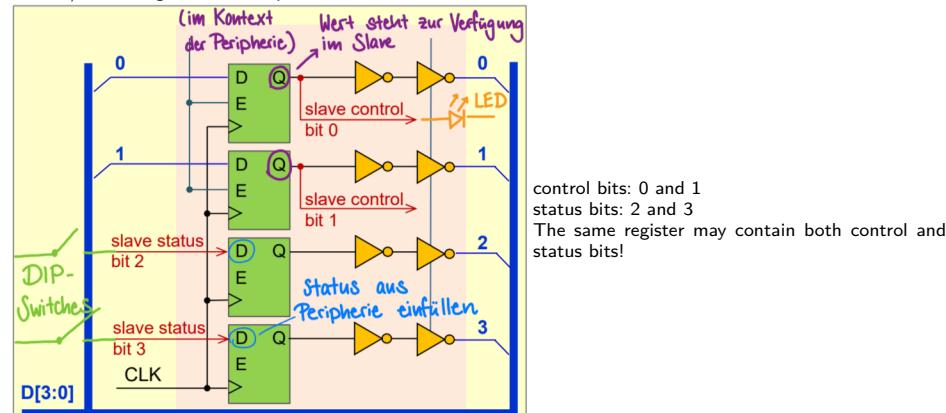
### Status Bits

- Allow CPU to monitor Slaves
- CPU reads register bit to monitor Slave
- Slave uses input of register bit to monitor itself (Slave writes to register bit)
- Example: SPI Busy bit
- Usually read-only access to status bits

### Example STM32 Power Control/Status Register PWR\_CSR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	VOS RDY		Reserved	BRE	EWUP		Reserved	BRR	PVDO	SBF	WUF				
				rw	rw			r	r	r	r				
<b>Control Bits</b>								<b>Status Bits</b>							
BRE: Backup regulator enable EWUP: Enable WKUP pin								BRR: Backup regulator ready PVDO: PVD output SBF: Standby flag WUF: Wakeup flag							

Control/Status register on example bus:



### Control and Status Registers on CT Board

Chip-internal and external registers (details on memory map in STM32 Reference Manual)

0x0000'0000	system (boot)
0x1FFF'FFFF	on-chip RAM
0x2000'0000	ST peripherals
0x3FFF'FFFF	CT board I/O
0x4000'0000	external memory
0x5FFF'FFFF	ARM Cortex-M
0x6000'0000	NVIC, ...
0x7FFF'FFFF	
0x8000'0000	
0x9FFF'FFFF	
0xA000'0000	
0xBFFF'FFFF	
0xC000'0000	
0xDFFF'FFFF	
0xE000'0000	
0xFFFF'FFFF	Cortex-M peripherals

1

1 ST peripherals  
e.g. Timers, ADC, UART, SPI, ...

2

2 CT board I/O  
LEDs, dip switches, LCD, ...

3

3 ARM Cortex-M  
NVIC, ...

Zwiebelbild:

ARM Cortex-M peripherals  
Design  
CPU, NVIC, power control  
legt bereits NVIC Adressen fest  
Bereich system (boot) bereits festgelegt → ARM Design

STM : ST-Peripherals  
→ Timers, SPI, I2C, ADC, UART  
ab Adress-Space 0x4000'0000

INes : CT board, I/O  
→ LEDs, dip switches, LCD  
Anfangsadressen 0x6000'0000 - 0x7FFF FFFF

## Address Decoding

### Address Decoding

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

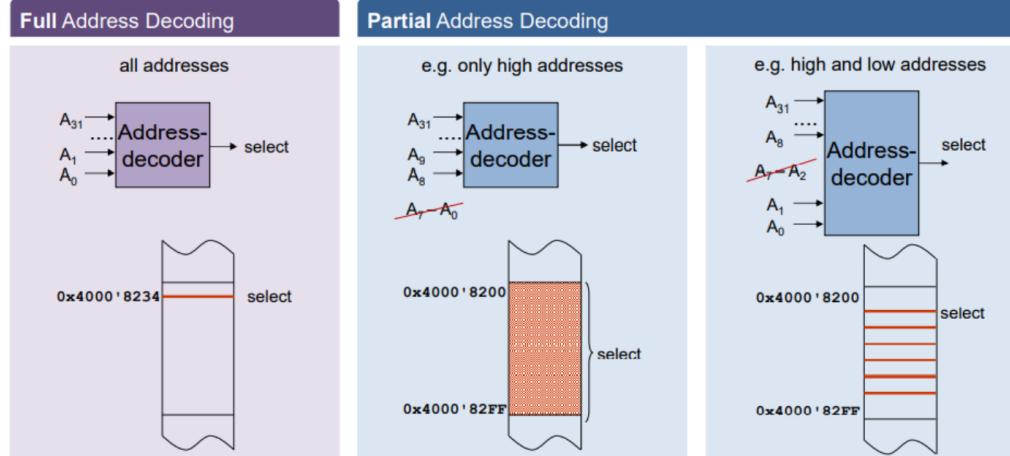
- CPU uses address lines to select a peripheral
- Each peripheral has a unique address range
- Address decoding logic generates a chip select signal for each peripheral

### 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
- example: LEDs and DIP switches on CT board

### Partial Address Decoding

- Only a subset of address lines are decoded
- A control register can be accessed at multiple locations
- 1:n mapping: Multiple addresses map to the same hardware register
- Motivation: Simpler and possible Aliasing (Map a hardware register to several addresses)



### Simple Address Decoder

Basic address decoder with 3 address lines that selects when address is 0x5:

```
// In hardware description language (e.g., Verilog):
2 assign select = (A[2] & !A[1] & A[0]); // Decodes address 0x5 (101 binary)
```

### Address Range Calculation

For partial address decoding, if only higher-order address bits  $A[n : m]$  are decoded (where  $n > m$ ), then:

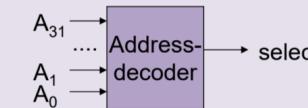
- Each decoded address represents a range of  $2^m$  addresses
- The size of this address range is  $2^m$  bytes
- The start address has all lower bits set to 0
- The end address has all lower bits set to 1

For example, if only A[31 : 8] are decoded, each decoded address represents a 256-byte range ( $2^8 = 256$ ).

How does a Slave know that it is being addressed?  
⇒ Address decoding logic in the Slave (each on its own)

### Full Address Decoding

- all addresses from A31 to A0



- select is active for exactly one address

- E.g. at 0x4000'8234

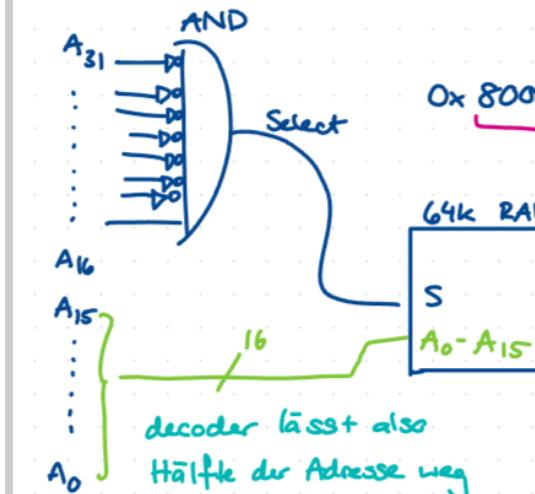
### Partial Address Decoding

- only addresses from A31 to A8



- select is active for any address within a given range (e.g. ignoring some lower address lines)

- E.g. from 0x4000'8200 to 0x4000'82FF  
→ 0x4000'82xx



Antang der Adresse  
muss einfach 8000 sein,  
Rest ist "egal"

→ Partial Decoding

## Address Decoding KR and Examples

### SEP Handout page 1!!! FMC Decoding

#### Address Decoding

Anzahl adressierbarer Bytes berechnen

Bus mit n Adressleitungen → es können  $2^n$  Bytes adressiert werden

Partial Address Decoding

Mit n Adressleitungen, x davon werden dekodiert → Register kann unter  $2^{(n-x)}$  Adressen angesprochen werden

Spezifische Leitungen dekodieren

- $A[i] \rightarrow$  Address-Bit = 1
- $\neg A[i] \rightarrow$  Address-Bit = 0
- nichts → egal (0 oder 1, schreibe X)

Vorgehen

1. Schreibe Adresse in binär auf
2. Setze für jede Adressleitung Wert nach Legende ein
3. Für alle Stellen mit X können alle möglichen Kombinationen eingesetzt werden
4. Diese ergeben alle Adressen, unter denen das Register angesprochen werden kann

Umgekehrt (spezifische Adresse gegeben)

1. Schreibe Adresse in binär
2. Schreibe für alle Leitungen  $A[i]$  bzw.  $\neg A[i]$  (beachte Endianness)
3. select =  $A[n] \& A[n-1] \& \dots \& A[0]$  (mit ! falls nötig)

#### Address Decoding mit 6 Adressleitungen

System Bus mit  $A[5:0]$ , decode nur obere 4 Leitungen: select =  $A[5] \& A[4] \& \neg A[3] \& \neg A[2]$

Lösung:

- Binär: 1100XXb
- Alle möglichen Kombinationen für XX: 00, 01, 10, 11
- Adressen: 0x30, 0x31, 0x32, 0x33

Für Adresse 0x28 (101000b): select =  $A[5] \& \neg A[4] \& A[3] \& \neg A[2] \& \neg A[1] \& \neg A[0]$

#### Address Decoding Example

Given a 6-bit address bus  $A[5:0]$ , determine the addresses that will select a device with the following address decoding logic:

$$\text{select} = A[5] \& A[4] \& \neg A[3] \& \neg A[2]$$

Solution:

The decoding logic fixes 4 address bits:  $A[5:2] = 1100$

The bits  $A[1:0]$  are "don't care" bits (not included in the logic), giving 4 possible addresses:

- $A[5:0] = 110000 = 0x30$
- $A[5:0] = 110001 = 0x31$
- $A[5:0] = 110010 = 0x32$
- $A[5:0] = 110011 = 0x33$

Therefore, the device can be addressed at any of these four addresses: 0x30, 0x31, 0x32, or 0x33.

## Wait States for Slow Peripherals

### Wait States

Wait states are extra clock cycles inserted to allow slow peripherals to respond.

- Without wait states, the slowest slave would determine bus cycle time
- Wait states can be:
  - Programmed at a bus interface unit depending on the address
  - Requested by slaves through a "readySignal" (for long or variable access times)

### Slow Slaves

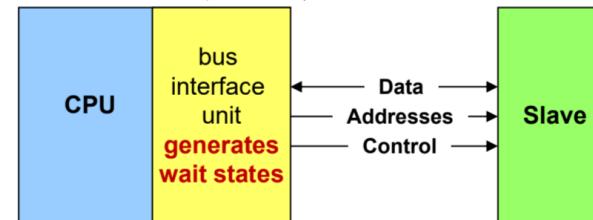
Problem: Individual Slave Access times

- If slowest slave defines bus cycle time → reduced bus performance
- How can we get an individual bus cycle time for each slave?

### Solutions for slow slaves

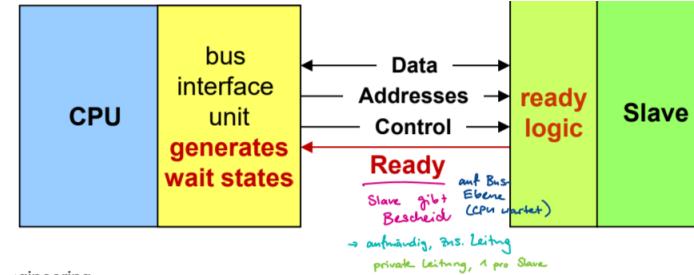
- **Individual Wait States** can be programmed at a bus interface unit

Insert wait states to slow down the CPU to match the speed of the slowest peripheral (depending on the address of an access/bus cycle)



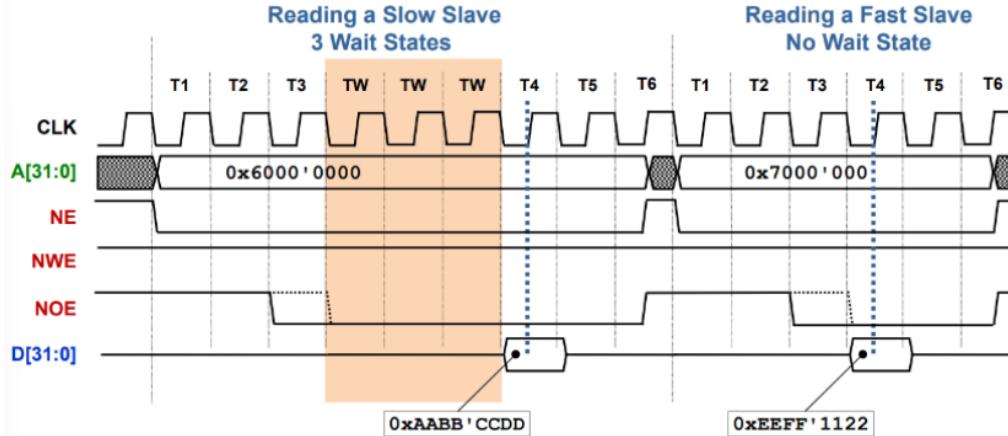
- **Bus Mastering** Slave tells bus interface unit when it is ready

Allow a peripheral to take control of the bus and perform its own accesses (e.g. DMA)  
Well suited for slaves with long or variable access times

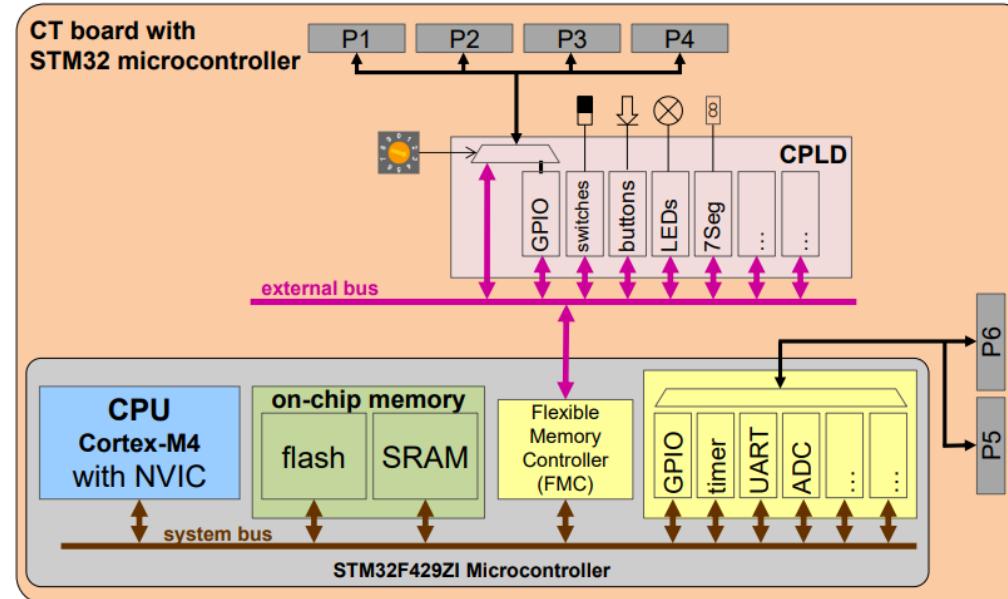


## Individual Wait States

Wait states are inserted to slow down the CPU to match the speed of the slowest peripheral (depending on the address of an access)



CT Board with STM32 Microcontroller and Buses



Real-world Systems are partitioned into multiple buses.

## STM32 Microcontroller

with CPU, on-chip memory, and peripherals interconnected through the system bus(es)

- On-chip system bus: 32 data lines, 32 address lines and control signals
- Off-chip external bus: 16 data lines, 24 address lines and control signals

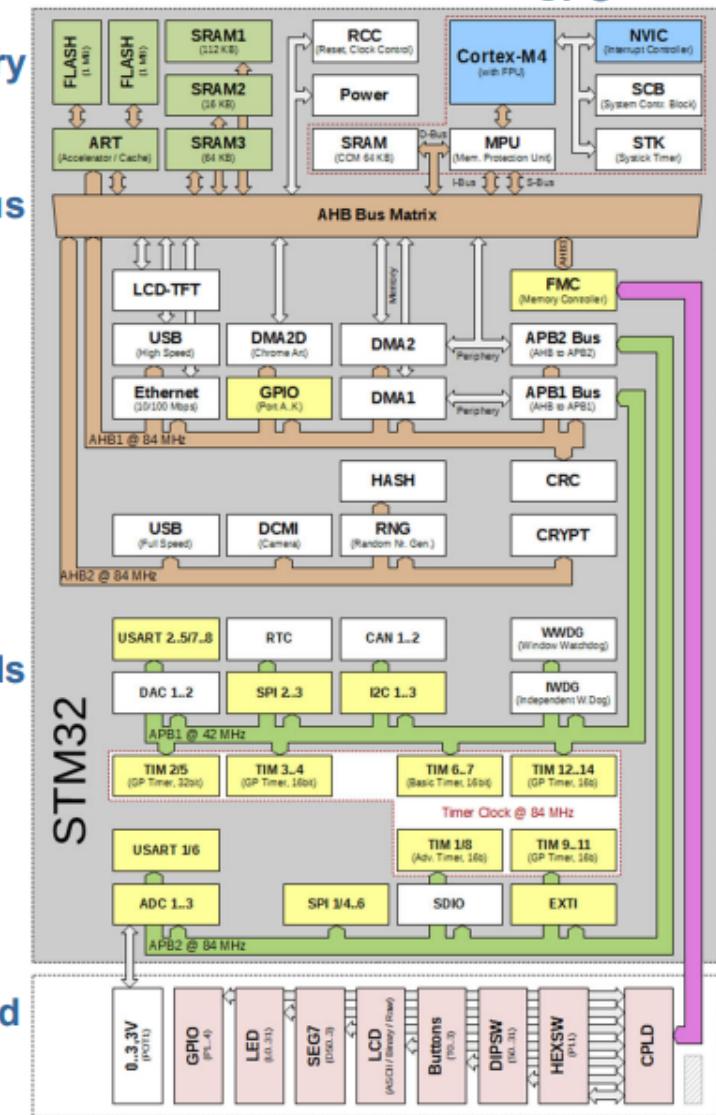
## Memory

## System Bus

## Peripherals

## STM32

## CT-Board I/O



A distributed system with parallel (simultaneous) processing of data in many peripherals. All under the supervision of the CPU.

Note: ARM calls their system buses AHB (ARM High-performance Bus) and APB (ARM Peripheral Bus). On complex chips, it is state-of-the-art to partition the system bus into multiple interconnected buses.

# Programming Memory-Mapped Peripherals

## Accessing Control Registers in C

### Accessing Control Registers in C

#### Hardware View

- Signals
- Timing
- Address decoding
- Wait states
- Control and status registers

#### Software View

- Accessing control and status registers in C

### Problem

Compiler may remove statements that have no effect from the compiler's point of view

```
uint32_t ui;

void ex_func(void)
{
    ui = 0xAAAAAAA;
    ui = 0xBBBBBBBB;
    while (ui == 0) {
        ...
    }
}
```

Optimizing compiler will remove these statements as they seem to have no effect

kann gefahrlos  
gestrichen werden  
(Kontrollregister, wird direkt verworfen)

- Accesses to control registers (read/write) are memory accesses
- If writing has a side effect on the hardware and/or if reading may result in a different value than was set before in the code, the program will not behave as intended/expected

### Solution

- Use the volatile keyword/qualifier in C to prevent the compiler from removing statements that have no effect from the compiler's point of view  
→ prohibit compiler optimizations on the variable
- The compiler will not optimize away accesses to a variable declared as volatile
- The compiler will not reorder accesses to a variable declared as volatile

```
volatile uint32_t ui;

void ex_func(void)
{
    ui = 0xAAAAAAA;
    ui = 0xBBBBBBBB;
    while (ui == 0) {
        ...
    }
}
```

statements will not be removed by compiler

- Tell compiler that the variable may change at any time, outside the control of the compiler (e.g. by hardware or interrupt handler)
- The compiler cannot make any assumptions about the value of the variable  
needs to execute all read/write accesses as programmed  
prevents compiler optimizations

### Accessing Control Registers in C

Key considerations:

- Compiler optimization may remove statements that appear to have no effect
- Register accesses have side effects that the compiler doesn't understand
- Use volatile qualifier to prevent compiler optimization

### Using volatile for register access

```
1 // Without volatile, compiler might remove these statements
2 uint32_t ui;
3 void ex_func(void) {
4     ui = 0xAAAAAAA; // Appears to have no effect
5     ui = 0xBBBBBBBB; // Appears to have no effect
6     while (ui == 0) {
7         ...
8     }
9 }

10 // With volatile, all accesses are preserved
11 volatile uint32_t ui;
12 void ex_func(void) {
13     ui = 0xAAAAAAA; // Will be executed
14     ui = 0xBBBBBBBB; // Will be executed
15     while (ui == 0) {
16         ...
17     }
18 }
```

### Accessing Registers Through Pointers

Step 1: Create pointer to register –

Define a pointer to a volatile memory-mapped register.

Step 2: Assign address –

Cast the register's physical address to a pointer type.

Step 3: Access register –

Use pointer dereference to read or write.

```
1 // Create a pointer to volatile uint32_t
2 volatile uint32_t *p_reg;
3
4 // Set LEDs
5 p_reg = (volatile uint32_t *) (0x60000100);
6 *p_reg = 0xAA55AA55; // Write pattern to LEDs
7
8 // Wait for DIP switches to be non-zero
9 p_reg = (volatile uint32_t *) (0x60000200);
10 while (*p_reg == 0) {
11     ...
12 } // Wait until any switch is pressed
```

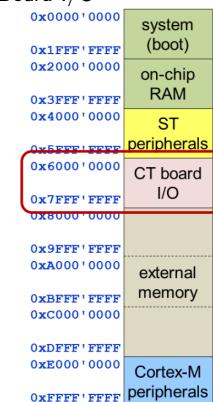
Access through Pointers e.g. writing to and reading from CT Board I/O

```
// a pointer called p_reg pointing to
// a volatile uint32_t
volatile uint32_t *p_reg;

cast 'unsigned integer' to
'pointer to volatile uint32_t'

// set LEDs
p_reg = (volatile uint32_t *) (0x60000100);
*p_reg = 0xAA55AA55;
write 0xAA55'AA55 to LEDs

// wait for dip_switches to be non-zero
p_reg = (volatile uint32_t *) (0x60000200);
while ( *p_reg == 0 ) {
}
read dip-switches
```



## Using Preprocessor Macros for Register Access

### Step 1: Define register macros

Create macros that encapsulate register addresses with proper casting and dereferencing.

### Step 2: Use macros for access

Access registers through the defined macros.

```
1 // Define register macros
2 #define LED31_0_REG    (*((volatile uint32_t *)(0x60000100)))
3 #define BUTTON_REG     (*((volatile uint32_t *)(0x60000210)))
4
5 // Write to LED register
6 LED31_0_REG = 0xBBCCDDEE;
7
8 // Read button register
9 uint32_t aux_var = BUTTON_REG;
```

### Using Preprocessor Macros → #define

```
1 #define LED31_0_REG (*((volatile uint32_t *) 0x60000100))
2
3 #define BUTTON_REG (*((volatile uint32_t *) 0x60000210))
4
5 // Write LED register to 0xBBCC'DDEE
6 LED31_0_REG = 0xBBCCDDEE;
7 // Read Button register to aux_var
8 aux_var = BUTTON_REG;
```

(\*((volatile uint32\_t \*) 0x60000100))  
→ **dereference** the pointer to the register address  
→ **cast** the address to a pointer to a 32-bit register

## Memory-Mapped I/O Access in C

### Wichtige Infos zu Memory-Mapped Register Access in C

#### Register-Größen

- 8 bits → Byte (uint8\_t)
- 16 bits → Halfword (uint16\_t)
- 32 bits → Word (uint32\_t)

#### Register an spezifischer Adresse zugreifen

```
1 #define MY_SIZE_REG (*((volatile uintX_t *)(0x...)))
```

#### Operationen

- Wert aus REG in Variable einlesen: var = MY\_REG;
- REG auf bestimmten Wert setzen: MY\_REG = 0x...;
- Einzelne Bits verandern: MY\_REG |= 0x...; oder MY\_REG &= ~0x...;
- Warten bis sich Wert ändert: while (!(MY\_REG & 0x...)) {}

#### Wichtige Punkte

- Immer volatile verwenden für Hardware-Register
- Korrekte Datentypen (uint8\_t, uint16\_t, uint32\_t) verwenden
- Bit-Masken für einzelne Bits verwenden

## Memory-Mapped Register Access in C

### Define register addresses

- Use #define for each register address
- Create pointer types for each data width (8-bit, 16-bit, 32-bit)
- Use volatile qualifier to prevent optimization

### Reading from registers

- Cast the register address to a volatile pointer of appropriate width
- Dereference the pointer to read the value
- Use bit masks and shifts to extract specific bits if needed

### Writing to registers

- Cast the register address to a volatile pointer of appropriate width
- Assign a value to the dereferenced pointer to write
- For bit manipulation operations:
  - Use bitwise OR (|) to set specific bits without affecting others
  - Use bitwise AND (&) with inverted mask to clear specific bits
  - Use bitwise XOR (^) to toggle specific bits

### Waiting for status bits

- Create a polling loop that checks the status bit
- Use appropriate bit masks to isolate the status bit
- Use volatile to ensure the register is read on each iteration

### Register-Zugriffe in C Verschiedene Aufgaben mit Control Registern:

- 8-bit Register an 0x6100'0007 lesen
- 16-bit Register an 0x6100'0008 auf 0xFFFF setzen
- Warten bis Bit 15 in 32-bit Register gesetzt ist
- Bit 16 setzen ohne andere Bits zu veraendern

```
1 // a) 8-bit Register lesen
2 #define MY_BYTE_REG (*((volatile uint8_t *) (0x61000007)))
3 uint8_t my_var;
4 my_var = MY_BYTE_REG;
5
6 // b) 16-bit Register setzen
7 #define MY_HALFWORD_REG (*((volatile uint16_t *) (0x61000008)))
8 MY_HALFWORD_REG = 0xFFFF;
9
10 // c) Warten auf Bit 15
11 #define MY_WORD_REG (*((volatile uint32_t *) (0x6100000C)))
12 while (!(MY_WORD_REG & 0x00008000)){
13 }
14
15 // d) Bit 16 setzen
16 #define MY_WORD_REG2 (*((volatile uint32_t *) (0x61000010)))
17 MY_WORD_REG2 |= 0x00010000;
```

# GPIO (General Purpose I/O)

## GPIO Fundamentals

### GPIO

#### Situation:

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

#### Problem:

- Limited number of pins
- Many functions to be implemented (many functional blocks included)

### GPIO Overview

General Purpose Input/Output (GPIO) pins allow the microcontroller to interact with the external world.

- Pins can be configured as digital inputs or outputs
- Most pins support multiple functions (pin sharing) through internal multiplexers
- Configuration is done through memory-mapped registers
- Each GPIO port typically has 16 pins (e.g., GPIOA, GPIOB, etc.)

#### Solution:

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

#### Pin Sharing

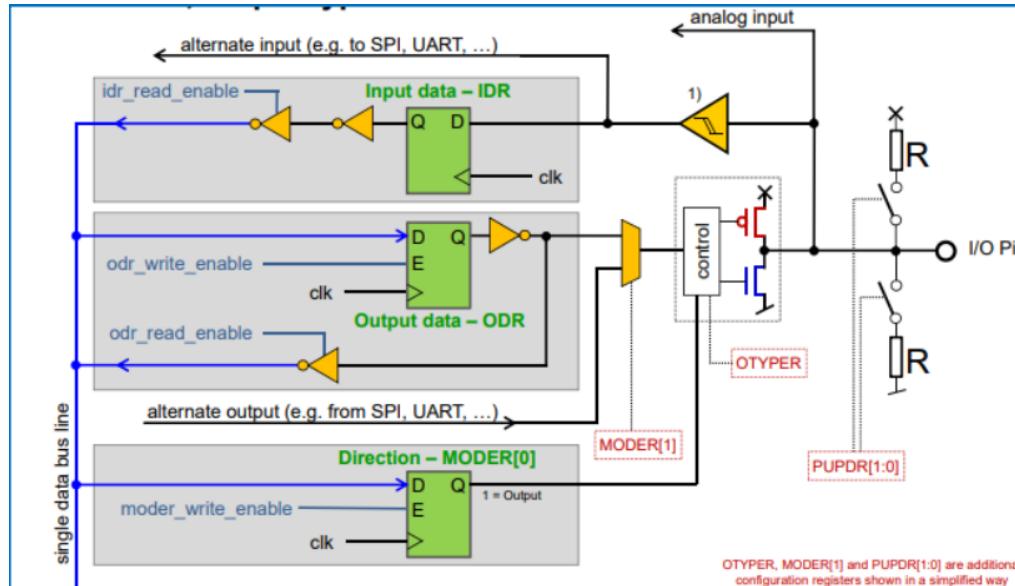
Multiple functions can share a single physical pin:

- Digital inputs/outputs (GPIO)
- Serial interfaces (UART, SPI, I2C)
- Timers/Counters
- ADC (Analog-to-Digital Conversion)
- Alternate functions

Not all functions can be used simultaneously; configuration registers define pin usage.

## GPIO Structure

### Structure



### Push-Pull vs Open-Drain Outputs

#### Push-Pull:

- Can actively drive output high (to VDD) or low (to GND)
- Faster switching times, can source and sink current
- Default output mode for GPIO pins

#### Open-Drain:

- Can only actively drive output low
- Relies on external pull-up resistor to reach high state
- Multiple devices can share a line without conflicts (e.g., I2C bus)
- Used in "wired-AND" configurations

## Data Registers and Operations

### Register Access

#### Register Address

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

#### GPIO Registers

Each GPIO port has several configuration registers:

- **GPIOx\_MODER**: Mode register (input, output, alternate function, analog) - configures pin as input or output (direction control)
- **GPIOx\_OTYPER**: Output type register (push-pull or open-drain)
- **GPIOx\_OSPEEDR**: Output speed register (low, medium, high, very high) - configures speed
- **GPIOx\_PUPDR**: Pull-up/pull-down register
- **GPIOx\_IDR**: Input data register (read-only) - reads the pin state
- **GPIOx\_ODR**: Output data register (read/write) - sets the output state
- **GPIOx\_BSRR**: Bit set/reset register (atomic operations)
- **GPIOx\_LCKR**: Configuration lock register
- **GPIOx\_AFRL/H**: Alternate function registers (AF selection for pins 0-7 and 8-15)

#### Data Operations

- Input: Read register **GPIOx\_IDR** (Input Data Register)
- Output: Write register **GPIOx\_ODR** (Output Data Register) or **GPIOx\_BSRR** (Bit Set Reset Register)

## Hardware Abstraction Layer (HAL)

### HAL for GPIO

The Hardware Abstraction Layer provides a structured way to access GPIO registers:

- Based on structs that map to hardware registers
- Typedef for register structure (e.g., `reg_gpio_t`)
- Pointers to each GPIO port (e.g., GPIOA, GPIOB)
- Base addresses defined in header file
- Helper macros for bit manipulation

This approach makes code more readable and maintainable than direct register address manipulation.

```
#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 to GPIOI

#### Using HAL for GPIO

```

1 // Using HAL style access
2 #include "reg_stm32f4xx.h" // Contains structure definitions
3
4 // Configure PA3 as output
5 GPIOA->MODER &= ~(3 << 6); // Clear bits 6-7
6 GPIOA->MODER |= (1 << 6); // Set bit 6 (output mode)
7
8 // Instead of direct register access:
9 // volatile uint32_t *reg = (volatile uint32_t *) (0x40020000);
10 // *reg &= ~(3 << 6);
11 // *reg |= (1 << 6);

```

## Hardware Abstraction Layer (HAL)

### reg\_stm32f4xx.h

#### Base addresses

##### Pointers to struct of type reg\_gpio\_t

```
#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 )
```

base addresses

#### Offset

##### Typedef for reg\_gpio\_t

```
/*
 * \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;
```

register names as  
in reference manual

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

size of registers

## GPIO Configuration

### SEP Handout: Datenblattauszug GPIO S.2-3 Addresses and Configurations

#### Input Data Register (IDR)

The GPIOx\_IDR is a read-only register containing the input value of the corresponding I/O port.

#### Direction Configuration (MODER)

The GPIOx\_MODER register configures each pin's direction:

- 00: Input mode
- 01: General purpose output mode
- 10: Alternate function mode
- 11: Analog mode

Each pin uses 2 bits in the register, allowing for 16 pins per port.

#### Output Type (OTYPER)

The GPIOx\_OTYPER register configures the output driver type:

- 0: Push-pull (can actively drive high or low)
- 1: Open-drain (can drive low, relies on external pull-up for high)

#### Pull-up/Pull-down (PUPDR)

The GPIOx\_PUPDR register configures internal resistors:

- 00: No pull-up, no pull-down
- 01: Pull-up
- 10: Pull-down
- 11: Reserved

#### Output Data Register (ODR)

The GPIOx\_ODR can be read and written to set the output state of GPIO pins.

#### Speed Configuration (OSPEEDR)

The GPIOx\_OSPEEDR register configures the output slew rate:

- 00: Low speed
- 01: Medium speed
- 10: High speed
- 11: Very high speed

#### Bit Set/Reset Register (BSRR)

The GPIOx\_BSRR allows atomic bit operations:

- Bits [15:0]: Set corresponding ODR bit by writing '1'
- Bits [31:16]: Reset corresponding ODR bit by writing '1'

This ensures atomic access without read-modify-write operations.

#### Setting and Clearing Bits GPIOx\_BSRR

- 0-15: Set Bits → 1: Set, 0: No Change
- 16-31: Reset/Clear Bits → 1: Reset, 0: No Change
- Ensures atomic access in software (no interruption possible)

## GPIO Configuration KR and examples

### Configuring GPIO Pins

Identify GPIO port and pin

- Find the port letter (A-K) and pin number (0-15) from documentation
- Look up the corresponding pin number on the microcontroller package

Calculate register addresses

- Find the base address for the GPIO port in the memory map
- Calculate register addresses by adding offsets to the base address
- For STM32F4: Base addresses are at 0x4002 0000 + (0x400 \* port\_index)
  - GPIOA: 0x4002 0000
  - GPIOB: 0x4002 0400
  - GPIOC: 0x4002 0800
  - And so on...

Enable clock for GPIO port

- Use RCC\_AHB1ENR register to enable clock for the GPIO port
- Set the corresponding bit for the port

Configure specific pin properties

- Use GPIOx\_MODER to set pin mode/direction (input, output, alternate function, analog)
- Use GPIOx\_OTYPER to set output type (push-pull or open-drain)
- Use GPIOx\_OSPEEDR to set output speed (low, medium, high, very high)
- Use GPIOx\_PUPDR to configure pull-up/pull-down resistors

Set initial state (for outputs)

For output pins, set the initial state in ODR or using BSRR.

**GPIO Configuration Exercise** Configure pin PA3 as an output with open-drain configuration, low speed, and pull-up enabled. Then set the pin to low state.

First, identify the registers and their addresses:

- GPIOA base address: 0x4002 0000
- GPIOA\_MODER: 0x4002 0000 (offset 0x00)
- GPIOA\_OTYPER: 0x4002 0004 (offset 0x04)
- GPIOA\_OSPEEDR: 0x4002 0008 (offset 0x08)
- GPIOA\_PUPDR: 0x4002 000C (offset 0x0C)

Then, calculate the bit fields for pin 3:

- MODER3[1:0] = 01 (output mode) at bits 7:6
- OTYPER3 = 1 (open-drain) at bit 3
- OSPEEDR3[1:0] = 00 (low speed) at bits 7:6
- PUPDR3[1:0] = 01 (pull-up) at bits 7:6

```

1 // Enable GPIOA clock
2 RCC->AHB1ENR |= (1 << 0); // Bit 0 for GPIOA
3 // Configure PA3 as output (bits 6-7 = 01)
4 GPIOA->MODER &= ~(3 << 6); // Clear bits 6-7
5 GPIOA->MODER |= (1 << 6); // Set bit 6 (output mode)
6 // Configure as open-drain (bit 3 = 1)
7 GPIOA->OTYPER |= (1 << 3);
8 // Configure as low speed (bits 6-7 = 00)
9 GPIOA->OSPEEDR &= ~(3 << 6); // Clear bits 6-7 (low speed)
10 // Configure with pull-up (bits 6-7 = 01)
11 GPIOA->PUPDR &= ~(3 << 6); // Clear bits 6-7
12 GPIOA->PUPDR |= (1 << 6); // Set bit 6 (pull-up)
13 // Set pin to low state using BSRR
14 GPIOA->BSRR = (1 << (3 + 16)); // Set bit 19 (reset PA3)
15 // or using ODR
16 GPIOA->ODR &= ~(1 << 3); // Set PA3 low
```

## Reading and Writing GPIO

### Reading Input Pins

Read the current state of GPIO pins using the IDR register.

### Writing Output Pins - Using ODR

Set or clear output pins by modifying the ODR register.

### Writing Output Pins - Using BSRR (preferred)

Set or clear output pins atomically using the BSRR register.

```
1 // Reading input from GPIOA pin 0
2 uint32_t buttonState = (GPIOA->IDR & (1 << 0)) != 0;
3
4 // Writing output using ODR (not atomic)
5 // Set pin high
6 GPIOA->ODR |= (1 << 5);
7 // Set pin low
8 GPIOA->ODR &= ~(1 << 5);
9
10 // Writing output using BSRR (atomic)
11 // Set pin high (bits 0-15)
12 GPIOA->BSRR = (1 << 5);
13 // Set pin low (bits 16-31)
14 GPIOA->BSRR = (1 << (5 + 16));
```

## GPIO LED und Button Konfiguration

### STM32F429 Discovery Board:

- User Button B1: GPIO Port A, Pin 0
- User LED LD3 (grün): GPIO Port G, Pin 13
- User LED LD4 (rot): GPIO Port G, Pin 14

Konfiguriere Pin 13 als Output mit Open-Drain und Pull-up.

### Register und Adressen:

- GPIOG\_MODER: 0x40021800
- GPIOG\_OTYPER: 0x40021804
- GPIOG\_OSPEEDR: 0x40021808
- GPIOG\_PUPDR: 0x4002180C

### Bit-Konfiguration für Pin 13:

- MODER[27:26] = 01 (Output Mode)
- OTYPER[13] = 1 (Open-Drain)
- OSPEEDR[27:26] = 01 (Medium Speed)
- PUPDR[27:26] = 01 (Pull-up)

### C-Code:

```
1 // LED ein-/ausschalten mit BSRR
2 // LED einschalten (Pin 13)
3 GPIOG->BSRR = (1 << 13);
4
5 // LED ausschalten (Pin 13)
6 GPIOG->BSRR = (1 << (16+13));
7
8 // Button lesen (Pin 0)
9 uint32_t button_state = GPIOA->IDR & 0x1;
```

Das BSRR (Bit Set/Reset Register) ermöglicht atomare Bit-Operationen. Die unteren 16 Bits (0-15) setzen die entsprechenden Pins, die oberen 16 Bits (16-31) löschen die entsprechenden Pins (Bit 16 löscht Pin 0, Bit 17 löscht Pin 1, etc.).

## Using GPIO Data Registers

### Reading input pins

- Use GPIOx\_IDR register to read the state of input pins
- Bit n corresponds to pin n (0-15)
- Apply appropriate bit mask to extract the bit(s) of interest

### Writing output pins

- Use GPIOx\_ODR register to write to output pins
- Bit n corresponds to pin n (0-15)
- Writing to ODR affects all bits at once, which can lead to race conditions

### Atomic bit manipulation

- Use GPIOx\_BSRR register for atomic bit setting/clearing:
  - Bits 15:0: Setting corresponding ODR bits (write 1 to set)
  - Bits 31:16: Clearing corresponding ODR bits (write 1 to clear)
- Example: To set pin 5, write (1 « 5) to BSRR
- Example: To clear pin 5, write (1 « (16+5)) to BSRR

## GPIO Data Operations Example

### Write code to:

1. Read the state of user button B1 on port A, pin 0
2. Turn on the green LED LD3 on port G, pin 13 when B1 is pressed
3. Turn on the red LED LD4 on port G, pin 14 when B1 is not pressed

### First, define the register addresses and pin mappings:

```
1 // Button B1 is on PA0
2 // Green LED LD3 is on PG13
3 // Red LED LD4 is on PG14
4
5 #include "reg_stm32f4xx.h"
6
7 void button_led_control(void) {
8     // Read button state (PA0)
9     uint32_t button_state = GPIOA->IDR & (1 << 0);
10
11    if (button_state) {
12        // Button pressed - turn on green LED (PG13) and turn off red LED (PG14)
13        // Set bit 13, clear bit 14 atomically
14        GPIOG->BSRR = (1 << 13) | (1 << (16+14));
15    } else {
16        // Button not pressed - turn on red LED (PG14) and turn off green LED (PG13)
17        // Set bit 14, clear bit 13 atomically
18        GPIOG->BSRR = (1 << 14) | (1 << (16+13));
19    }
20 }
```

Note: This assumes that the GPIO ports have been properly configured:

- PA0 (Button B1) as input with pull-up/pull-down as needed
- PG13 (LED LD3) as push-pull output
- PG14 (LED LD4) as push-pull output

## Serial Data Transfer

### Serial Communication Basics

#### Serial vs. Parallel Communication

Microcontrollers often use serial connections for communication:

**Serial Connection:** Data transmitted one bit at a time over fewer wires

- Simpler (saves PCB area)
- Reduces number of switching lines (reduced power, improved EMC)
- Requires higher-level protocol for interpretation

**Parallel Bus:** Data transmitted over multiple lines simultaneously

- Faster for short distances
- More complex routing
- Higher power consumption and EMC issues

#### Serial Communication Modes

- **Simplex:** Unidirectional, one-way only communication
- **Half-duplex:** Bidirectional, but only one direction at a time
- **Full-duplex:** Bidirectional, both directions simultaneously

#### Serial Communication Timing

**Synchronous:** Both nodes use the same clock

- Clock often provided by master
- Examples: SPI, I2C

**Asynchronous:** Each node uses an individual clock

- Start/stop bits used for synchronization
- Example: UART

### Overview of Serial Communication Protocols

#### Overview

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 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.

#### SPI - Serial Peripheral Interface

- Master/slave architecture
- Synchronous full-duplex transmissions (MOSI, MISO) with shared clock
- Selection of device through Slave Select (SS) → multiple slaves possible, !! separate SS line for each slave
- no acknowledge, no error detection
- Clock signal (SCK) for synchronization: four mode → CPOL (clock polarity), CPHA (clock phase)
- High speed (up to 50+ Mbps)

#### UART - Universal Asynchronous Receiver Transmitter (Serial Interface)

- Transmitter and receiver use diverging clocks
- Asynchronous (no shared clock) → synchronization using start and stop bits → overhead
- longer connections require line drivers → RS-232/RS-485
- simple wiring (2-3 wires)
- Moderate speed (up to 5 Mbps)

#### I2C - Inter-Integrated Circuit

- Multi-master, multi-slave
- Synchronous half-duplex transmission (SCL, SDA) with shared clock - SCL for synchronization
- 7-bit slave addresses
- Acknowledge, error detection
- Medium speed (100 kbps, 400 kbps, up to 5 Mbps)

### Protocol Comparison and Selection

#### Selecting the Appropriate Serial Protocol

Consider application requirements -

- **Distance:**
  - UART with RS-232 levels for longer distances
  - SPI and I2C typically for on-board connections
- **Speed:**
  - SPI for highest speed requirements
  - I2C for moderate speed with fewer pins
  - UART for simpler, moderate speed connections
- **Number of devices:**
  - I2C for many devices on shared bus
  - SPI for multiple devices but requires separate SS line for each
  - UART typically for point-to-point (multiple UARTs needed for multiple devices)

Evaluate protocol overhead -

- **UART:** Start, stop, and optional parity bits and requires accurate clock timing
- **SPI:** Minimal protocol overhead, no addressing or acknowledgment
- **I2C:** Start/stop conditions, addressing, and acknowledgment → higher overhead, but better error detection

Consider hardware support -

- Check if target microcontroller has hardware support for chosen protocol
- Consider available pins and alternate functions
- Evaluate available software libraries and drivers
- Consider power requirements (I2C has pull-up resistors that consume power)

Select the most appropriate serial protocol for each of the following applications:

1. A system needs to connect a microcontroller to three temperature sensors. The sensors are low-cost devices that should be replaceable without redesigning the PCB. Data rate requirement is low.
2. A data acquisition system needs to transfer large amounts of data from an ADC to a microcontroller at 20 Mbps.
3. A microcontroller needs to communicate with a PC via USB port.
4. A control system needs to connect to four different devices at varying distances up to 30 meters in an electrically noisy industrial environment.

Solution:

1. **I2C is most appropriate for the temperature sensors:**
  - Multiple devices (three sensors) can be connected to the same two wires
  - Each sensor has a unique address, making them individually addressable
  - Low data rate requirement is well within I2C capabilities
  - Sensors can be replaced without changing connections (as long as addresses are configured properly)
  - Reduced pin count (only SCL and SDA) simplifies PCB design

2. **SPI is most appropriate for the high-speed ADC:**

- 20 Mbps data rate exceeds practical I2C speeds and most UART configurations
- SPI can easily handle 20+ Mbps with direct hardware support
- SPI's full-duplex capability allows simultaneous control and data transfer
- Minimal protocol overhead maximizes throughput for large data amounts
- Hardware-based chip select ensures proper timing for ADC operations

3. **UART is most appropriate for PC communication:**

- UART is the typical protocol used with USB-to-serial converter chips
- Simple point-to-point connection is sufficient for PC communication
- Standard baudrates (9600, 115200, etc.) are well-supported by PC software
- No need for extra clock signals simplifies interfacing with USB adapters
- Asynchronous nature accommodates timing differences between systems

4. **RS-485 (based on UART) is most appropriate for the industrial control system:**

- RS-485 extends UART to longer distances (up to 1200m)
- Differential signaling provides excellent noise immunity in industrial environments
- Multi-drop capability allows connection to four devices on a single bus
- Higher voltage levels offer better signal integrity over 30m distances
- Various data rates possible depending on cable length (up to 10 Mbps at shorter distances)
- Industry standard protocol with wide hardware support

# SPI - Serial Peripheral Interface

- SPI** SPI is a synchronous serial bus primarily for on-board connections:
- Used for short-distance communication
  - Connects microcontroller to external devices (sensors, displays, flash memory, etc.)

## SPI Properties

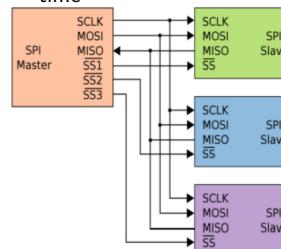
- No defined addressing scheme (uses SS lines instead) → KISS (Keep It Simple Stupid)
- No built-in acknowledgment or error detection (implemented in higher-level protocols)
- Originally for single-byte transfers
  - SS deactivated after each byte
  - Today also used for streams
- Flexible data rate (clock signal is transmitted with data)
- No flow control (master controls timing, slave cannot influence)
- Susceptible to noise (spikes on clock signal)

## SPI Signals

- SCLK** (Serial Clock): Generated by master
- MOSI** (Master Out Slave In)
- MISO** (Master In Slave Out)
- SS/CS** (Slave Select/Chip Select)

## SPI Master-Slave Architecture

- Master generates common clock signal (SCLK) for all slaves
- Slaves: Selectable by Slave Select (SS) signal
  - Individual Select  $SS_1, SS_2, SS_3$
  - $SS_x = 1 \rightarrow$  slave output  $MISO_x$  is tri-state
- MOSI line connects to inputs of all slaves
- MISO lines from all slaves are connected to a single master input
- Inactive slaves ( $SS = '1'$ ) put their MISO line in tri-state (high impedance)
- Only one selected slave drives its MISO line at a time



## SPI Modes and Timing

### Clock Polarity and Phase

**CPOL** (Clock Polarity): Idle state of clock

- CPOL = 0: Clock idles at low level
- CPOL = 1: Clock idles at high level

**CPHA** (Clock Phase): Which edge is used for data sampling

- CPHA = 0: Data sampled on first clock edge
- CPHA = 1: Data sampled on second clock edge

### Four SPI Modes

**Mode 0:** CPOL = 0, CPHA = 0

- Clock idles low
- Data sampled on rising edge, changed on falling edge

**Mode 2:** CPOL = 1, CPHA = 0

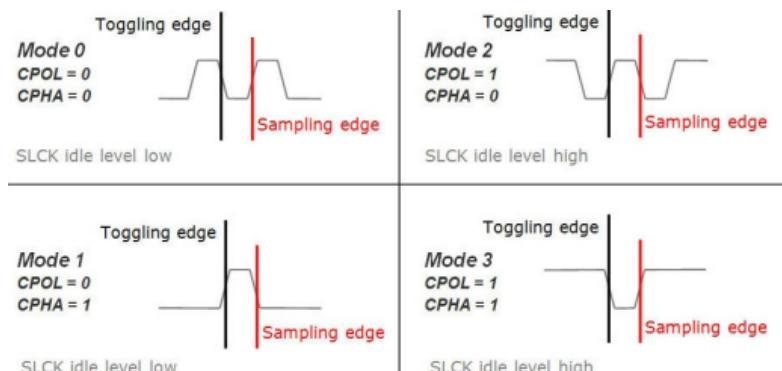
- Clock idles high
- Data sampled on falling edge, changed on rising edge

**Mode 3:** CPOL = 1, CPHA = 1

- Clock idles high
- Data sampled on rising edge, changed on falling edge

**Mode 1:** CPOL = 0, CPHA = 1

- Clock idles low
- Data sampled on falling edge, changed on rising edge

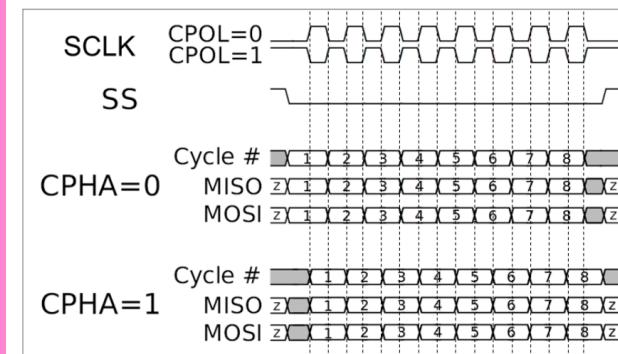


## SPI Timing-Diagramm zeichnen

SPI-Modi (CPOL/CPHA)

- Mode 0: CPOL=0, CPHA=0 (Idle=Low, Sample=Rising)
- Mode 1: CPOL=0, CPHA=1 (Idle=Low, Sample=Falling)
- Mode 2: CPOL=1, CPHA=0 (Idle=High, Sample=Falling)
- Mode 3: CPOL=1, CPHA=1 (Idle=High, Sample=Rising)

Timing zeichnen



Calculate timing parameters

- Bit cell time = 1/clock\_frequency
- Frame time = (number\_of\_bits)/clock\_frequency
- Example: 8 bits at 100 kHz = 80 µs per frame

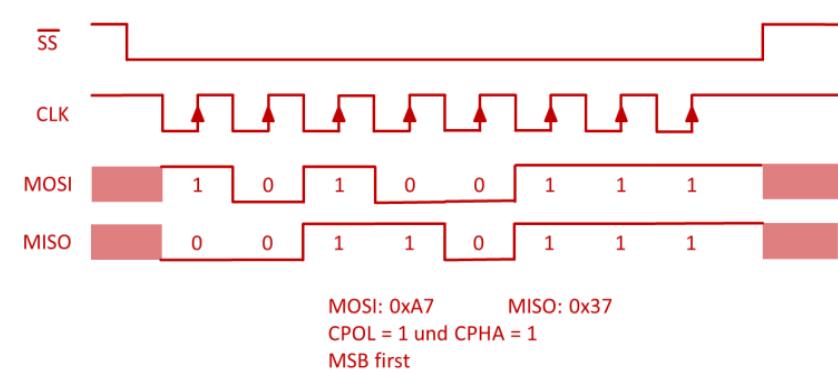
## SPI Mode 3 Timing

SPI Interface mit 8 Bit Daten:

- MOSI: 0xA7 (10100111b)
- MISO: 0x37 (00110111b)

- Mode 3: CPOL=1, CPHA=1
- MSB first
- SCLK: 100kHz

Timing-Diagramm:



### Timing-Charakteristika:

- SCLK Idle = High (CPOL=1)
- Daten wechseln auf steigende Flanke (CPHA=1)
- Sampling auf fallende Flanke
- Bit-Zeit = 1/100kHz = 0.01 ms = 10 µs

### Daten-Übertragung:

- SS: Low während gesamter Übertragung
- SCLK: startet High, 8 Taktzyklen
- MOSI: 1-0-1-0-0-1-1-1 (MSB first)
- MISO: 0-0-1-1-0-1-1-1 (MSB first)

Bei SPI ist die Übertragung immer full-duplex - Master und Slave senden gleichzeitig. Auch wenn nur in eine Richtung Daten benötigt werden, findet auf der anderen Leitung eine "DummyÜbertragung statt.

Signale

- SS** (Slave Select): aktiv-low, Slave-Auswahl
- SCLK**: Clock vom Master
- MOSI**: Master Out Slave In
- MISO**: Master In Slave Out

# SPI Implementation

## SPI Implementation Using Shift Registers

- Both master and slave contain 8-bit shift registers
- Bits are shifted out on one clock edge (toggling edge)
- Bits are sampled on the other clock edge (sampling edge)
- 8 clock cycles exchange 8 bits in each direction simultaneously
- After 8 clock cycles, data has been exchanged in both directions
- Status flags (TXE, RXNE) indicate buffer status

## STM32 SPI Register Configuration

```
1 // SPI configuration example
2 // Configure SPI1 in master mode, CPOL=1, CPHA=1, 8-bit data
3
4 // Enable SPI1 clock
5 RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;
6
7 // Configure SPI1
8 SPI1->CR1 = (1 << 0) // CPHA=1
9   | (1 << 1) // CPOL=1
10  | (1 << 2) // Master mode
11  | (3 << 3) // BR[2:0]=011: fPCLK/16 (prescaler)
12  | (0 << 7) // MSB first (LSBFIRST=0)
13  | (1 << 8) // SSI=1 (needed for software SS)
14  | (1 << 9); // SSM=1 (software slave management)
15
16 // Enable SPI
17 SPI1->CR1 |= (1 << 6); // SPE=1 (SPI enable)
```

## Transmitting Data with SPI

### Step 1: Prepare SPI

Configure and enable the SPI peripheral.

### Step 2: Check TXE flag

Wait until the transmit buffer is empty.

### Step 3: Write data

Write data to the data register.

### Step 4: Wait for completion

Wait until transmission is complete by checking BSY flag.

```
1 // Transmit a byte over SPI
2 uint8_t transmit_byte(uint8_t data) {
3     // Step 1: SPI should already be configured
4
5     // Step 2: Wait until TXE=1 (transmit buffer empty)
6     while (!(SPI1->SR & (1 << 1))) { }
7
8     // Step 3: Write data to transmit
9     SPI1->DR = data;
10
11    // Step 4: Wait for reception (needed to get received data)
12    while !(SPI1->SR & (1 << 0))) { }
13
14    // Return received data (read DR clears RXNE flag)
15    return SPI1->DR;
16}
```

## STM32 SPI Implementation

### STM32 SPI Architecture

The STM32F4 SPI peripheral includes:

- Configuration registers (SPI\_CR1, SPI\_CR2)
- Status register (SPI\_SR)
- Data register (SPI\_DR) for transmit and receive
- Status flags for synchronization (TXE, RXNE, BSY)
- Support for different communication modes
- DMA capability for high-speed transfers

## Handling Full-Duplex SPI Communication

- Enable SPI with SPE bit: Ensure SPI is properly configured and enabled.
- Write first byte to transmit: Write the first byte to the data register (DR) to start transmission/reception.
- Process data in a loop: Check TXE (Transmit Buffer Empty) and RXNE (Receive Buffer Not Empty) flags to handle both transmission and reception.
- Wait for completion: Wait for BSY (Busy) flag to be cleared, indicating that all transfers are complete.

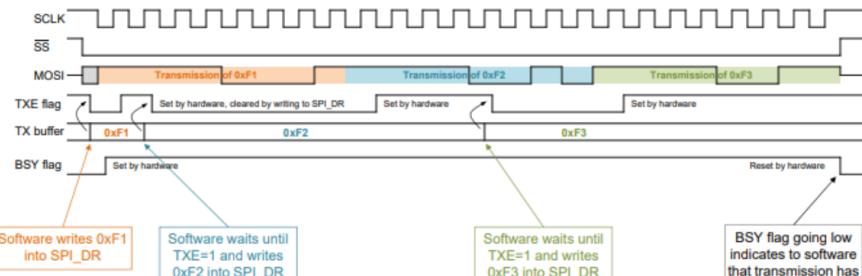
```
1 void spi_transfer(uint8_t *tx_data, uint8_t *rx_data, uint16_t length) {
2     // Enable SPI
3     SPI1->CR1 |= (1 << 6); // SPE=1
4     if (length > 0) { // Write first TX byte to start transmission
5         SPI1->DR = tx_data[0];
6     }
7     uint16_t tx_count = 1;
8     uint16_t rx_count = 0;
9     while (rx_count < length) { // Process data
10        // Check if we can transmit more data
11        if ((tx_count < length) && (SPI1->SR & (1 << 1))) { // TXE=1
12            SPI1->DR = tx_data[tx_count++];
13        }
14        // Check if we received data
15        if (SPI1->SR & (1 << 0)) { // RXNE=1
16            rx_data[rx_count++] = SPI1->DR;
17        }
18    }
19    while (SPI1->SR & (1 << 7)) { } // Wait until BSY=0 (SPI not busy)
20 }
```

## Synchronizing Hardware and Software

- TXE (TX Buffer Empty) → Software can write next TX Byte to SPI\_DR
- RXNE (RX Buffer Not Empty) → a byte has been received. Software can read it from SPI\_DR
- BSY (Busy) → Transmission in progress

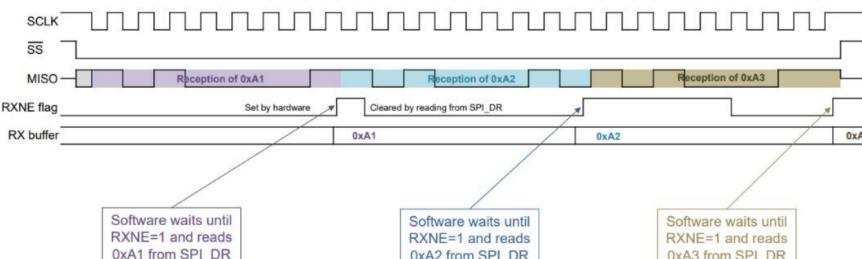
### Transmitting in Software

- Example: SW wants to transmit bytes 0xF1, 0xF2, 0xF3



### Receiving in Software

- Example: SW receives bytes 0xA1, 0xA2, 0xA3



# UART and I2C

## UART - Universal Asynchronous Receiver Transmitter

### UART

is an asynchronous serial communication interface:

#### Asynchronous data transfer

- Mismatch of clock frequencies in TX and RX → no shared clock between transmitter and receiver
- Overhead for synchronization → extra Bits
- Effort for synchronization → extra Hardware

#### Full-duplex communication

- Point-to-point communication (one transmitter, one receiver)
- Full-duplex (simultaneous bidirectional communication)
- Start and stop bits used for synchronization
- Typically 2-wire interface (TX and RX) for data
- Optional control signals (RTS/CTS, etc.) for flow control

#### Advantages

- Clock does not have to be transmitted
- transmission delays automatically compensated
- no need for a common clock signal

#### Disadvantages

- Synchronization at start of each data item
- Limited to point-to-point communication
- Slower than synchronous protocols (e.g., SPI)
- More complex error handling due to asynchronous nature

### UART Synchronization

- Receiver **detects start bit** (falling edge from idle to '0')
- Receiver **samples** middle of each bit based on configured baud rate
- Both devices must use the same **configuration**:
  - Baud rate (e.g., 9600, 115200 bits/s)
  - Number of data bits (5-8)
  - Parity (none, odd, even, mark, space)
  - Number of stop bits (1, 1.5, 2)

## UART Timing and Calculations

### UART Calculations

$$\text{Bit Time (seconds)}: T_{bit} = \frac{1}{\text{Baud Rate}}$$

### Frame Time (seconds)

$$T_{frame} = T_{bit} \times \text{Total Bits per Frame (TBF)}$$

$$\text{TBF} = 1 (\text{start}) + \text{Data Bits} + \text{Parity Bits} + \text{Stop Bits}$$

### Maximum Data Rate (bytes/second):

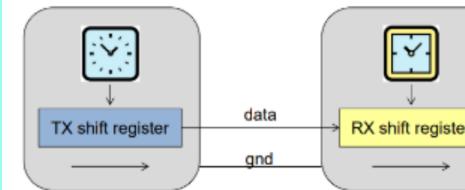
$$\text{Data Rate} = \frac{\text{Baud Rate}}{\text{Total Bits per Byte (TBB)}}$$

$$\text{TBB} = 1 (\text{start}) + \text{Data Bits} + \text{Parity Bits} + \text{Stop Bits}$$

### UART Transmission

Connecting shift registers with diverging clock sources:

- same target frequency
- different tolerances and divider ratios
- requires **synchronization** at start of each data item receiver  
→ each data item (5-8 bits) req. synchron.



### On-Board Connections

- Signal Levels: 3V or 5V with reference to ground
- Off-Board connections req. strong output drivers

### UART Signals

Basic UART connections require:

**TX** (Transmit): Data output (transmitter to receiver)

**RX** (Receive): Data input (transmitter to receiver)

**GND** (Ground): Common reference level

### Extended UART (with hardware flow control):

- **RTS** (Request to Send): Output indicating readiness to send
- **CTS** (Clear to Send): Input indicating partner is ready to receive

### UART Frame Structure

- **Idle state**: Line is high ('1') when no transmission occurs
- **Start bit**: Always '0', signals beginning of frame
- **Data bits**: 5 to 9 bits (typically 8), LSB first
- **Parity bit** (optional): For error detection
- **Stop bit(s)**: 1, 1.5, or 2 bits, always '1'

### Analyzing UART Transmission

Identify frame/bit structure

- Start bit: Always '0' (falling edge)
- Data bits: LSB first, 5-8 bits
- Parity bit: Optional, can be even, odd, mark ('1'), or space ('0')
- Stop bit(s): Always '1' (can be 1, 1.5, or 2 bits)
- Idle state: Line remains high ('1') → no transmission

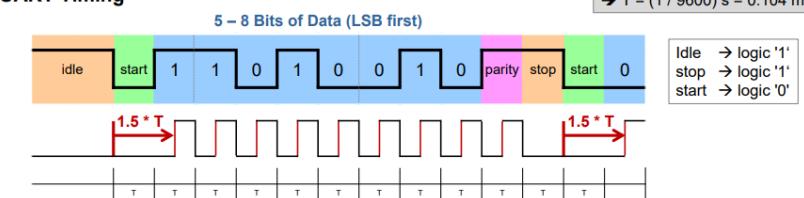
Receiver synchronizes anew at each start bit.

Calculations

- See formulas **UART Calculations** and **Clock Tolerance in UART**
- Clock Tolerance/Taktabweichung: max. deviation = 0.5T (Bit-Zeit) at last data bit (sync. only at start bit)

Timing diagram

### UART Timing



UART Übertragung 'AC'    UART mit 19200 Bit/s, 7 Daten-Bits, 1 Stop-Bit (kein Parity)  
ASCII('A') = 0x41 = 100 0001b, ASCII('C') = 0x43 = 100 0011b

### Calculations:

- Bit-Zeit:  $T = 1/19200 \text{ s} = 52.1 \mu\text{s}$
- Total Bits per Frame (TBF) = 1S (start) + 7D (data) + 1E (stop) = 9 Bits → for each frame/character
- Frame Time:  $T_{frame} = TBF \times T = 9 \text{ Bits} \times 52.1 \mu\text{s} = 469.0 \mu\text{s}$  → for each frame/character
- Total Transmission Time:  $2 \times T_{frame} = 938.0 \mu\text{s}$

Between frames, the line is idle (I) - high.

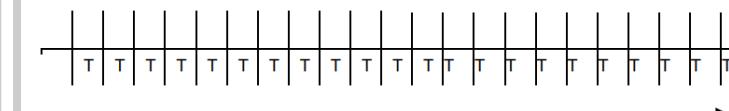
### Timing Diagram:

Frame for 'A' (0x41 = 100 0001b):

- Start bit (S): 0
- Data bits (D), LSB first: 1, 0, 0, 0, 0, 0, 1
- Stop bit (E): 1

Frame for 'C' (0x43 = 100 0011b):

- Start bit (S): 0
- Data bits (D), LSB first: 1, 1, 0, 0, 0, 0, 1
- Stop bit (E): 1



### Legende:

- I = Idle-Bit, S = Start-Bit, D = Daten-Bit, E = Stop-Bit
- LSB first: Daten-Bits werden von rechts nach links übertragen

### Taktabweichung (Clock Deviation):

- Fallende Flanke Start-Bit bis Mitte D6 = 7.5 Bits
- Max. Abweichung für D6: 0.5 Bits
- Taktabweichung:  $100\% \times 0.5 / 7.5 = 6.67\%$
- Bei 19200 Bit/s:  $19200 \times 6.67\% = 1280 \text{ Bit/s}$   
⇒ Empfänger-Takt (Receiver Clock) zwischen 17920 Bit/s und 20480 Bit/s

# UART on STM32F4

## STM32F4 UART/USART Peripherals

- The STM32F4 includes several UART/USART modules with features:
- Full-duplex communication (USART)
  - Programmable baud rate
  - Configurable data bits, stop bits, and parity
  - Interrupt generation on events (TX empty, RX not empty, etc.)
  - DMA support for efficient data transfer
  - Hardware flow control (CTS/RTS) on USARTs
  - Synchronous mode available on USARTs

## STM32F4 UART Configuration

Configure UART2 for 115200 baud, 8-N-1

```

1 // 1. Enable UART2 and GPIO clock
2 RCC->APB1ENR |= RCC_APB1ENR_USART2EN; // Enable UART2 clock
3 RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; // Enable GPIOA clock
4 // 2. Configure GPIO pins for UART (PA2 = TX, PA3 = RX)
5 // Set alternate function mode (0x10)
6 GPIOA->MODER &= ~(0x3 << (2*2) | 0x3 << (2*3));
7 GPIOA->MODER |= (0x2 << (2*2) | 0x2 << (2*3));
8 // Set to AF7 (UART2)
9 GPIOA->AFR[0] &= ~(0xF << (4*2) | 0xF << (4*3));
10 GPIOA->AFR[0] |= (0x7 << (4*2) | 0x7 << (4*3));
11 // 3. Configure UART
12 // Reset UART configuration
13 USART2->CR1 = 0;
14 USART2->CR2 = 0;
15 USART2->CR3 = 0;
16 // Set baud rate (assuming 84MHz APB1 clock)
17 // BRR = f_CK / baud rate = 84,000,000 / 115,200 = 729.16
18 USART2->BRR = 729;
19 // Enable UART, TX, and RX
20 USART2->CR1 = USART_CR1_UE | USART_CR1_TE | USART_CR1_RE;

```

## UART Data Transmission and Reception

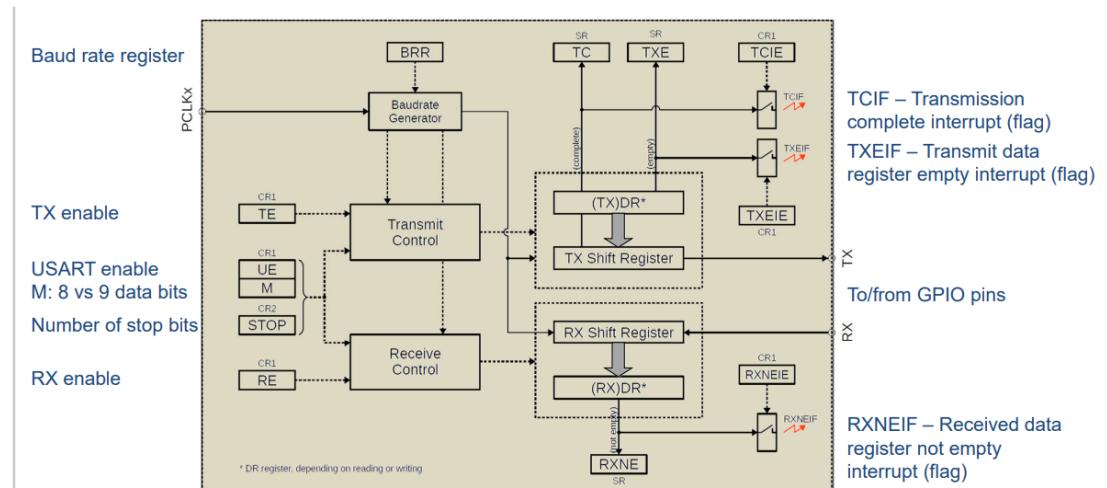
**Transmitting Data** Write data to the data register after checking the TXE flag.

**Receiving Data** Read from the data register after checking the RXNE flag.

```

1 // Send a character
2 void UART_SendChar(USART_TypeDef *uart, char c) {
3     // Wait until TXE flag is set (transmit buffer empty)
4     while (!(uart->SR & USART_SR_TXE)) { }
5     // Write the character to the data register
6     uart->DR = c;
7 }
8 // Send a string
9 void UART_SendString(USART_TypeDef *uart, const char *str) {
10    while (*str) {
11        UART_SendChar(uart, *str++);
12    }
13 }
14 // Receive a character (blocking)
15 char UART_ReceiveChar(USART_TypeDef *uart) {
16     // Wait until RXNE flag is set (data received)
17     while (!(uart->SR & USART_SR_RXNE)) { }
18     // Return received data
19     return uart->DR;
20 }

```



## USART Registers

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0x00	USART_SR																	CTS	LBD	TXE	TC	RXNE	IOE	ORE	NF	FE	1	0							
0x04	USART_DR																	DR[8:0]																	
0x08	USART_BRR																	DIV_Mantissa[15:4]	DIV_Fraction[3:0]																
0x0C	USART_CR1																	Over8	UE	M	VARE	PCE	PS	PEIE	TCIE	TXEIE	TE	FEI	FNU	SBR					
0x10	USART_CR2																	LINEN	STOP[1:0]	CLKEN	CPOL	CRFA	RISE	Received	LEDR	RCEN	ICLKE	TE	FEI	ADD[3:0]					
0x14	USART_CR3																	ONERB	CSE	CRFA	CPOL	DMAR	DMAR	DMAR	DMAR	SCEN	NACK	HSEL	IRLP	IRLN	EIE				

Base addresses of 8 U(S)ART blocks

```

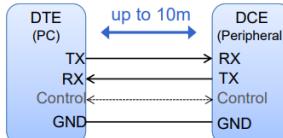
#define USART1 ((reg_usart_t*)0x40011000)
#define USART2 ((reg_usart_t*)0x40004400)
#define USART3 ((reg_usart_t*)0x40004800)
#define USART4 ((reg_usart_t*)0x40004c00)
#define USART5 ((reg_usart_t*)0x40005000)
#define USART6 ((reg_usart_t*)0x40011400)
#define USART7 ((reg_usart_t*)0x40007800)
#define USART8 ((reg_usart_t*)0x40007c00)

```

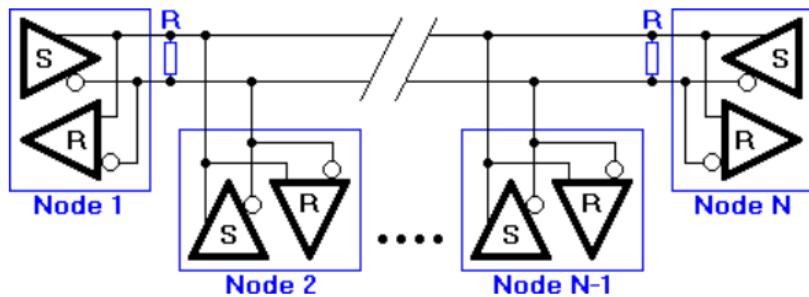
Use of DR and TXE / RXNE is analogous to SPI

**RS232 - Interconnecting Equipment by Cable**

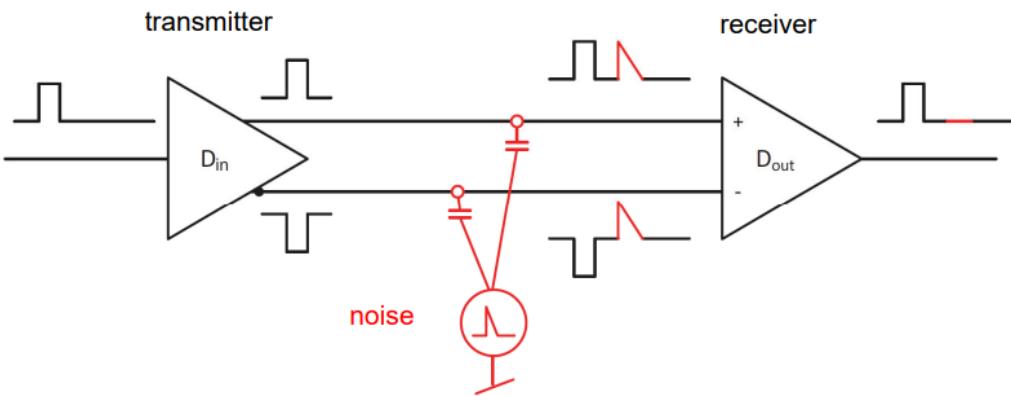
- Simple, bidirectional point-to-point interface based on UART
- Optional control signals, e.g. CTS - Clear To Send
- Ground → common reference level for all signals (single ended)
- Driver circuit allows transmissions up to  $\sim 10$  m
- Logic '1' -3V to -15V and Logic '0' 3 V to 15 V

**RS232 - Interconnecting Equipment by Cable** RS-485 - Differential Transmission Based on UART

- Differential signals
- Less susceptible to disturbances → longer distances, 100+ Meters
- Transmit and receive share the same lines
- Industrial automation: lowest layer of Profibus

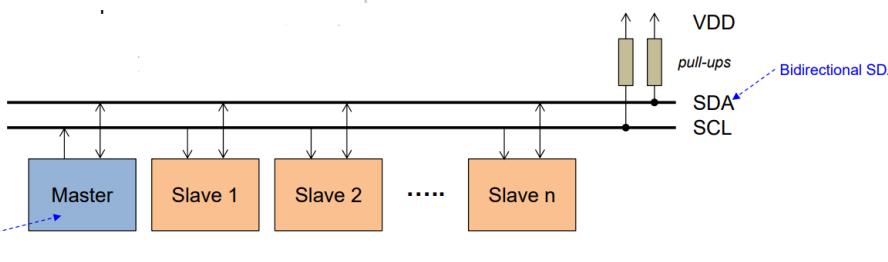
**RS-485 - Differential Transmission**

- Transmitter transforms single-ended signal into differential signal
- Capacitive coupled noise affects both lines
- Receiver forms the difference of the two signals
- Noise on the two lines cancels itself to a large extent



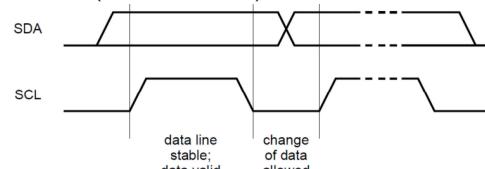
## I2C - Inter-Integrated Circuit

- I2C is a synchronous, half-duplex, multi-master bus:
- Developed by Philips (now NXP) in 1982
  - 2-wire interface (SCL and SDA)
  - Multi-master, multi-slave capability
  - Unique 7-bit or 10-bit address for each device
  - Synchronous (master provides clock)
  - Half-duplex (data flows in one direction at a time)
  - Well-suited for connecting multiple boards or chips
  - Bit rates from 100 kbit/s (standard) to 5 Mbit/s (ultra-fast)



**I2C Signals** I2C requires just two bidirectional lines with pull-up resistors:

- SCL** (Serial Clock Line): Clock signal generated by master
- SDA** (Serial Data Line): Bidirectional data line

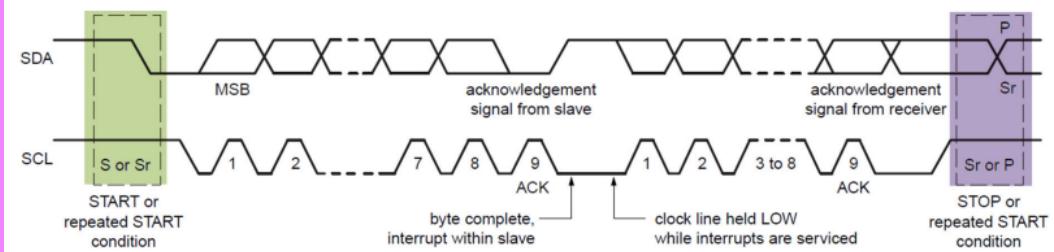


Both lines are open-drain, requiring external pull-up resistors.

**I2C Data Transfer** Data transfer sequence:

- Master initiates transfer with START condition
- Master sends 7-bit slave address + R/W bit (0 = write, 1 = read)
- Addressed slave acknowledges (ACK) by pulling SDA low
- Data bytes transferred (8 bits followed by ACK/NACK)
- Master terminates transfer with STOP condition

Data is transferred MSB first, with the receiver acknowledging each byte.



source: NXP

Bit numbering starts with 1; not with 0

'ACK' is active low

**I2C Write Operation**

Master wants to write data 0x9C to slave with address 0x66

1. Master generates START condition
2. Master sends slave address (0x66) with R/W = 0 (write): 0xCC
3. Slave acknowledges (ACK)
4. Master sends data byte 0x9C
5. Slave acknowledges (ACK)
6. Master generates STOP condition

I2C bus: START - 0xCC - ACK - 0x9C - ACK - STOP

**I2C Read Operation**

Master wants to read data from slave with address 0x66

1. Master generates START condition
2. Master sends slave address (0x66) with R/W = 1 (read): 0xCD
3. Slave acknowledges (ACK)
4. Slave sends data byte (e.g., 0x9C)
5. Master sends NACK to indicate end of transfer
6. Master generates STOP condition

I2C bus: START - 0xCD - ACK - 0x9C - NACK - STOP

### I2C Bus Conditions

- START condition:** SDA goes from high to low while SCL is high
- STOP condition:** SDA goes from low to high while SCL is high
- Data valid:** SDA remains stable while SCL is high
- Data change:** SDA changes only when SCL is low

### I2C Accesses



### I2C Acknowledge and Not-Acknowledge

After each byte transfer:

- ACK (Acknowledge):** Receiver pulls SDA low during 9th clock cycle
- NACK (Not-Acknowledge):** SDA remains high during 9th clock cycle

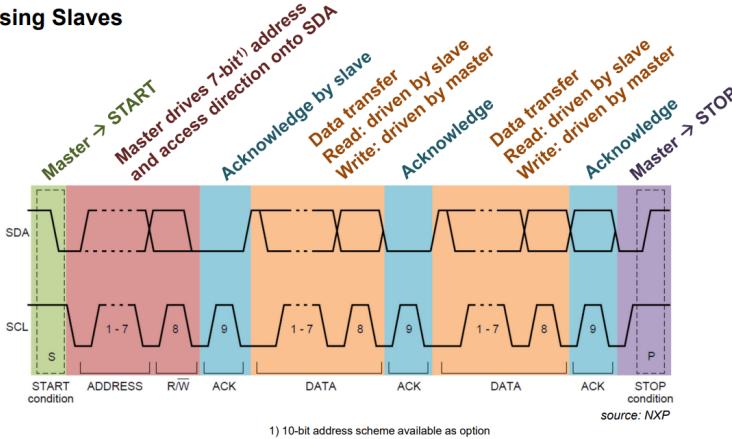
NACK can indicate:

- No receiver with the transmitted address
- Receiver unable to receive or transmit
- Receiver doesn't understand the data/command
- Receiver cannot receive more data
- Master-receiver signals end of transfer to slave-transmitter

## I2C Timing and Protocol

### I2C Timing MSB first!

#### Addressing Slaves



1) 10-bit address scheme available as option

ACK = 0 → Übertragung erfolgreich

ACK = 1 → Übertragung nicht erfolgreich

#### Analyzing I2C Communication

##### Identify I2C protocol elements

- Start condition: SDA falls while SCL is high
- Stop condition: SDA rises while SCL is high
- Data bit transfer: SDA stable while SCL is high
- Acknowledge (ACK): Receiver pulls SDA low during 9th clock cycle
- Not-acknowledge (NACK): SDA remains high during 9th clock cycle

##### Parse I2C frames

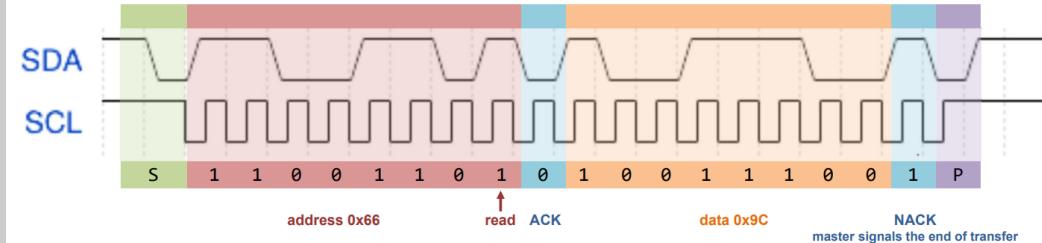
- First byte after start: 7-bit address + R/W bit
  - R/W = 0: Write operation
  - R/W = 1: Read operation
- Check for ACK/NACK after each byte
- Data bytes follow (8 bits each)
- Master or slave can end transmission with STOP condition

##### Interpret timing diagram

- Examine SCL and SDA lines
- Identify start and stop conditions
- Group bits into bytes (8 bits + ACK/NACK)
- Determine if each byte is address or data
- Note the direction of data transfer

## I2C Communication Analysis

Analyze the following I2C timing diagram and describe the transaction: (ACK is active low)



Solution:

The I2C transaction consists of:

1. Start condition (SDA falls while SCL is high)
2. First byte: 11001101 (data transmitted MSB first)
  - This is the address byte: 1100110 (0x66) with R/W bit = 1 (read operation)
  - Followed by an ACK (SDA pulled low during 9th clock cycle)
3. Second byte: 00111001 (data received from slave)
  - This is data: 0x39 or 0x9C (depending on bit order interpretation)
  - Followed by a NACK (SDA remains high during 9th clock cycle)
  - NACK indicates that the master does not want to receive more data
4. Stop condition (SDA rises while SCL is high)

The complete transaction is:

- Master initiates communication with device at address 0x66
- Master requests to read from the device (R/W=1)
- Device acknowledges (ACK)
- Device sends one data byte (0x39 or 0x9C)
- Master signals end of transfer with NACK
- Master terminates communication

## STM32F4 I2C Peripherals

The STM32F4 includes I2C modules with features:

- Compatible with I2C standard protocol
- Multiple speed modes (standard, fast)
- 7-bit and 10-bit addressing
- Multi-master capability
- Programmable clock stretching
- Programmable NOSTRETCH capability
- DMA support for efficient data transfer
- SMBus support

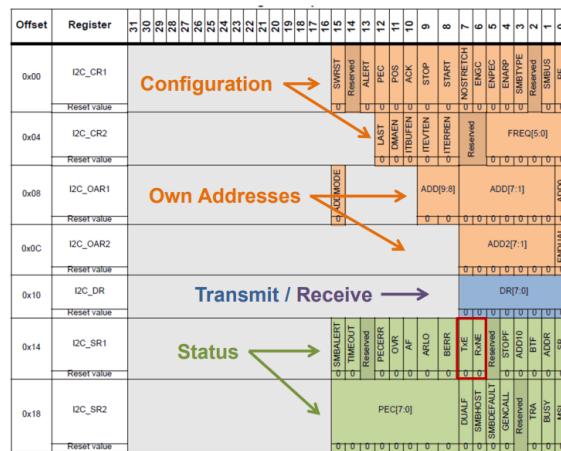
## STM32F4 I2C Configuration

```

1 // Configure I2C1 for 100kHz standard mode
2 // 1. Enable I2C1 and GPIO clock
3 RCC->APB1ENR |= RCC_APB1ENR_I2C1EN; // Enable I2C1 clock
4 RCC->AHB1ENR |= RCC_AHB1ENR_GPIOB; // Enable GPIOB clock
5
6 // 2. Configure GPIO pins for I2C (PB6 = SCL, PB7 = SDA)
7 // Set alternate function mode (0x10)
8 GPIOB->MODER &= ~(0x3 << (2*6) | 0x3 << (2*7));
9 GPIOB->MODER |= (0x2 << (2*6) | 0x2 << (2*7));
10 // Set open-drain output type
11 GPIOB->OTYPER |= (1 << 6) | (1 << 7);
12 // Set to AF4 (I2C1)
13 GPIOB->AFR[0] &= ~(0xF << (4*6) | 0xF << (4*7));
14 GPIOB->AFR[0] |= (0x4 << (4*6) | 0x4 << (4*7));
15
16 // 3. Reset I2C
17 I2C1->CR1 = I2C_CR1_SWRST; // Software reset
18 I2C1->CR1 = 0; // Clear reset
19
20 // 4. Configure I2C
21 // Set peripheral clock frequency (42MHz)
22 I2C1->CR2 = 42; // FREQ = 42MHz
23 // Set CCR for 100kHz standard mode
24 // CCR = PCLK1 / (2 * I2C_FREQ) = 42MHz / (2 * 100kHz) = 210
25 I2C1->CCR = 210;
26 // Set TRISE
27 // TRISE = (max rise time * PCLK1) + 1 = (1000ns * 42MHz) + 1 = 43
28 I2C1->TRISE = 43;
29 // 5. Enable I2C
30 I2C1->CR1 |= I2C_CR1_PE; // Peripheral enable

```

## I2C Registers



Base addresses of 3 I2C blocks

```

#define I2C1 ((reg_i2c_t *)0x40005400)
#define I2C2 ((reg_i2c_t *)0x40005800)
#define I2C3 ((reg_i2c_t *)0x40005C00)

```

## I2C Master Transmit and Receive

### Master Transmit Sequence

Generate START, send address+write, send data bytes, generate STOP.

### Master Receive Sequence

Generate START, send address+read, receive data bytes with ACK/NACK, generate STOP.

```

1 // I2C Master Transmit
2 void I2C_MasterTransmit(I2C_TypeDef *i2c, uint8_t address, uint8_t *data, uint16_t size) {
3     // 1. Wait until I2C bus is free
4     while (i2c->SR2 & I2C_SR2_BUSY) { }
5     // 2. Generate START condition
6     i2c->CR1 |= I2C_CR1_START;
7     // 3. Wait for EV5 (START sent)
8     while (!(i2c->SR1 & I2C_SR1_SB)) { }
9     // 4. Send slave address (write mode)
10    i2c->DR = address << 1; // Address + Write bit (0)
11    // 5. Wait for EV6 (address sent)
12    while (!(i2c->SR1 & I2C_SR1_ADDR)) { }
13    // 6. Clear ADDR by reading SR2
14    (void)i2c->SR2;
15    // 7. Send data bytes
16    for (uint16_t i = 0; i < size; i++) {
17        // Wait until TXE is set
18        while (!(i2c->SR1 & I2C_SR1_TXE)) { }
19
20        // Send data byte
21        i2c->DR = data[i];
22    }
23    // 8. Wait for transfer to complete
24    while (!(i2c->SR1 & I2C_SR1_TXE) || !(i2c->SR1 & I2C_SR1_BTF)) { }
25    // 9. Generate STOP condition
26    i2c->CR1 |= I2C_CR1_STOP;
27 }

```

## Register Bits

### I2C CR1 I<sup>2</sup>C control register 1

SWRST	Software reset
ALERT	SMBus alert
PEC	Packet error checking
POS	Acknowledge / PEC Position
ACK	Acknowledge enable
STOP	Stop generation
START	Start generation
NOSTRETCH	Clock stretching disable (slave)
ENGC	General call enable
ENPEC	PEC enable
ENARP	ARP enable
SMBTYP	SMbus type (device vs. host)
SMBUS	SMbus mode (I2C vs. SMbus)
PE	Peripheral enable

### I2C OAR1 I<sup>2</sup>C own address register 1

ADDMODE	Addressing mode (7-bit vs. 10-bit)
ADD[9:8]	Interface address
ADD[7:1]	Interface address
ADD0	Interface address

### I2C SR2 I<sup>2</sup>C status register 2

PEC[7:0]	Packet error checking register
DUALF	Dual flag (slave)
SMBHOST	SMBus host header (slave)
SMBDEFAULT	SMBus device default address (slave)
GENCALL	General call address (slave)
TRA	Transmitter/receiver (R/W bit)
BUSY	Bus busy (communication ongoing on bus)
MSL	Master/slave

### I2C OAR2 I<sup>2</sup>C own address register 2

ADD2[7:1]	Interface address in dual adr. mode
ENDUAL	Dual addressing mode enable

### I2C SR1 I<sup>2</sup>C status register 1

SMBALERT	SMBus alert
TIMEOUT	Timeout or Tlow error
PECERR	PEC error in reception
OVR	Overrun/Underrun
AF	Acknowledge failure
ARLO	Arbitration lost (master)
BERR	Bus error
TxE	Data register empty
RxNE	Data register not empty
STOPF	Stop detected (slave)
ADD10	10-bit header sent (master)
BTF	Byte transfer finished
ADDR	ADDR sent (master) / matched (slave)
SB	Start bit (master)

### I2C DR I<sup>2</sup>C data register 2

DR[7:0]	8-bit data register
---------	---------------------

# Timer/Counter

## Timer/Counter Basics

### Timer/Counter Fundamentals

- Timer:** Counts clock cycles or processor cycles (periodic)
- Counter:** Counts external events or signals

#### Common applications:

- Triggering periodic software tasks (e.g., display refresh)
- Sampling inputs (e.g., buttons) at regular intervals
- Counting pulses on an input pin
- Measuring time between events
- Generating pulse sequences on output pins

### Timer/Counter Structure

Basic components of a timer/counter system:

- Counter Register:** 16-bit or 32-bit counter that increments/decrements
- Prescaler:** Divides input clock to extend counting range
- Auto-Reload Register (ARR):** Sets upper limit for counting
- Source Selection:** Internal clocks, input pins, other timers
- Control Logic:** Configures counting mode and operation
- Interrupt Flags:** Signal events to the CPU

### Timer Period Calculation

For a given timer frequency, the period is calculated as:

$$T_{timer} = \frac{1}{f_{timer}}$$

Timer frequency is derived from the system clock:

$$f_{timer} = \frac{f_{system}}{(PSC + 1)}$$

Calculate the prescaler (PSC) and auto-reload (ARR) values to generate a 50ms timer period using a 84MHz clock.

We need to find PSC and ARR values to generate a 50ms (0.05s) period.

Step 1: Choose an appropriate prescaler value.  
Let's pick a prescaler to generate a 1MHz timer clock:  
 $PSC = 84\text{MHz} / 1\text{MHz} - 1 = 84 - 1 = 83$

Step 2: Calculate the auto-reload value.  
 $ARR = f_{timer} \times T_{desired} - 1$   
 $ARR = 1\text{MHz} \times 0.05\text{s} - 1$   
 $ARR = 50,000 - 1 = 49,999$

**Input Capture** Input capture is used to measure the timing of external events:

- Captures the timer counter value when an event occurs on an input pin
- Events can be rising edge, falling edge, or both
- Useful for measuring pulse width, period, frequency, or phase difference
- Each capture channel has its own register (CCRx)

Applications:

- Measuring pulse width (e.g., from sensors)
- Measuring frequency of input signals
- Timing between events

### Counter Modes

#### Up-counting mode:

- Counter starts from 0
- Increments up to auto-reload value (ARR)
- Generates overflow event when reaching ARR
- Resets to 0 and continues

#### Down-counting mode:

- Counter starts from auto-reload value (ARR)
- Decrements down to 0
- Generates underflow event when reaching 0
- Reloads ARR value and continues

#### Center-aligned mode:

- Counter counts from 0 to ARR-1, then back to 1
- Generates events at both up and down counting
- Useful for symmetric PWM generation

For a desired output frequency:

$$ARR = \frac{f_{timer}}{f_{desired}} - 1$$

For a desired time period:

$$ARR = f_{timer} \times T_{desired} - 1$$

Calculate the prescaler (PSC) and auto-reload (ARR) values to generate a 50ms timer period using a 84MHz clock.

However, this exceeds the 16-bit range (maximum 65,535).

Let's adjust to use a 10kHz timer clock:  
 $PSC = 84\text{MHz} / 10\text{kHz} - 1$   
 $= 8,400 - 1 = 8,399$

$$ARR = 10\text{kHz} \times 0.05\text{s} - 1 = 500 - 1 = 499$$

Therefore:  
 $PSC = 8,399$ ,  $ARR = 499$

## PWM (Pulse Width Modulation)

### PWM Basics

Pulse Width Modulation is a technique to create analog-like signals using digital outputs:

- Signal alternates between high and low state
- Fixed frequency (period)
- Variable duty cycle (ratio of high time to period)
- The average voltage is proportional to duty cycle

Applications:

- LED dimming
- Motor speed control
- Digital-to-analog conversion
- Signal generation

### PWM Generation Using Output Compare

Implement PWM using output compare function:

- Counter continuously runs through defined range (0 to ARR)
- Output pin toggles when counter matches the capture/compare value (CCR)
- In up-counting PWM mode 1:
  - Output active when counter  $< CCR$
  - Output inactive when counter  $\geq CCR$
- Duty cycle controlled by changing CCR value

### PWM Terminology

- Period (T):** Time for one complete cycle
- Frequency (f):** Number of cycles per second ( $f = 1/T$ )
- Duty Cycle (D):** Ratio of high time to period ( $D = T_{high}/T$ )
- Resolution:** min. change in duty cycle possible

### PWM Alignment Types

#### Edge-aligned PWM:

- One edge of the pulse is fixed, the other is modulated
- Up-counting mode: Leading edge fixed, trailing edge modulated
- Down-counting mode: Trailing edge fixed, leading edge modulated

#### Center-aligned PWM:

- Center of pulse is fixed, both edges are modulated
- Produces less harmonic distortion in motor control applications
- Implemented using up/down counting mode

### PWM Duty Cycle Calculation

#### Up-counting mode:

$$\text{Duty Cycle} = \frac{CCR}{ARR + 1} \times 100\%$$

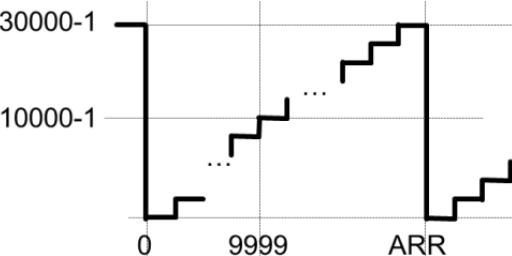
#### Down-counting mode:

$$\text{Duty Cycle} = \left(1 - \frac{CCR}{ARR + 1}\right) \times 100\%$$

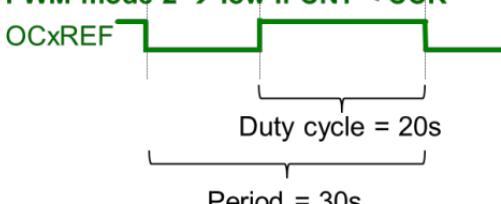
To achieve a specific duty cycle:

$$CCR (\text{up-counting}) = (ARR + 1) \times \frac{\text{Duty Cycle}}{100\%}$$

$$CCR (\text{down-counting}) = (ARR+1) \times (1 - \frac{\text{Duty Cycle}}{100\%})$$



**PWM mode 2 → low if CNT < CCR**



Bei PWM-Signalen ist es wichtig zu verstehen, dass die ARR den gesamten Zählbereich definiert (und damit die Periode), während CCR den Umschaltpunkt definiert (und damit den Duty Cycle). Die PWM-Modi bestimmen, wann das Signal High bzw. Low ist.

# Timer Capture/Compare Timing Analysis

## Timer-Komponenten verstehen

### Prescaler

- Funktionalität: Divisor für Eingangssignal
- Zweck: Es wird nur jeder n-te Wert gezählt
- Erweitert den Zählbereich des Timers

### Counter

- Funktionalität: Aktueller Timerwert
- Zweck: Wird mit jedem n-ten Tick um eins erhöht/erniedrigt
- Kann up- oder down-counting sein

### Auto Reload Register (ARR)

- Upcounter: Timer zählt bis zu diesem Wert, dann Überlauf
- Downcounter: Startwert für Timer
- Definiert die Periode des Timers

### Capture/Compare Register (CCR)

- Capture: Bei einem Event wird Counterwert hier gespeichert
- Compare: Event/Interrupt wird ausgelöst, wenn Counter diesen Wert erreicht
- Für PWM: Definiert Duty Cycle

## Timer-Funktionen

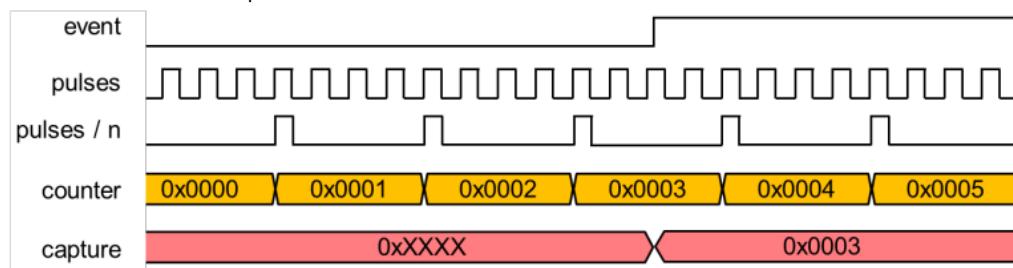
Timer als Upcounter, Prescaler auf 4 (Register = 0x03), Capture bei steigender Flanke.

**Capture-Funktion:** Bei einem Event wird der Inhalt des Counter Registers in das Capture/Compare Register kopiert. Der Counter läuft weiter.

**Compare-Funktion:** Sobald der Counter den Wert des Capture/Compare Registers erreicht hat, wird ein Event oder Interrupt ausgelöst. Der Counter läuft weiter.

### Timing-Diagramm:

- Prescaler teilt durch 4: nur jeder 4. Puls wird gezählt
- Bei Event wird aktueller Counter-Wert (z.B. 0x0003) in CCR gespeichert
- Counter läuft nach Capture weiter



## Analyzing Timer Signal Generation

### Understand the timing diagram

- Identify the timer mode (up-counting, down-counting)
- Note the prescaler value and input frequency
- Observe the event signals and their timing

### Trace the counter value

- Start with initial counter value
- For up-counting: Increment counter for each clock tick after prescaler
- For down-counting: Decrement counter for each clock tick after prescaler
- Reset counter when overflow/underflow occurs
- Note when capture events occur

### Analyze capture/compare events

- For capture mode: Counter value is stored in CCRx when event occurs
- For compare mode: Compare event occurs when CNT = CCRx
- Note the timing of interrupt flags (CCIF)

### Calculate timing parameters

- Actual period =  $(ARR+1) \times (PSC+1) / \text{timer\_clock}$
- Duty cycle =  $\text{time\_active} / \text{period} \times 100\%$
- For PWM: Duty cycle =  $CCRx / (ARR+1) \times 100\%$  (up-counting mode)

## Timer Signal Analysis Example

### Analyze the following timer configuration:

- Timer is configured as up-counter
- Source frequency is 0.5 MHz
- Prescaler (PSC) = 0x01F3 (499)
- Auto-reload (ARR) = 0x752F (30000-1)
- Compare value (CCR1) = 0x2710 (10000)
- CCMR1 = 0x0070 (PWM Mode 2)

Determine the period and duty cycle of the generated PWM signal.

### Solution:

First, calculate the effective counter frequency:

- Timer input = 0.5 MHz
- Prescaler =  $499+1 = 500$
- Counter frequency =  $0.5 \text{ MHz} / 500 = 1 \text{ kHz}$

### Calculate the period:

- ARR = 30000-1 = 29999
- Period =  $(ARR+1) / \text{counter frequency} = 30000 / 1000 \text{ Hz} = 30 \text{ seconds}$

### Determine the duty cycle:

- PWM Mode 2 means output is active when  $CNT > CCR1$
- In up-counting mode, this means output is active when  $10000 < CNT \leq 29999$
- Active time =  $(29999+1 - 10000) / 1000 \text{ Hz} = 20 \text{ seconds}$
- Duty cycle = Active time / Period =  $20 \text{ s} / 30 \text{ s} = 66.67\%$

Therefore, the PWM signal has:

- Period: 30 seconds
- Duty cycle: 66.67% (active for 20 seconds, inactive for 10 seconds)

# STM32F4 Timers/Counters

## STM32F4 Timer Types

### Basic Timers (TIM6, TIM7):

- 16-bit counter, prescaler, auto-reload
- Simplest timers, mainly for time-base generation

### General-Purpose Timers (TIM2-TIM5, TIM9-TIM14):

- TIM2, TIM5: 32-bit timers
- TIM3, TIM4: 16-bit timers
- Multiple capture/compare channels
- Input capture, output compare, PWM generation

### Advanced-Control Timers (TIM1, TIM8):

- Advanced PWM features
- Complementary outputs with dead-time insertion
- Break input for motor control safety

## Timer Clock Sources

- Internal Clock (CK\_INT): Default source, derived from system clock
- External Clock Mode 1: Timer clock from external pins (TIMx\_CH1, TIMx\_CH2)
- External Clock Mode 2: Timer clock from external trigger input (TIMx\_ETR)
- Internal Trigger Inputs (ITRx): Using one timer as prescaler for another

## Timer Configuration

### Understanding Timer Configuration

- Prescaler (PSC): Divides input clock frequency by (PSC+1)
- Counter (CNT): Current count value
- Auto-reload register (ARR): Value to reload after overflow/underflow
- Update flag (UIF): Set when counter overflows/underflows

#### Calculate timer parameters

- Determine desired period (T): The time between events
- Calculate required counts: counts =  $T \times \text{input\_frequency}$
- If counts > 65,536 (16-bit limit), use prescaler:
  - Choose prescaler value (PSC):  $PSC = PSC = (\text{Clock\_Freq} / \text{gewünschte\_Freq}) - 1$
  - Adjusted counter value:  $ARR = ARR = (\text{gewünschte\_Freq} / \text{overflow\_Freq}) - 1$
- For precision, minimize prescaler while keeping ARR within limits

## Timer konfigurieren

### Clock-Source auswählen

- TIMx\_SMCR Register: SMS[2:0] = 000 für interne Clock (CK\_INT)
- Andere Quellen: externe Pins, andere Timer (siehe Reference Manual)

### Register setzen

- TIMx\_PSC: Prescaler-Wert
- TIMx\_ARR: Auto-Reload-Wert
- TIMx\_CR1: Timer-Konfiguration (Up/Down)
- Enable interrupt if needed in TIMx\_DIER (UIE bit)
- Enable timer by setting CEN bit in TIMx\_CR1

### Zählrichtung festlegen:

- TIMx\_CR1 Register: DIR Bit
- DIR = 0: Upcounter
- DIR = 1: Downcounter

### Overflow-Zeit berechnen:

- Gewünschte Zeit und Clock-Frequenz gegeben
- Mehrere Kombinationen möglich!

## Key Timer Registers

Important registers for timer configuration:

- **TIMx\_CR1** (Control Register 1):
  - CEN: Counter Enable
  - DIR: Direction (0=up, 1=down)
  - CMS: Center-aligned Mode Selection
- **TIMx\_PSC** (Prescaler):
  - Divides clock frequency by a factor between 1 and 65536
  - Actual division factor is  $PSC + 1$
- **TIMx\_ARR** (Auto-reload Register):
  - Sets period in up-counting mode
  - Sets initial value in down-counting mode
- **TIMx\_CNT** (Counter):
  - Current counter value
- **TIMx\_SR** (Status Register):
  - UIF: Update Interrupt Flag
  - CCxIF: Capture/Compare Interrupt Flags
- **TIMx\_DIER** (DMA/Interrupt Enable Register):
  - UIE: Update Interrupt Enable
  - CCxIE: Capture/Compare Interrupt Enable

Basic Timer Configuration to generate an interrupt every 1 second (timer clock frequency is 84 MHz)

### Calculations and Setup

First, calculate the required counts: counts =  $1\text{s} \times 84\text{MHz} = 84,000,000$

Since  $84,000,000 > 65,536$  (16-bit limit), we need a prescaler:

- Choose prescaler:  $PSC = 8400 - 1 = 8399$
- This divides the clock to  $84\text{MHz}/8400 = 10\text{kHz}$
- Adjusted counter value:  $ARR = 10,000 - 1 = 9999$

### Configuration Steps

**Step 1: Enable timer clock** - Enable the clock to the timer peripheral using RCC register.

**Step 2: Configure prescaler** - Set the prescaler value to divide the timer clock.

**Step 3: Configure auto-reload value** - Set period in up-counting mode or initial value in down-counting mode

**Step 4: Configure counting mode** - Select up-counting, down-counting, or center-aligned mode.

**Step 5: Configure interrupts (if needed)** - Enable update or capture/compare interrupts.

**Step 6: Enable the timer** - Set the CEN bit in the CR1 register.

```

1 // Configure TIM2 for a 1Hz interrupt (1s period)
2 // Assuming system clock = 84MHz
3
4 // Step 1: Enable TIM2 clock
5 RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
6
7 // Step 2: Configure prescaler
8 // PSC = (SystemCoreClock / 2) / 10000 - 1
9 TIM2->PSC = 8399; // Produces 10kHz timer clock (84MHz/8400)
10
11 // Step 3: Configure auto-reload value
12 // ARR = (timer clock / desired frequency) - 1
13 TIM2->ARR = 9999; // 10kHz/10000 = 1Hz
14
15 // Step 4: Configure counting mode (up-counting, default)
16 TIM2->CR1 &= ~TIM_CR1_DIR; // Up-counting mode
17
18 // Step 5: Enable update interrupt
19 TIM2->DIER |= TIM_DIER_UIE;
20 // Step 6: Enable timer
21 TIM2->CR1 |= TIM_CR1_CEN;
22
23 // Configure NVIC to handle TIM2 interrupt
24 NVIC_EnableIRQ(TIM2_IRQn);
25 NVIC_SetPriority(TIM2_IRQn, 1);

```

- Timer clock frequency: 84MHz
- Prescaler: 8400 (gives 10kHz)
- Auto-reload: 10,000 (gives 1Hz)
- Timer period: 1 second

## Timer für 200ms Overflow Timer konfigurieren:

- Source: CK\_INT mit 84 MHz
- Upcounter
- Overflow alle 200 ms

### Lösung 1:

```

1 TIM3_SMCR &= 0xFFFF8; // SMS[2:0] = 000 (CK_INT)
2 TIM3_CR1 &= 0xFF8F; // DIR = 0 (Upcounter)
3 TIM3_PSC = 0x20CF; // (8400-1) -> 10 kHz
4 TIM3_ARR = 0x07CF; // (2000-1) -> 200 ms

```

### Lösung 2:

```

1 TIM3_PSC = 0x0347; // (840-1) -> 100 kHz
2 TIM3_ARR = 0x4E1F; // (20000-1) -> 200 ms

```

Berechnung:  $84\text{MHz} / 8400 = 10\text{kHz}$ ,  $10\text{kHz} / 2000 = 5\text{Hz} = 200\text{ms}$  Periode

## Input Capture

### Input Capture Configuration

#### Step 1: Configure GPIO pin

Configure the GPIO pin as alternate function for the timer channel.

#### Step 2: Configure timer base

Set up the timer's prescaler and period.

#### Step 3: Configure capture channel

Configure the channel for input capture and set the edge sensitivity.

#### Step 4: Enable interrupts

Enable capture interrupt and NVIC if notification is required.

#### Step 5: Enable timer

Start the timer counting.

```
1 // Configure TIM3 Channel 1 for input capture on rising edge
2
3 // Step 1: Configure GPIO pin (PA6 for TIM3_CH1)
4 RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
5 GPIOA->MODER &= ~GPIO_MODEER6;
6 GPIOA->MODER |= GPIO_MODEER6_1; // Alternate function
7 GPIOA->AFR[0] &= ~GPIO_AFRL_AFRL6;
8 GPIOA->AFR[0] |= 2 << GPIO_AFRL_AFRL6_Pos; // AF2 for TIM3
9
10 // Step 2: Configure timer base
11 RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;
12 TIM3->PSC = 83; // 84MHz/84 = 1MHz timer clock
13 TIM3->ARR = 65535; // Maximum period
14
15 // Step 3: Configure capture channel
16 // CC1S[1:0] = 01 (input, IC1 mapped to TI1)
17 TIM3->CCMR1 &= ~TIM_CCMR1_CC1S;
18 TIM3->CCMR1 |= TIM_CCMR1_CC1S_0;
19
20 // Configure input filter and prescaler if needed
21 TIM3->CCMR1 &= ~(TIM_CCMR1_IC1F | TIM_CCMR1_IC1PSC);
22
23 // Configure edge sensitivity (rising edge)
24 TIM3->CCER &= ~(TIM_CCER_CC1P | TIM_CCER_CC1NP);
25
26 // Enable capture
27 TIM3->CCER |= TIM_CCER_CC1E;
28
29 // Step 4: Enable capture interrupt
30 TIM3->DIER |= TIM_DIER_CC1IE;
31 NVIC_EnableIRQ(TIM3_IRQn);
32
33 // Step 5: Enable timer
34 TIM3->CR1 |= TIM_CR1_CEN;
```

## PWM Output Configuration

### Understanding PWM Output Configuration

Understand PWM parameters

- **Period:** Controlled by ARR value
- **Duty cycle:** Controlled by CCRx value
- **Frequency:** Determined by timer clock, prescaler, and ARR

Implementation:

- PWM Mode 1 (Upcounting): OCxREF = '1' solange TIMx\_CNT < TIMx\_CCR
- PWM Mode 2 (Upcounting): OCxREF = '0' solange TIMx\_CNT < TIMx\_CCR
- Downcounting: Logik ist invertiert

Calculate PWM parameters

- PWM frequency = Timer clock / ((PSC+1) × (ARR+1))
- For up-counting mode:
  - CCRx controls when output switches from active to inactive
  - Duty cycle = CCRx / (ARR+1) × 100%
- For down-counting mode:
  - CCRx controls when output switches from inactive to active
  - Duty cycle = (ARR+1-CCRx) / (ARR+1) × 100%

### Periode vs. Duty Cycle

- Periode wird durch ARR bestimmt
- Duty Cycle wird durch CCR bestimmt
- Frequenz = Timer\_Clock / ((PSC + 1) × (ARR + 1))

Configure PWM output

- Configure timer basic parameters (PSC, ARR)
- Select PWM mode in CCMRx register:
  - PWM Mode 1: Output active when CNT < CCRx
  - PWM Mode 2: Output active when CNT > CCRx
- Set CCRx value to control duty cycle
- Enable output in CCER register (CCxE bit)
- Configure GPIO pin for alternate function (timer output)
- Enable timer by setting CEN bit in CR1

### Register konfigurieren:

- TIMx\_CCMR: OCxM[2:0] = 110 (PWM Mode 1) oder 111 (PWM Mode 2)
- TIMx\_CCER: CCxE = 1 (Output Enable)
- TIMx\_CCR: Duty Cycle Wert

### PWM-Signal aus Registerwerten analysieren

Gegebene Konfiguration:

- Source: 0.5 MHz
- TIM3\_PSC = 0x01F3 (500-1)
- TIM3\_ARR = 0x752F (30000-1)
- TIM3\_CCR1 = 0x2710 (10000)
- TIM3\_CCMR1 = 0x0070 (PWM Mode 2)

### Analyse:

- Prescaler: 500-1 → 0.5 MHz / 500 = 1 kHz
- ARR: 30000-1 → Periode  
= 30000 / 1 kHz = 30 Sekunden
- CCR: 10000 → 10 Sekunden
- PWM Mode 2: Low wenn CNT < CCR

### Resultat:

- Periode: 30 Sekunden
- Duty Cycle: 20 Sekunden High (30s - 10s)
- 0...9999: Low (10s), 10000...29999: High (20s)

## PWM Configuration

### Step 1: Configure GPIO pin

Configure the GPIO pin as alternate function for the timer channel.

### Step 2: Configure timer base

Set prescaler and auto-reload value to define PWM frequency.

### Step 3: Configure PWM mode

Set output compare mode to PWM and configure polarity.

### Step 4: Set initial duty cycle

Write the initial CCR value.

### Step 5: Enable output and timer

Enable output and start the timer.

## PWM Configuration PWM output at 1kHz with 50% duty cycle

```
1 // Step 1: Configure GPIO pin (PA6 for TIM3_CH1)
2 RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
3 GPIOA->MODER &= ~GPIO_MODER_MODER6;
4 GPIOA->MODER |= GPIO_MODER_MODER6_1; // Alternate function
5 GPIOA->AFR[0] &= ~GPIO_AFRL_AFRL6;
6 GPIOA->AFR[0] |= 2 << GPIO_AFRL_AFRL6_Pos; // AF2 for TIM3
7
8 // Step 2: Configure timer base
9 RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;
10 TIM3->PSC = 83; // 84MHz/84 = 1MHz timer clock
11 TIM3->ARR = 999; // 1MHz/1000 = 1kHz PWM frequency
12
13 // Step 3: Configure PWM mode
14 // CC1S = 00: Channel configured as output
15 TIM3->CCMR1 &= ~TIM_CCMR1_CC1S;
16
17 // OC1M = 110: PWM mode 1 (active when counter < CCR1)
18 TIM3->CCMR1 &= ~TIM_CCMR1_OC1M;
19 TIM3->CCMR1 |= TIM_CCMR1_OC1M_1 | TIM_CCMR1_OC1M_2;
20
21 // Enable output compare preload
22 TIM3->CCMR1 |= TIM_CCMR1_OC1PE;
23
24 // Set active high polarity
25 TIM3->CCER &= ~TIM_CCER_CC1P;
26
27 // Step 4: Set initial duty cycle (50%)
28 TIM3->CCR1 = 500; // 50% of 1000
29
30 // Step 5: Enable output and timer
31 TIM3->CCER |= TIM_CCER_CC1E; // Enable output
32 TIM3->CR1 |= TIM_CR1_CEN; // Enable counter
```

**PWM Configuration** Configure Timer 4 to generate a 20kHz PWM signal with 75% duty cycle on Channel 2, system clock = 84MHz:

### 1. Calculate timer settings for 20kHz:

- Let's use prescaler = 3 (divide by 4)
- Timer clock = 84MHz/4 = 21MHz
- For 20kHz: ARR = 21MHz/20kHz - 1 = 1049

### 2. Calculate CCR value for 75% duty cycle:

- Up-counting mode, PWM mode 1
- CCR = (ARR+1) \* 0.75 = 1050 \* 0.75 = 787

### 3. Configuration code:

```
1 // Configure GPIO pin (PB7 for TIM4\_CH2)
2 RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;
3 GPIOB->MODER &= ~GPIO_MODER_MODER7;
4 GPIOB->MODER |= GPIO_MODER_MODER7_1; // Alternate function
5 GPIOB->AFR[0] &= ~GPIO_AFRL_AFRL7;
6 GPIOB->AFR[0] |= 2 << GPIO_AFRL_AFRL7_Pos; // AF2 for TIM4
7 // Configure timer base
8 RCC->APB1ENR |= RCC_APB1ENR_TIM4EN;
9 TIM4->PSC = 3; // 84MHz/4 = 21MHz
10 TIM4->ARR = 1049; // 21MHz/1050 = 20kHz
11 // Configure PWM mode
12 TIM4->CCMR1 &= ~TIM_CCMR1_CC2S; // Channel as output
13 TIM4->CCMR1 &= ~TIM_CCMR1_OC2M;
14 TIM4->CCMR1 |= TIM_CCMR1_OC2M_1 | TIM_CCMR1_OC2M_2; // PWM mode 1
15 TIM4->CCMR1 |= TIM_CCMR1_OC2PE; // Enable preload
16 TIM4->CCER &= ~TIM_CCER_CC2P; // Active high
17 // Set duty cycle (75%)
18 TIM4->CCR2 = 787;
19 // Enable output and timer
20 TIM4->CCER |= TIM_CCER_CC2E;
21 TIM4->CR1 |= TIM_CR1_CEN;
```

**PWM Configuration** Configure Timer 4 for PWM output with: ARR = 0x9C3F (value 39999), Up-counting mode, PWM Mode 1 and 25% duty cycle

Then, reconfigure for down-counting mode with PWM Mode 2 while maintaining the same electrical signal.

Solution:

For up-counting mode with PWM Mode 1:

- Total count is 40,000 (ARR+1)
- For 25% duty cycle, the output should be high for 10,000 counts
- In PWM Mode 1 (up-counting): CCR1 = 25% of 40,000 = 10,000 = 0x2710

For down-counting mode with PWM Mode 2:

- In PWM Mode 2 (down-counting), output is active when CNT > CCR1
- To maintain 25% duty cycle: CCR1 = ARR+1-10,000 = 40,000-10,000 = 30,000 = 0x752F

C code for configuration:

```
1 // Up-counting mode, PWM Mode 1, 25% duty cycle
2 TIM4->ARR = 0x9C3F; // Auto-reload value (39999)
3 TIM4->CCR1 = 0x2710; // Compare value for 25% duty cycle (10000)
4 TIM4->CCMR1 = 0x0060; // PWM Mode 1 for channel 1
5 TIM4->CCER |= 0x0001; // Enable output for channel 1
6 TIM4->CR1 &= ~TIM_CR1_DIR; // Up-counting mode
7 TIM4->CR1 |= TIM_CR1_CEN; // Enable timer
8
9 // Down-counting mode, PWM Mode 2, 25% duty cycle
10 TIM4->ARR = 0x9C3F; // Auto-reload value (39999)
11 TIM4->CCR1 = 0x752F; // Compare value for 25% duty cycle (30000)
12 TIM4->CCMR1 = 0x0070; // PWM Mode 2 for channel 1
13 TIM4->CCER |= 0x0001; // Enable output for channel 1
14 TIM4->CR1 |= TIM_CR1_DIR; // Down-counting mode
15 TIM4->CR1 |= TIM_CR1_CEN; // Enable timer
```

# Analog-to-Digital Converter (ADC)

## ADC Fundamentals

### ADC Overview

An Analog-to-Digital Converter (ADC) converts continuous analog signals into discrete digital values:

- Transforms analog voltage levels into corresponding digital values
- Resolution determined by number of bits (N)
- $2^N$  possible digital values (e.g., 12-bit ADC has 4096 levels)
- Converts real-world continuous signals (temperature, pressure, etc.) into digital form for processing

### ADC Terminology

Key terms and concepts:

- Resolution:** Number of bits (N) in the digital output
- LSB** (Least Significant Bit): Smallest detectable voltage change  
–  $LSB = \frac{V_{REF}}{2^N}$
- FSR** (Full Scale Range): Range between minimum and maximum digital codes  
–  $FSR = V_{REF} - 1LSB$
- Sampling Rate:** Number of conversions per second
- Conversion Time:** Time from start of sampling to digital output availability
- Reference Voltage ( $V_{REF}$ ):** Voltage that defines the ADC's full-scale range

### ADC Operating Principle

#### Flash ADC (Parallel ADC):

- Uses comparators for each voltage level
- Input voltage compared to reference voltages simultaneously
- Fast but requires many components (e.g., 255 comparators for 8-bit)

#### Successive Approximation Register (SAR) ADC:

- Uses binary search algorithm
- Compares input to successive approximations
- Takes N steps for N-bit resolution
- Good balance of speed, power, and complexity
- Most common in microcontrollers

## ADC Errors and Characteristics

### Sampling Theorem

According to the Nyquist-Shannon sampling theorem:

- Sampling rate must be at least twice the highest frequency component of the input signal
- $f_{sampling} \geq 2 \times f_{max}$
- Prevents aliasing (false lower frequencies appearing in sampled signal)

### ADC Error Types

#### Quantization Error:

- Inherent error due to rounding analog values to discrete digital levels
- Range between -0.5 LSB and +0.5 LSB
- Cannot be eliminated, but reduced by increasing resolution

#### Offset Error:

- Deviation from ideal ADC at zero input
- For ideal ADC, first transition occurs at 0.5 LSB
- Can be corrected through calibration

#### Gain Error:

- Difference in slope between actual and ideal transfer function
- Expressed in LSB or as percentage of full-scale range (%FSR)
- Can be corrected through calibration

### ADC Calculations

#### LSB Voltage:

$$V_{LSB} = \frac{V_{REF}}{2^N} \quad (1)$$

#### Digital Output Value:

$$D_{out} = \frac{V_{in} \times 2^N}{V_{REF}} \quad (2)$$

#### Analog Input from Digital Value:

$$V_{in} = \frac{D_{out} \times V_{REF}}{2^N} \quad (3)$$

#### Percent Full Scale Range:

$$\%FSR = \frac{V_{in}}{V_{REF}} \times 100\% \quad (4)$$

### Analyzing ADC Errors

#### Understand error types

- Quantization error:** Inherent error between -0.5 LSB and +0.5 LSB
- Offset error:** Deviation from ideal transfer function at zero input
- Gain error:** Deviation of the slope from ideal transfer function
- Full-scale error:** Combination of offset and gain error at maximum input

#### Determine ADC resolution

- Calculate LSB size:  $LSB = V_{REF} / 2^N$  where N is the number of bits
- FSR (Full Scale Range) =  $V_{REF} - 1$  LSB

#### Draw ideal transfer function

- Plot digital output vs. analog input
- Idealer ADC: erste Transition bei 0.5 LSB
- Each subsequent transition at  $(i + 0.5) \times LSB$
- For N-bit ADC:  $2^N - 1$  total steps

#### Offset-Fehler

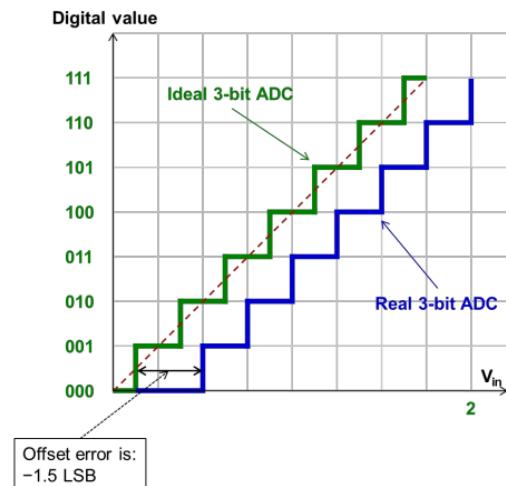
- Definition: Abweichung bei Eingangsspannung = 0V
- Messung: Zero-Scale-Spannung bis erste Transition
- Berechnung:  $Offset\_Error = gemessene\_Transition - 0.5 \text{ LSB}$

#### Calculate error values

- Offset error in LSB:** Measure deviation at zero input
- Offset error in volts:** Multiply LSB by offset error in LSB
- Gain error in LSB:** Difference between actual and ideal slope
- Error as percentage of FSR (%FSR):**  $(Error \text{ in } \text{volts} / FSR \text{ in } \text{volts}) \times 100\%$

#### Transfer-Funktion zeichnen

1. Verschiebe ideale Funktion um Offset-Fehler (positiv = rechts, negativ = links)
2. Erste Transition:  $0.5 \text{ LSB} + Offset\_Error$



#### 1. Ideal transfer function:

- N = 3 bits → 8 possible output codes (000 to 111)
- LSB = 2V / 2<sup>3</sup> = 2V / 8 = 0.25V
- First transition: 0.5 × LSB = 0.125V
- Code transitions at: 0.125V, 0.375V, 0.625V, 0.875V, 1.125V, 1.375V, 1.625V

#### 2. Actual transfer function with offset error:

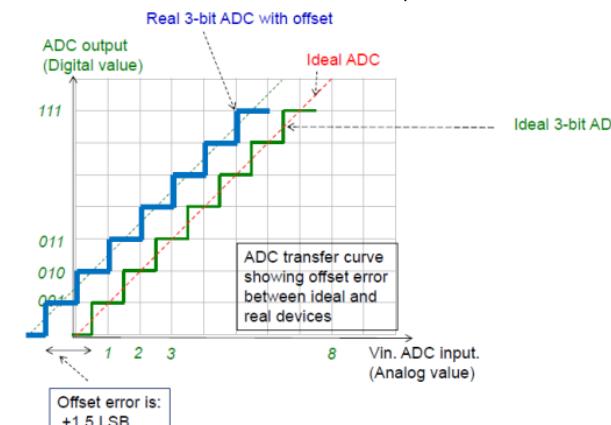
- Offset error = -1.5 LSB (entire transfer function shifts left)
- Offset Error in volts: all transitions shift by  $-1.5 \times 0.25V = -0.375V$  (Offset error × LSB)
- New transitions at: -0.25V, 0V, 0.25V, 0.5V, 0.75V, 1V, 1.25V
- Full Scale Range (FSR) = 2V - 0.25V = 1.75V
- Offset-Fehler (%FSR) =  $0.375V \times 100 / 1.75V = 21.43\%$

#### 3. Zusammenfassung:

- Ideale erste Transition: 0.5 LSB = 0.125V
- Reale erste Transition: 0.125V - 0.375V = -0.25V
- Gesamte Kurve um 1.5 LSB nach links verschoben

**Digitale Werte:** Real-ADC erreicht digitalen Wert 001 bereits bei negativer Eingangsspannung, während ideal-ADC erst bei 0.125V den Wert 001 erreichen würde.

**ADC Positive Offset Error** In case of a positive offset error of +1.5 LSB, the graph will look like this:



## ADC on STM32F4

### STM32F4 ADC Architecture

The STM32F4 includes ADC modules with the following features:

- Three ADCs (ADC1, ADC2, ADC3)
- 12-bit resolution (configurable to 10, 8, or 6 bits)
- Up to 24 external channels (16 on each ADC)
- Internal channels: temperature sensor, V<sub>REFINT</sub>, V<sub>BAT</sub>

Multiple operating modes:

- Single conversion vs. continuous conversion
- Single channel vs. scan mode (multiple channels)

Sampling and conversion features:

- Maximum sampling rate up to 2.4 MSPS (million samples per second)
- DMA capability and configurable sampling time
- Analog watchdog for threshold monitoring

### ADC Registers

Key ADC registers on STM32F4:

- **ADC\_SR:** Status register (flags for EOC, overrun, etc.)
- **ADC\_CR1:** Control register 1 (scan mode, resolution, etc.)
- **ADC\_CR2:** Control register 2 (conversion start, data alignment, etc.)
- **ADC\_SMPRx:** Sample time registers
- **ADC\_SQRx:** Regular sequence registers
- **ADC\_DR:** Data register (conversion result)
- **ADC\_CCR:** Common control register (for all ADCs)

**SEP Handout Beilage ADC: addresses and configurations for ADC registers, Data Alignment**

### Basic ADC Configuration

Step 1: Enable GPIO and ADC clocks

Enable the clock to the GPIO port and ADC.

Step 2: Configure GPIO pins

Configure the GPIO pins as analog inputs.

Step 3: Configure ADC parameters

Set resolution, scan mode, conversion mode, and alignment.

Step 4: Configure channel and sampling time

Set the channel sequence and sampling time.

Step 5: Enable ADC

Turn on the ADC.

```

1 // Configure ADC1 Channel 0 (PA0) for single conversion
2
3 // Step 1: Enable GPIO and ADC clocks
4 RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; // Enable GPIOA clock
5 RCC->APB2ENR |= RCC_APB2ENR_ADC1EN; // Enable ADC1 clock
6 // Step 2: Configure PA0 as analog input
7 GPIOA->MODER |= GPIO_MODEER_MODERO; // Analog mode (0b11)
8 // Step 3: Configure ADC parameters
9 // ADC Common Control Register
10 ADC->CCR &= ~ADC_CCR_ADCPRE; // ADCPRE = 0 (APB2/2, typically 42MHz/2 = 21MHz)
11
12 // ADC1 Control Register 1
13 ADC1->CR1 &= ~ADC_CR1_RES; // 12-bit resolution (default)
14 ADC1->CR1 &= ~ADC_CR1_SCAN; // Disable scan mode (single channel)
15 // ADC1 Control Register 2
16 ADC1->CR2 &= ~ADC_CR2_CONT; // Single conversion mode
17 ADC1->CR2 &= ~ADC_CR2_ALIGN; // Right alignment
18 ADC1->CR2 &= ~ADC_CR2_EXTEN; // Software trigger
19
20 // Step 4: Configure channel and sampling time
21 // Configure for channel 0
22 ADC1->SQR1 &= ~ADC_SQR1_L; // 1 conversion in regular sequence
23 ADC1->SQR3 &= ~ADC_SQR3_SQ1; // Clear channel selection
24 ADC1->SQR3 |= (0 << ADC_SQR3_SQ1_Pos); // Channel 0 as first conversion
25 // Set sampling time for channel 0 (e.g., 84 cycles)
26 ADC1->SMPR2 &= ~ADC_SMPR2_SMP0; // Clear bits
27 ADC1->SMPR2 |= (4 << ADC_SMPR2_SMP0_Pos); // 84 cycles
28
29 // Step 5: Enable ADC
30 ADC1->CR2 |= ADC_CR2_ADON; // Turn on ADC

```

### ADC Conversion Modes

#### Single vs. Continuous Conversion:

- **Single Conversion:** Performs one conversion, then stops
- **Continuous Conversion:** Continuously performs conversions without CPU intervention

#### Single Channel vs. Scan Mode:

- **Single Channel:** Converts one channel only
- **Scan Mode:** Converts multiple channels in sequence

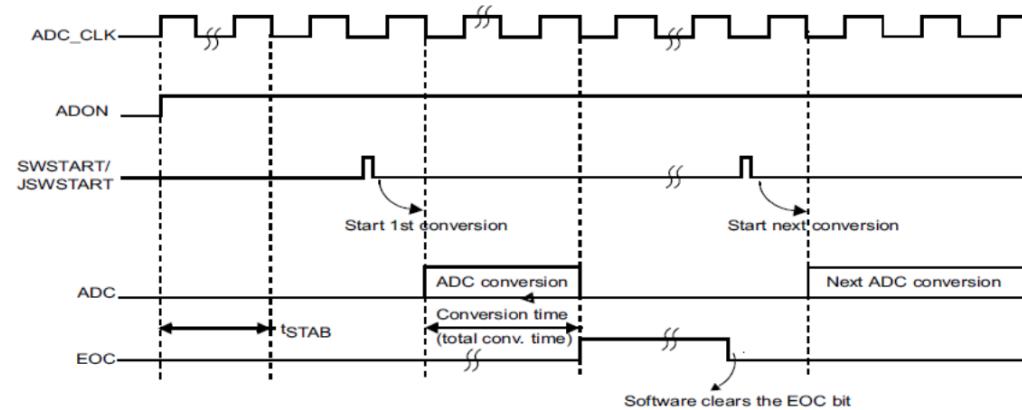
This results in four possible combinations:

- Single channel, single conversion (simplest mode)
- Single channel, continuous conversion (monitor one input)
- Multi-channel, single conversion (read multiple inputs once)
- Multi-channel, continuous conversion (monitor multiple inputs)

**Fast Conversion Mode** It is possible to perform faster conversion by reducing the ADC resolution. The RES bits are used to select the number of bits available in the data register. The minimum conversion time for each resolution is then as follows:

- 12 bits:  $3 + 12 = 15$  ADCCLK cycles
- 10 bits:  $3 + 10 = 13$  ADCCLK cycles
- 8 bits:  $3 + 8 = 11$  ADCCLK cycles
- 6 bits:  $3 + 6 = 9$  ADCCLK cycles

### ADC Timing Diagram



### Performing ADC Conversion

#### Start conversion

Trigger the conversion using software or external trigger.

#### Wait for completion

Check the EOC flag to determine when conversion is complete.

#### Read result

Read the data register to get the conversion result.

```

1 // Function to perform single ADC conversion
2 uint16_t ADC_ReadChannel(void) {
3     // Start conversion
4     ADC1->CR2 |= ADC_CR2_SWSTART; // Software trigger to start conversion
5
6     // Wait for conversion to complete
7     while (!(ADC1->SR & ADC_SR_EOC)) { }
8
9     // Read and return result
10    return ADC1->DR;
11 }
12
13 // Function to convert ADC value to voltage
14 float ADC_ConvertToVoltage(uint16_t adcValue) {
15     // Assuming VREF = 3.3V and 12-bit resolution (4096 levels)
16     float voltage = (float)adcValue * 3.3f / 4095.0f;
17     return voltage;
18 }
```

### Multi-Channel ADC Configuration

#### Configure scan mode

Enable scan mode to convert multiple channels.

#### Set up channel sequence

Configure the sequence and number of channels.

#### Process results

Read results for each channel in sequence.

```

1 // Configure ADC for multi-channel scanning (channels 0, 1, and 4)
2
3 // Enable scan mode
4 ADC1->CR1 |= ADC_CR1_SCAN;
5
6 // Set number of conversions in sequence (3)
7 ADC1->SQR1 &= ~ADC_SQR1_L;
8 ADC1->SQR1 |= (2 << ADC_SQR1_L_Pos); // L = 2 means 3 conversions
9
10 // Set channel sequence
11 ADC1->SQR3 = 0; // Clear all
12 ADC1->SQR3 |= (0 << ADC_SQR3_SQ1_Pos); // CH0 as 1st conversion
13 ADC1->SQR3 |= (1 << ADC_SQR3_SQ2_Pos); // CH1 as 2nd conversion
14 ADC1->SQR3 |= (4 << ADC_SQR3_SQ3_Pos); // CH4 as 3rd conversion
15
16 // Set sampling times for each channel
17 ADC1->SMPR2 &= ~(ADC_SMPR2_SMPO | ADC_SMPR2_SMP1 | ADC_SMPR2_SMP4);
18 ADC1->SMPR2 |= (4 << ADC_SMPR2_SMPO_Pos); // 84 cycles for CH0
19 ADC1->SMPR2 |= (4 << ADC_SMPR2_SMP1_Pos); // 84 cycles for CH1
20 ADC1->SMPR2 |= (4 << ADC_SMPR2_SMP4_Pos); // 84 cycles for CH4
```

## ADC Configuration Exercise

Configure ADC1 to measure an analog voltage on pin PA5, with 12-bit resolution. Convert the result to a voltage between 0-3.3V.

1. Configure PA5 as an analog input
2. Configure ADC1 for 12-bit resolution, single conversion, single channel
3. Set appropriate sampling time
4. Perform conversion and convert result to voltage

```
1 // Configure PA5 as analog input
2 RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; // Enable GPIOA clock
3 RCC->APB2ENR |= RCC_APB2ENR_ADC1EN; // Enable ADC1 clock
4 GPIOA->MODER |= GPIO_MODER_MODERS; // Set PA5 to analog mode (0b11)
5
6 // Configure ADC1
7 ADC1->CR1 &= ~ADC_CR1_RES; // 12-bit resolution (default)
8 ADC1->CR2 &= ~ADC_CR2_CONT; // Single conversion mode
9 ADC1->CR2 &= ~ADC_CR2_ALIGN; // Right alignment
10
11 // Configure for channel 5 (PA5)
12 ADC1->SQR1 &= ~ADC_SQR1_L; // 1 conversion in sequence
13 ADC1->SQR3 &= ~ADC_SQR3_SQ1; // Clear channel selection
14 ADC1->SQR3 |= (5 << ADC_SQR3_SQ1_Pos); // Channel 5 as first conversion
15
16 // Set sampling time (84 cycles)
17 ADC1->SMPR2 &= ~ADC_SMPR2_SMP5;
18 ADC1->SMPR2 |= (4 << ADC_SMPR2_SMP5_Pos);
19
20 // Enable ADC
21 ADC1->CR2 |= ADC_CR2_ADON;
22
23 // Function to read ADC and convert to voltage
24 float ReadVoltage(void) {
25     // Start conversion
26     ADC1->CR2 |= ADC_CR2_SWSTART;
27
28     // Wait for conversion to complete
29     while (!(ADC1->SR & ADC_SR_EOC)) { }
30
31     // Read ADC value
32     uint16_t adcValue = ADC1->DR;
33
34     // Convert to voltage (0-3.3V)
35     float voltage = (float)adcValue * 3.3f / 4095.0f;
36
37     return voltage;
38 }
```

## Analog Watchdogs

### Analog Watchdog

The Analog Watchdog monitors ADC conversion results against programmable thresholds:

- Generates an interrupt if a conversion result is outside the threshold range
- Can be configured to monitor a single channel or all channels
- Useful for detecting abnormal voltage levels without CPU polling
- Programmable high and low thresholds

Applications:

- Over-voltage/under-voltage detection
- Temperature limit monitoring
- Battery level monitoring

## Using DMA with ADC

### Configure DMA

Set up DMA channel to transfer ADC results to memory.

### Enable DMA for ADC

Configure ADC to use DMA for data transfer.

### Start conversion

Start ADC conversion in continuous mode.

```
1 // Configure ADC with DMA for continuous multi-channel sampling
2
3 // Configure DMA
4 RCC->AHB1ENR |= RCC_AHB1ENR_DMA2EN; // Enable DMA2 clock
5
6 // Configure DMA2 Stream0 for ADC1
7 DMA2_Stream0->CR &= ~DMA_SxCR_EN; // Disable DMA stream
8 while (DMA2_Stream0->CR & DMA_SxCR_EN) { } // Wait until disabled
9
10 DMA2_Stream0->CR = 0;
11 DMA2_Stream0->CR |= (0 << DMA_SxCR_CHSEL_Pos); // Channel 0
12 DMA2_Stream0->CR |= DMA_SxCR_PL_1; // Priority high
13 DMA2_Stream0->CR |= DMA_SxCR_MSIZE_0; // Memory data size: 16-bit
14 DMA2_Stream0->CR |= DMA_SxCR_PSIZE_0; // Peripheral data size: 16-bit
15 DMA2_Stream0->CR |= DMA_SxCR_MINC; // Memory increment mode
16 DMA2_Stream0->CR &= ~DMA_SxCR_PINC; // Peripheral fixed
17 DMA2_Stream0->CR |= DMA_SxCR_CIRC; // Circular mode
18
19 // Set addresses
20 DMA2_Stream0->PAR = (uint32_t)&ADC1->DR; // Source: ADC1 data register
21 DMA2_Stream0->MOAR = (uint32_t)adc_values; // Destination: buffer
22 DMA2_Stream0->NDTR = 3; // Number of data items (3 channels)
23
24 // Enable DMA stream
25 DMA2_Stream0->CR |= DMA_SxCR_EN;
26
27 // Configure ADC for DMA
28 ADC1->CR2 |= ADC_CR2_DMA; // Enable DMA mode
29 ADC1->CR2 |= ADC_CR2_CONT; // Continuous conversion mode
30
31 // Start ADC conversion
32 ADC1->CR2 |= ADC_CR2_SWSTART;
```

**ADC Interrupts** An interrupt can be produced on the end of conversion for regular and injected groups, when the analog watchdog status bit is set and when the overrun status bit is set. Separate interrupt enable bits are available for flexibility. Two other flags are present in the ADC\_SR register, but there is no interrupt associated with them:

- JSTRT (Start of conversion for channels of an injected group)
- STRT (Start of conversion for channels of a regular group)

Table 70. ADC interrupts

Interrupt event	Event flag	Enable control bit
End of conversion of a regular group	EOC	EOCIE
End of conversion of an injected group	JEOC	JEOCIE
Analog watchdog status bit is set	AWD	AWDIE
Overrun	OVR	OVRIE

# ADC EVERYTHING TOGETHER

## ADC-Register programmieren

### Status-Register auswerten

- EOC (End of Conversion): Bit 1 in ADC\_SR
- OVR (Overrun): Bit 5 in ADC\_SR
- Warten auf Conversion: while !(ADC\_SR & 0x02)
- Overrun prüfen: if (ADC\_SR & 0x20)

### Auflösung konfigurieren

- RES[1:0] in ADC\_CR1 Register (Bits 25:24)
- 00: 12-bit, 01: 10-bit, 10: 8-bit, 11: 6-bit
- Einzelne Bits setzen/löschen mit Bit-Masken

### Adressen berechnen

- ADC1: 0x40012000 + Register\_Offset
- ADC2: 0x40012100 + Register\_Offset
- ADC3: 0x40012200 + Register\_Offset
- Siehe Reference Manual für genaue Offsets

### Bit-Manipulation

- Bit setzen: REG |= (1 << bit\_nr)
- Bit löschen: REG &= ~(1 << bit\_nr)
- Mehrere Bits: erst löschen, dann setzen

Bei ADC-Registern ist es wichtig, die korrekten Bit-Positionen zu verwenden. EOC ist Bit 1 (nicht Bit 0), OVR ist Bit 5. Die Auflösung wird durch zwei Bits (RES[1:0]) in Bits 25:24 des CR1-Registers gesteuert.

## Configuring and Using STM32 ADCs

### Enable peripheral clocks

- Enable GPIO clock for analog pin
- Enable ADC clock in RCC register

### Configure GPIO pin for analog mode

- Set GPIO MODE register bits to analog mode (11)

### Configure ADC common settings

- Configure ADC clock prescaler in ADC\_CCR
- Enable/disable temperature sensor and internal reference

### Configure ADC specific settings

- Set resolution in ADC\_CR1 (RES bits)
- Configure scan mode if using multiple channels
- Set conversion mode in ADC\_CR2 (CONT bit):
  - 0: Single conversion
  - 1: Continuous conversion
- Set data alignment in ADC\_CR2 (ALIGN bit)
- Configure ADC trigger source if needed

### Configure channel settings

- Set sampling time in ADC\_SMPR1/2 registers
- Configure regular sequence in ADC\_SQR1/2/3:
  - Set sequence length in L[3:0] bits
  - Set channel order in SQx[4:0] bits

### Start conversion and read results

- Enable ADC by setting ADON bit in ADC\_CR2
- Start conversion by setting SWSTART bit in ADC\_CR2
- Poll EOC flag in ADC\_SR or use interrupt
- Read conversion result from ADC\_DR

**ADC Configuration and Usage** Write C code to configure and use ADC1 to read from channel 5 with 12-bit resolution in single conversion mode. The code should:

1. Wait until ADC1 conversion has completed
2. Set an 8-bit variable (var2) to 0xFF if there was a loss of data on ADC2, otherwise reset that variable to 0
3. Set ADC3 resolution to 10-bit

Use register addresses from the provided documentation.

### Solution:

First, let's define the relevant register addresses:

```
1 // ADC1 registers
2 #define ADC1_SR      (*((volatile uint32_t*)(0x40012000))) // Status register
3 #define ADC1_CR1     (*((volatile uint32_t*)(0x40012004))) // Control register 1
4 #define ADC1_CR2     (*((volatile uint32_t*)(0x40012008))) // Control register 2
5 #define ADC1_DR      (*((volatile uint32_t*)(0x4001204C))) // Data register
6 // ADC2 registers
7 #define ADC2_SR      (*((volatile uint32_t*)(0x40012100))) // Status register
8 // ADC3 registers
9 #define ADC3_CR1     (*((volatile uint32_t*)(0x40012204))) // Control register 1
10 // Bit positions
11 #define ADC_SR_EOC   (1 << 1) // End of conversion flag
12 #define ADC_SR_OVR   (1 << 5) // Overrun flag
13 #define ADC_CR1_RES_MASK (3 << 24) // Resolution mask
14 #define ADC_CR1_RES_10BIT (1 << 24) // 10-bit resolution
```

Now, let's solve each part of the problem:

1. **Wait until ADC1 conversion has completed** → wait for the EOC flag in ADC1\_SR (Bit 1)

```
1 // Wait for end of conversion flag
2 while (!(ADC1_SR & ADC_SR_EOC)) {
3     // Empty loop, wait until EOC is set
4 }
```

2. **Check for overrun on ADC2 and set var2 accordingly** → check the OVR flag in ADC2\_SR (Bit 5)

```
1 uint8_t var2;
2 if (ADC2_SR & ADC_SR_OVR) {
3     var2 = 0xFF; // Overrun occurred
4 } else {
5     var2 = 0x00; // No overrun
6 }
```

3. **Set ADC3 resolution to 10-bit** → clear the RES bits in ADC3\_CR1 and set to 10-bit

```
1 ADC3_CR1 &= ~ADC_CR1_RES_MASK; // Clear bits
2 ADC3_CR1 |= ADC_CR1_RES_10BIT; // Set 10-bit resolution
```

# Memory

## Memory Technologies Overview

### Semiconductor Memory Classifications

Semiconductor memories can be classified into two main categories:

- **Volatile Memory:** Loses data when power is turned off
  - SRAM (Static Random Access Memory)
  - DRAM (Dynamic Random Access Memory)
- **Non-volatile Memory:** Retains data even without power
  - ROM (Read-Only Memory)
  - PROM (Programmable ROM)
  - EPROM (Erasable PROM)
  - EEPROM (Electrically Erasable PROM)
  - Flash Memory
  - NV-RAM (Non-Volatile RAM)

### Memory Organization

Memory devices are organized as arrays of bit cells:

- **Array Size:**  $n \times m$  (n words with m data bits each)
- **Address Lines:** k bits can address  $2^k$  words
- **Data Lines:** Width of data bus (8, 16, 32 bits, etc.)
- **Control Lines:** Enable read/write operations

## PROM, EEPROM, and Flash Memory

### PROM (Programmable Read-Only Memory)

- One-time programmable memory
- Programming involves physically altering the circuit (e.g., blowing fuses)
- Once programmed, contents cannot be changed
- Used for permanent storage of code or data

### EEPROM (Electrically Erasable PROM)

- Uses floating-gate transistors to store data
- Can be electrically programmed and erased
- Byte-level erase and write operations
- Limited write cycles (typically 100,000 to 1,000,000)
- Slower and more expensive than SRAM
- Used for storing configuration data or parameters

### Flash Memory

- Based on floating-gate transistor technology (like EEPROM)
- Higher density and lower cost per bit than EEPROM
- Block-wise erase operations (not byte-level)
- Write operations can only change bits from '1' to '0'
- Erase operations reset all bits in a block to '1'
- Limited write/erase cycles (typically 10,000 to 100,000)
- Used for code storage and mass data storage

### NOR vs. NAND Flash

#### NOR Flash:

- Random access (like RAM)
- Execute-in-place capability (XIP)
- Fast read access
- Slow write and erase operations
- Lower density
- Used for code storage and execution

#### NAND Flash:

- Page-based access (not random)
- Cannot execute code directly
- Slow random read access
- Fast sequential read and write
- Higher density
- Used for mass storage (SSDs, memory cards)

#### NOR Flash - Anwendungen:

- Mikrocontroller-Firmware (direkte Ausführung)
- BIOS/UEFI Code
- Embedded Systems mit wenig RAM

#### NAND Flash - Anwendungen:

- Solid State Drives (SSDs)
- SD-Karten, USB-Sticks
- Smartphones (App-Speicher)

**Wichtiger Unterschied:** NOR ermöglicht wahlfreie Zugriffe wie RAM, während NAND blockweise gelesen werden muss und eine spezielle Controller-Logik benötigt.

Bei der Speicher-Auswahl ist das Anwendungsprofil entscheidend: NOR Flash für direkten Code-Zugriff, NAND Flash für große Datens Mengen, SRAM für schnelle Zwischenspeicherung, DRAM für große Arbeitsspeicher mit hoher Dichte.

## SRAM (Static RAM)

### SRAM Structure and Characteristics

- Uses flip-flop circuit for each bit (typically 6 transistors)
- Maintains data as long as power is supplied
- No refresh required (unlike DRAM)
- Fast access times (a few nanoseconds)
- Low power consumption in standby mode
- Higher cost and lower density than DRAM
- Used for cache memory and high-speed buffers

### SRAM Cell

A typical SRAM cell consists of:

- Cross-coupled inverters forming a latch to store one bit
- Two access transistors to connect the cell to bit lines
- Word line to enable/disable access to the cell
- High state ('1') and low state ('0') stable as long as power is maintained

### SRAM Operations

#### Read Operation:

- Word line is activated
- Access transistors connect cell to bit lines
- Sense amplifiers detect voltage difference on bit lines
- Data is read from bit lines

#### Write Operation:

- Word line is activated
- Access transistors connect cell to bit lines
- Write drivers force bit lines to desired values
- Cell state changes to match bit line values

### Asynchronous SRAM Interface

Asynchronous SRAM devices typically have these control signals:

- **CS (Chip Select):** Enables the memory device (active low)
- **OE (Output Enable):** Enables data output during read (active low)
- **WE (Write Enable):** Indicates write operation (active low)
- **Address Lines:** Select memory location
- **Data Lines:** Bidirectional lines for data transfer

## SDRAM (Synchronous DRAM)

### SDRAM Structure and Characteristics

- Uses a capacitor and one transistor for each bit
- Requires periodic refresh to maintain data (capacitor leakage)
- Synchronous interface (clocked)
- Row and column addressing (multiplexed address bus)
- Higher density and lower cost than SRAM
- Higher power consumption due to refresh
- Used for main memory in computers and embedded systems

### SDRAM Operation

Key aspects of SDRAM operation:

- **Refresh:** Periodic read and rewrite of all memory cells
- **Row Activation:** Opening a row copies data to row buffer
- **Column Access:** Selecting specific bytes from row buffer
- **Precharge:** Preparing a bank for next row activation
- **Burst Mode:** Sequential access to multiple columns

### SRAM vs. SDRAM Comparison

Feature	SRAM	SDRAM
Cell Structure	6 transistors (flip-flop)	1 transistor + 1 capacitor
Refresh	Not required	Required (periodic)
Density	Lower	Higher
Cost per bit	Higher	Lower
Access Time	Faster, uniform	Variable (row hit vs. miss)
Interface	Often asynchronous	Synchronous (clocked)
Power Consumption	Lower static power	Higher due to refresh
Applications	Cache, high-speed buffer	Main memory

# Memory Technology Comparison

## Comparing Memory Technologies

### Identify memory characteristics

- **Volatility:** Whether memory loses contents when power is removed
  - *Volatile:* SRAM, DRAM/SDRAM
  - *Non-volatile:* PROM, EEPROM, Flash, NV-RAM
- **Storage mechanism:** How bits are physically stored
  - *SRAM:* Flip-flop-based cells (6 transistors)
  - *DRAM:* Capacitor-based cells (1 transistor + 1 capacitor)
  - *Flash:* Floating-gate transistors
- **Access method:** How data is accessed
  - *Random access:* Any location accessed in the same time (SRAM, NOR Flash)
  - *Block access:* Efficient for large blocks (NAND Flash)
  - *Sequential access:* Fast for sequential data (SDRAM with burst mode)

### Compare memory performance metrics

- **Access time:** Time to read/write data
  - SRAM: 2-10ns
  - SDRAM: 60ns+ for first access, then fast for burst
  - NOR Flash: 120ns read
  - NAND Flash: Slow random access (25µs first byte)
- **Density:** Storage capacity per unit area
  - SRAM: Low (large cells, expensive)
  - DRAM: High (small cells, inexpensive)
  - NAND Flash: Very high (highest density, lowest cost per bit)
- **Power consumption:** Energy required for operation
  - SRAM: Low static power (no refresh needed)
  - DRAM: Higher (requires refresh)
  - Flash: Very low when not being written/erased

### Analyze application suitability

- **SRAM:** Cache memory, small temporary storage
- **DRAM/SDRAM:** Main memory, large temporary storage
- **NOR Flash:** Program code storage, direct execution
- **NAND Flash:** Mass storage, data logging, SSD drives

## NOR vs. NAND Flash Vergleich

### NOR Flash

- ✓ Direkter Code-Execution (XIP - Execute in Place)
- ✓ Wahlfreie Byte-Zugriffe
- ✓ SRAM-kompatible Schnittstelle
- ✓ Bits einzeln auf '0' schreibbar
- ✓ Für Programmcode und persistente Daten
- ✗ Nicht für große Datenblöcke optimiert

### NAND Flash

- ✓ Effizient für große Datenblöcke
- ✓ Hohe Schreibgeschwindigkeit für Blöcke
- ✓ Spezielle Schnittstelle (nicht SRAM-kompatibel)
- ✓ Hohe Latenz für ersten Zugriff
- ✓ Für SSDs verwendet
- ✗ Code muss ins RAM geladen werden
- ✗ Keine wahlfreien Byte-Zugriffe

### Gemeinsame Eigenschaften

- ✓ Floating Gate Technologie
- ✓ Nur sektorweises Löschen auf '1'
- ✗ Bits können nicht einzeln auf '1' geschrieben werden

## SRAM vs. SDRAM Unterschiede

### Speichertyp

- **SRAM:** Statisch (Flip-Flop basiert)
- **SDRAM:** Dynamisch (Kondensator basiert)

### Refresh-Verhalten

- **SRAM:** Stores data as long as power is applied, without requiring refresh.
- **SDRAM:** Stores data as charge in a capacitor that leaks over time. Requires periodic refresh to maintain data integrity.

### Schnittstelle

- **SRAM:** Asynchron (NWE, NOE Signale)
- **SDRAM:** Synchron (RAS, CAS Signale, getaktet)

### Zugriffszeit und Timing

- **SRAM:** Provides fast random access with consistent timing for all accesses ( 5ns). All operations take roughly the same time.
- **SDRAM:** Synchronous interface with clock. Has higher latency for first access in a row ( 60ns), but very fast subsequent access within same row. Optimized for burst transfers.

### Speicherdichte und Kosten

- **SRAM:** Low density due to large cell size (6 transistors). Higher cost per bit. Typically limited to 64Mb per device.
- **SDRAM:** High density due to small cell size (1 transistor + 1 capacitor). Lower cost per bit. Can reach 4Gb or more per device.

### Typical applications:

- **SRAM:** CPU cache memory, small buffer memory, applications requiring fast random access.
- **SDRAM:** Main system memory, large buffers, applications requiring high capacity with reasonable access speed.

**OE-Pin Funktion bei SRAM:** Kontrolliert, ob das Memory Daten auf den Bus treibt oder ob sich die Ausgangstreiber im Floating-Zustand befinden.

## DRAM-Eigenschaften bewerten

### Speicherzelle

- ✓ Kondensator + Transistor (nicht RS Flip-Flop)
- ✓ Sehr kleine Zelle → hohe Dichte

### Zugriffsmuster

- ✓ Hohe Latenz für einzelne Zugriffe
- ✓ Optimiert für Blockzugriffe (Burst)

### Refresh und Leistung

- ✓ Periodischer Refresh wegen Leckströmen
- ✓ Volatiler Speicher
- ✗ Kein niedriger Leistungsverbrauch (Refresh erforderlich)

### Kosten

- ✗ Niedriger Preis pro Speicherzelle (nicht hoch)

## STM32F4 On-Chip Memory

### STM32F4 Memory Architecture

The STM32F429ZI microcontroller includes:

- **Flash Memory:** 2 MB (program storage)
  - NOR flash with execute-in-place capability
  - Divided into sectors of varying sizes (16KB to 128KB)
  - Organized in two banks for read-while-write operations
- **SRAM:** 256 KB total
  - SRAM1: 112 KB
  - SRAM2: 16 KB
  - SRAM3: 64 KB
  - CCM (Core Coupled Memory): 64 KB (accessible only by CPU)

### STM32F4 Flash Characteristics

- **Write Operations:** Can only change bits from '1' to '0'
- **Erase Operations:** Resets all bits in a sector to '1'
- **Programming Time:** Around 16 $\mu$ s per double word
- **Erase Time:** 1-2 seconds for a 128KB sector
- **Endurance:** 10,000 erase cycles
- **Access Time:** Higher latency than SRAM (requires wait states)

## External Memory Interface

### Flexible Memory Controller (FMC)

The STM32F4 Flexible Memory Controller (FMC) provides:

- Interface between on-chip system bus and external memory devices
- Support for different memory types:
  - SRAM, NOR Flash, PSRAM
  - NAND Flash
  - SDRAM
- Configurable bus width (8, 16, or 32 bits)
- Programmable timing parameters
- Memory banking with up to 6 banks

### FMC Signals

Key FMC signals for external SRAM/NOR flash:

- **A[25:0]:** Address bus
- **D[31:0]:** Data bus (bidirectional)
- **NE[4:1]:** Chip enable signals (active low)
- **NOE:** Output enable (active low)
- **NWE:** Write enable (active low)
- **NBL[3:0]:** Byte lane enables (active low)

### External Memory Access

Accessing external memory with different bus widths:

- **32-bit CPU Access to 32-bit Memory:** 1 external bus cycle
- **32-bit CPU Access to 16-bit Memory:** 2 external bus cycles
- **32-bit CPU Access to 8-bit Memory:** 4 external bus cycles

### Write Operations:

- CPU write stored in FMC FIFO buffer
- System bus released for other access
- FMC completes external write(s)

### Read Operations:

- System bus must wait until all external reads complete
- Multiple external cycles for narrow memory widths

## External Memory Configuration

### Connecting External Memory to STM32

Identify external memory type and requirements

- Determine memory interface type (asynchronous SRAM, NOR Flash, NAND Flash, SDRAM)
- Identify memory capacity and organization (width x depth)
- Note timing requirements from datasheet

### Configure FMC (Flexible Memory Controller)

- Enable FMC clock in RCC registers
- Configure memory bank registers based on memory type:
  - Bank 1-4: SRAM/NOR/PSRAM (BCRx and BTRx registers)
  - Bank 5-6: SDRAM (SDCR and SDTR registers)
- Set data bus width (8/16/32 bits)
- Configure timing parameters:
  - ADDSET: Address setup time
  - DATAST: Data setup time
  - BUSTURN: Bus turnaround time

### Configure GPIO pins for FMC

- Enable GPIO clocks in RCC registers
- Configure GPIO pins for alternate function (FMC)
- Set GPIO speed, typically high or very high

### Understand memory mapping

- Know the address ranges for each FMC bank:
  - Bank 1: 0x6000 0000 - 0x6FFF FFFF
  - Bank 2: 0x7000 0000 - 0x7FFF FFFF
  - Bank 3: 0x8000 0000 - 0x8FFF FFFF
  - Bank 4: 0x9000 0000 - 0x9FFF FFFF
  - SDRAM Bank 1: 0xC000 0000 - 0xCFFF FFFF
  - SDRAM Bank 2: 0xD000 0000 - 0xDFFF FFFF
- Understand chip select logic (NE1-NE4)
- Account for memory width in address calculations

A 64K × 8-bit asynchronous SRAM chip needs to be connected to the FMC of an STM32F429 microcontroller. The address 0x6800'0000 should be the lowest address used to access the memory.

Answer:

1. How many address pins does the memory need?
2. Which FMC signals should be connected to the SRAM chip?
3. At what address is the highest byte of the memory accessed?
4. Explain why the same memory location can be accessed at multiple addresses.

### 1. Number of address pins:

- Memory size = 64K =  $2^{16}$  bytes
- Need 16 address lines: ADDR[15:0]

### 2. FMC signals to connect:

- Address lines: A[15:0]
- Data lines: D[7:0] (8-bit data bus)
- Chip select: NE[3] (for Bank 3 based on address 0x6800'0000)
- Output Enable: NOE
- Write Enable: NWE

### 3. Highest memory address:

- Lowest address: 0x6800'0000
- Memory size: 64K = 0x10000 bytes
- Highest address: 0x6800'0000 + 0xFFFF = 0x6800'FFFF

### 4. Multiple address access:

- This occurs due to partial address decoding
- Only address lines A[15:0] are connected to the memory chip
- Higher address bits A[25:16] are not decoded/connected
- Any address where A[27:26] = 01 (Bank 2) and A[15:0] match will access the same memory location
- Example: Addresses 0x6800'0000, 0x6900'0000, 0x6A00'0000, etc. all access the same byte in memory
- The number of 64KB address blocks that map to the same memory is  $2^{10} = 1024$  (from the 10 undecoded bits A[25:16])

## Connecting Asynchronous SRAM to STM32F4

### Step 1: Configure GPIO pins

Set the GPIO pins for FMC signals to alternate function mode.

### Step 2: Configure FMC timing

Set appropriate timing parameters (ADDSET, DATAST) based on memory datasheet.

### Step 3: Configure FMC bank

Set the memory type, data width, and other parameters.

### Step 4: Enable FMC

Enable the FMC peripheral.

```
1 // Configure external SRAM (16-bit) on FMC bank 1
2
3 // Step 1: Configure GPIO pins for FMC
4 // Enable GPIO clocks
5 RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN | RCC_AHB1ENR_GPIOEEN |
6     RCC_AHB1ENR_GPIOFEN | RCC_AHB1ENR_GPIOGEN;
7
8 // Configure GPIO pins (example for some pins)
9 // Set alternate function mode (0x2)
10 GPIOD->MODER |= 0x55555555; // All pins to alternate function
11 GPIOE->MODER |= 0x55555555;
12 // Set to AF12 (FMC)
13 GPIOD->AFR[0] = 0xCCCCCCCC;
14 GPIOD->AFR[1] = 0xCCCCCCCC;
15 GPIOE->AFR[0] = 0xCCCCCCCC;
16 GPIOE->AFR[1] = 0xCCCCCCCC;
17
18 // Step 2: Enable FMC clock
19 RCC->AHB3ENR |= RCC_AHB3ENR_FMCEN;
20
21 // Step 3: Configure FMC bank 1 for SRAM
22 // Set timing for SRAM (example values)
23 FMC_Bank1->BTCSR[0] =
24     FMC_BCR1_MBKEN | // Memory bank enable
25     FMC_BCR1_MTYP_0 | // Memory type SRAM
26     FMC_BCR1_MWID_0 | // 16-bit data bus
27     FMC_BCR1_WREN; // Write enable
28
29 // Set timing (ADDSET=1, DATAST=2)
30 FMC_Bank1->BTCSR[1] =
31     (1 << FMC_BTR1_ADDSET_Pos) |
32     (2 << FMC_BTR1_DATAST_Pos);
```

## Memory Access and Address Decomposition

### Analyzing Memory Access Patterns

#### Determine memory access type

- Identify memory type and organization (width × depth)
- Determine CPU data bus width (typically 32-bit for Cortex-M)
- Identify the size of the access (byte, half-word, word)

#### Memory address decomposition

- Identify bank select bits from address (typically bits 27:26)
  - 00: Bank 1, 01: Bank 2, 10: Bank 3, 11: Bank 4
- Identify chip enable from bank and address
  - Within each bank, specific regions activate NE1-NE4
- Determine memory location within the device (lower address bits)

#### Analyze byte enables for sub-word accesses

- For 32-bit data bus: NBL[3:0] controls which bytes are active
  - Word access (4 bytes): NBL[3:0] = 0000 (all active)
  - Half-word access (2 bytes): NBL[3:0] = 0011 or 1100
  - Byte access (1 byte): NBL[3:0] = 0111, 1011, 1101, or 1110
- For 16-bit data bus: NBL[1:0] controls which bytes are active
  - Half-word access (2 bytes): NBL[1:0] = 00 (both active)
  - Byte access (1 byte): NBL[1:0] = 01 or 10

#### Determine access pattern

- For 8-bit memory with 32-bit CPU: 4 accesses per word
- For 16-bit memory with 32-bit CPU: 2 accesses per word
- Account for address alignment:
  - Unaligned access may require additional memory cycles
  - Byte ordering (little-endian for ARM) affects access pattern

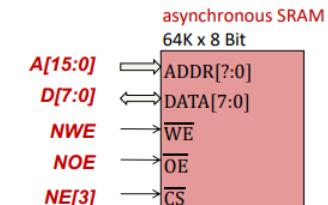
### SRAM-Anbindung berechnen

#### Anzahl Adresspins

Für  $x \times K \times y$  Bit SRAM: Adresspins =  $\log_2(x \times K) = \log_2(x \times 1024)$

#### FMC-Signale zuordnen

- NE[x]: Chip Select (Bank-Auswahl)
- NOE: Output Enable
- NWE: Write Enable
- A[n:0]: Adressleitungen
- D[m:0]: Datenleitungen



#### Basisadresse + Anzahl Adressen

- Basisadresse: niedrigste Adresse des Bausteins
- Höchste Adresse: Basisadresse + Speichergröße - 1
- Beispiel 64K:  $64K = 64 \times 1024 = 65536 = 0x10000$

#### Partial Address Decoding

- Nicht alle Adressleitungen werden dekodiert
- Führt zu mehreren gültigen Adressbereichen (Aliasing)
- Anzahl Bereiche =  $2^{\text{nicht dekodierte Bits}}$

## Memory Access Analysis

A 16-bit wide asynchronous SRAM is connected to the FMC of an STM32F429 microcontroller. Analyze what happens when the CPU writes a single byte to address 0x6402'8F21:

1. Determine the memory device and bank
2. Calculate the memory location in the device
3. Identify which byte line (NBL) is activated
4. Explain the complete address decomposition

### 1. Memory device and bank:

- Address 0x6402'8F21 begins with 0x64...
- From address bits [27:26] = 01, this is in Bank 2
- This activates NE2 (chip select 2)

### 2. Memory location in device:

- Only address bits [25:1] are passed to the SRAM device
- Bit [0] is used to select the high/low byte within a 16-bit word
- Memory location = 0x0014'7490 (shifted right by 1 bit and ignoring higher bits that aren't connected)

### 3. Byte line activation:

- Address bit [0] = 1, so we're accessing the high byte in the 16-bit word
- For 16-bit memory: NBL[1:0] = 10 (high byte active)
- NBL[0] = 1 (not active), NBL[1] = 0 (active)

### 4. Complete address decomposition:

- 0x6402'8F21 decomposed:
  - Bits [31:28]: 0x6 (not used for decoding)
  - Bits [27:26]: 01 (Bank 2, activates NE2)
  - Bits [25:1]: Memory address within SRAM (0x0014'7490)
  - Bit [0]: 1 (select high byte)
- FMC signals during access:
  - NE2 = 0 (active)
  - A[24:0] = 0x0014'7490 (memory address)
  - NBL[1:0] = 10 (access high byte)
  - NWE = 0 (active, writing)
  - Data appears on D[15:8] (high byte)

**64K x 8 Bit SRAM Anbindung** SRAM soll bei Basisadresse 0x6800'0000 angeschlossen werden.

a) **Anzahl Adresspins:**  $64K = 2^{16} \rightarrow 16$  Adresspins  $\rightarrow A[15:0]$

b) **FMC-Signale:**

- NE[3] (Bank 3: 0x6800'0000 - 0x6BFF'FFFF)
- NOE (Output Enable)
- NWE (Write Enable)
- A[15:0] (Adressleitungen)
- D[7:0] (8-bit Datenbus)

c) **Höchste Adresse:**  $0x6800'0000 + 64K - 1 = 0x6800'0000 + 0xFFFF = 0x6800'FFFF$

d) **Partial Address Decoding:** Bits A[25:16] nicht angeschlossen  $\rightarrow 10$  Bits  $\rightarrow 2^{10} = 1024$  Adressblöcke

e) **Aliasing-Bereiche:** 0x68XX'0000, 0x69XX'0000, 0x6AXX'0000, 0x6BXX'0000

## Address Space Calculation for External Memory

Calculate the address range for an external 32K x 8-bit SRAM connected to FMC bank 3.

For an external SRAM connected to FMC bank 3:

1. Base address of FMC bank 3 = 0x68000000
2. Memory size = 32K bytes = 32,768 bytes = 0x8000 bytes

Therefore, the address range for this SRAM would be:

- Start address: 0x68000000
- End address:  $0x68000000 + 0x8000 - 1 = 0x68007FFF$

Address range: 0x68000000 - 0x68007FFF

Note: Due to partial address decoding, this SRAM might also be accessible at other addresses within bank 3.

For example, it might also respond to addresses:  
0x68008000 - 0x6800FFFF, 0x68010000 - 0x68017FFF, etc.

## Cache

### Principle of Locality and Memory Hierarchy

#### Memory Hierarchy and Locality

Programs usually access small regions of memory in a given interval of time.

The memory hierarchy in computer systems is designed to exploit the principle of locality:

- **Spatial Locality:** If a memory location is accessed, nearby locations are likely to be accessed soon
  - Current data location is likely being close to next accessed location
  - Example: Sequential access to array elements
- **Temporal Locality:** If a memory location is accessed, it's likely to be accessed again soon
  - Current data location is likely being accessed again in near future
  - Example: Loop variables, frequently called functions

The memory hierarchy takes advantage of these patterns:

- Faster, smaller memories (cache) store recently/frequently accessed data
- Larger, slower memories (main memory, disk) store less frequently accessed data

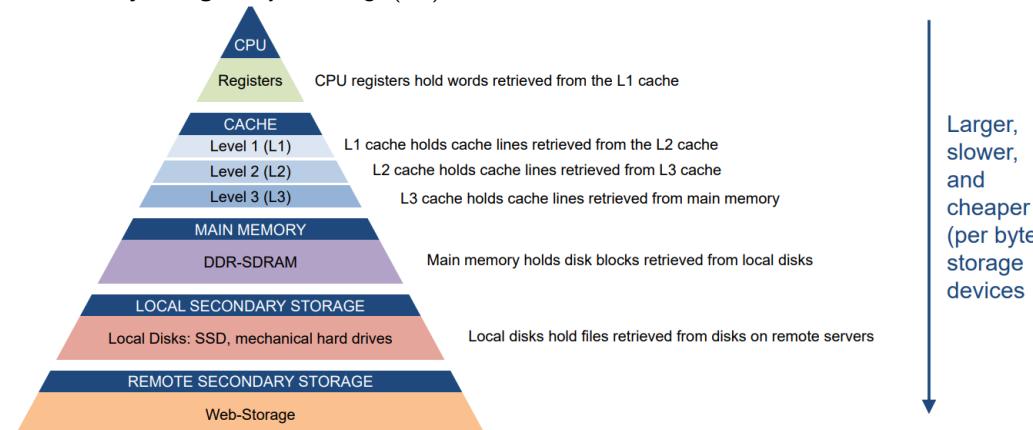
#### Principle of Locality

```
1 for (int i = 0; i < N; i++) { // incremental access
2     a[i] = b[i];           // spatial locality
3 }
4
5 if (a[1234] == a[4321]) { // temporal locality
6     a[1234] = 0;          // access to same location again
7 }
```

#### Memory Hierarchy Levels

Typical memory hierarchy in a modern system:

- **CPU Registers:** Fastest, smallest (bytes)
- **L1 Cache:** Very fast, small (KB)
  - Often split into instruction and data caches (Harvard architecture)
- **L2 Cache:** Fast, medium size (hundreds of KB)
- **L3 Cache:** Moderately fast, larger (MB)
- **Main Memory (RAM):** Slower, much larger (GB)
- **Secondary Storage:** Very slow, huge (TB)



#### Situation:

- Processor: fast cycle time
- Fast DRAM: Single accesses have large overhead (slow), efficiently reads only in bursts
- → bridging the gap such that pipelining and the bursts are effective!

#### Goal:

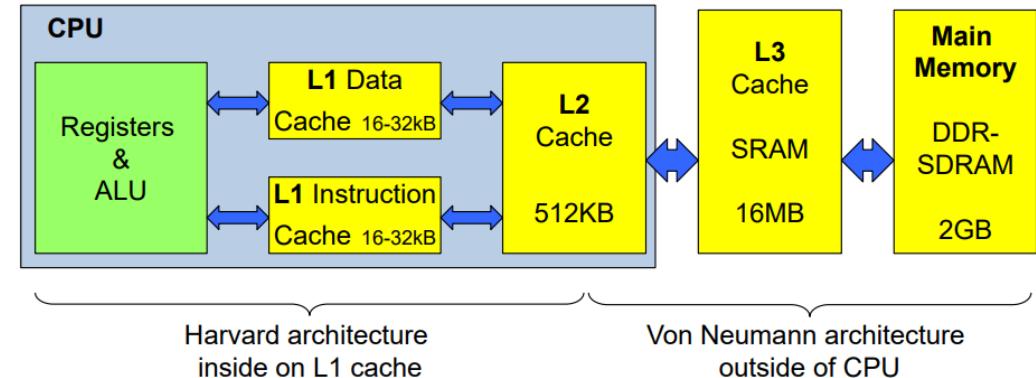
- Access 'slower' main memory in bursts and maintain a fast cache memory for fast single accesses
- But: Data consistency must be carefully managed, such that both, cache and main memory have the same data

## Cache Mechanics

### Cache Levels

### Typical Cache Architecture

Memory → larger, slower, cheaper



### Cache Operation

A cache is a small, fast memory that stores copies of data from frequently used main memory locations:

- Main memory is divided into fixed-size blocks
- Cache holds copies of some memory blocks
- When CPU needs data:
  - **Cache Hit:** Data is in cache → fast access
  - **Cache Miss:** Data not in cache → fetched from main memory, then stored in cache for future access

### Cache Terminology

Key terms in cache design:

- **Cache Line:** Basic unit of data transfer (typically 32-128 bytes)
- **Tag:** Part of address that identifies which memory block is stored
- **Valid Bit:** Indicates if the cache line contains valid data
- **Hit Rate:** Percentage of memory accesses found in cache
- **Miss Rate:** Percentage of memory accesses not found in cache
- **Hit Time:** Time to access data in cache
- **Miss Penalty:** Extra time required to fetch data from main memory

## Cache Organization

### Memory Addressing for Cache

A memory address is typically divided into:

- **Tag:** Identifies which memory block is stored
- **Index:** Determines location in cache (for direct-mapped and set-associative)
- **Offset:** Identifies specific byte within the block



The size of each field depends on the cache organization.

### Fully Associative Cache

Any memory block can be stored in any cache line

- Address format: [Tag | Offset]
- Each cache line stores:
  - Valid bit (is data valid?)
  - Tag (which memory block)
  - Data (contents of memory block)
- On memory access, the tag is compared with all cache lines in parallel
- Highest flexibility, best hit rate
- Requires many comparators (one per cache line)
- Complex hardware: requires comparing tag with all cache lines

### Direct Mapped Cache

Each memory block maps to exactly one cache line

- Mapping function: Line = Block mod  $m$  (where  $m$  is number of cache lines)
- Address format: [Tag | Index | Offset]
- The index directly selects which cache line to check
- Only one tag comparison needed
- Simple hardware but can suffer from conflict misses (resulting in lower hit rate)
  - Different memory blocks mapping to the same cache line

### N-Way Set Associative Cache

Cache is divided into sets, each containing  $N$  lines

- Memory block maps to a specific set, can go in any line within that set
- Mapping function: Set = Block mod  $s$  (where  $s$  is number of sets)
- Address format: [Tag | Set Index | Offset]
- Set index selects which set to check
- $N$  tag comparisons needed (one per line in set)
- Compromise between fully associative and direct mapped
  - More flexible than direct mapped
  - Less hardware than fully associative

### Cache Organization Analysis

Consider a system with a 4 KiB direct-mapped cache with 16-byte cache lines. The system uses 32-bit byte addresses.

1. How many cache lines are there?
2. How many bits are needed for the tag, index, and offset?
3. For memory address 0x1234ABCD, determine the tag, index, and offset.
4. Which other addresses would map to the same cache line?

#### 1. Number of cache lines:

- Cache size = 4 KiB = 4096 bytes
- Line size = 16 bytes
- Number of lines =  $4096 / 16 = 256$  lines

#### 2. Address bit fields:

- Offset bits =  $\log_2(16) = 4$  bits (addresses byte within the line)
- Index bits =  $\log_2(256) = 8$  bits (selects the cache line)
- Tag bits =  $32 - (4 + 8) = 20$  bits (identifies which memory block is cached)

#### 3. Address decomposition for 0x1234ABCD:

- Convert to binary: 0001 0010 0011 0100 1010 1011 1100 1101
- Offset: last 4 bits = 1101 = 0xD
- Index: next 8 bits = 1011 1100 = 0xBC
- Tag: remaining 20 bits = 0001 0010 0011 0100 1010 = 0x1234A

#### 4. Other addresses mapping to the same cache line:

- Addresses with the same index (0xBC) but different tags would map to the same line
- General form: 0XXXXXXBCY where:
  - XXXXX can be any value except 0x1234A (different tag)
  - BC is fixed (same index)
  - Y can be any value from 0 to F (different offset within the line)
- Examples: 0x0034ABCD, 0x5678ABC, 0x9ABCDBCE, etc.
- These addresses would cause cache conflicts if accessed in sequence

### Cache Organization Calculation

For a 16KB cache with 64-byte lines, calculate address breakdown for:

- Direct mapped
- 4-way set associative

Given:

- Cache size = 16KB = 16,384 bytes
- Line size = 64 bytes

#### 1. Direct Mapped Cache:

- Number of cache lines = Cache size / Line size =  $16,384 / 64 = 256$  lines
- Index bits =  $\log_2(256) = 8$  bits
- Offset bits =  $\log_2(64) = 6$  bits
- For 32-bit address: Tag =  $32 - 8 - 6 = 18$  bits

Address format: [18-bit Tag | 8-bit Index | 6-bit Offset]

#### 2. 4-Way Set Associative:

- Number of sets = Cache size / (Line size × Associativity) =  $16,384 / (64 \times 4) = 64$  sets
- Set index bits =  $\log_2(64) = 6$  bits
- Offset bits =  $\log_2(64) = 6$  bits
- For 32-bit address: Tag =  $32 - 6 - 6 = 20$  bits

Address format: [20-bit Tag | 6-bit Set Index | 6-bit Offset]

## Cache Organization and Addressing

### Analyzing Cache Organization

Identify cache parameters

- Cache size: Total data storage capacity
- Block/line size: Size of each cache line (typically 16-128 bytes)
- Associativity: Number of ways ( $n=1$  for direct mapped,  $n=\text{cache size}/\text{block size}$  for fully associative)
- Number of sets:  $s = \text{cache size} / (\text{block size} \times \text{associativity})$

### Address decomposition

- Divide address into tag, index, and offset fields
- Offset bits =  $\log_2(\text{block size})$
- Index bits =  $\log_2(\text{number of sets})$
- Tag bits = address bits - (offset bits + index bits)

### Determine cache organization

- **Direct mapped cache:**
  - One line per set (associativity = 1)
  - Number of sets = cache size / block size
  - Index field directly selects the cache line
- **Fully associative cache:**
  - One set with all lines (number of sets = 1)
  - No index field, only tag and offset
  - Requires comparison with all cache line tags
- **N-way set associative cache:**
  - N lines per set
  - Number of sets = cache size / ( $N \times \text{block size}$ )
  - Index field selects the set, tag comparison within the set

### Map memory addresses to cache

- Calculate set number:  $(\text{address} / \text{block size}) \bmod (\text{number of sets})$
- Calculate tag:  $\text{address} / (\text{block size} \times \text{number of sets})$
- Calculate offset:  $\text{address} \bmod \text{block size}$

## Cache Performance

**Cache Miss Types** Three fundamental types of cache misses:

- **Compulsory Misses** (Cold Misses):
  - First access to a block, must be fetched from memory
  - Unavoidable, regardless of cache size or organization
- **Capacity Misses**:
  - Cache too small to hold all blocks needed during program execution
  - Blocks evicted and later needed again
  - Can be reduced by increasing cache size
- **Conflict Misses**:
  - Multiple blocks map to same cache line/set
  - More common in direct mapped, less in set associative
  - Can be reduced by increasing associativity

## Cache Performance

Average memory access time (AMAT) can be calculated as:

$$AMAT = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$$

Example:

- Hit Time = 1 cycle
- Miss Penalty = 100 cycles
- Miss Rate = 3% (0.03)

$$AMAT = 1 + 0.03 \times 100 = 1 + 3 = 4 \text{ cycles}$$

Reducing miss rate from 3% to 1% changes AMAT to:

$$AMAT = 1 + 0.01 \times 100 = 1 + 1 = 2 \text{ cycles}$$

This is a 50% performance improvement!

## Cache Size vs. Hit Rate

Relationship between cache parameters and hit rate:

- Increasing cache size improves hit rate
  - But with diminishing returns
  - Larger cache may have longer hit time
- Increasing associativity improves hit rate
  - Greatest benefit from direct-mapped to 2-way
  - Diminishing returns beyond 4-way or 8-way
  - Higher associativity increases complexity and hit time
- Increasing block size improves spatial locality exploitation
  - But increases miss penalty
  - Very large blocks may cause pollution

## Cache Performance Analysis

### Analyzing Cache Hit Rates and Performance

Calculate basic cache metrics

- Hit rate = Number of hits / Total number of accesses
- Miss rate = 1 - Hit rate
- Average memory access time (AMAT) = Hit time + (Miss rate × Miss penalty)

Identify types of cache misses

- **Compulsory misses**: First access to a block (cold start)
- **Capacity misses**: Cache is full, blocks need to be evicted
- **Conflict misses**: In direct-mapped or set-associative caches when multiple blocks map to same set

Analyze memory access patterns

- Spatial locality: Sequential or nearby accesses
- Temporal locality: Repeated accesses to same location
- Stride patterns: Regular jumps between memory locations

Evaluate impact of cache parameters

- Effect of cache size: Larger cache reduces capacity misses
- Effect of block size: Larger blocks improve spatial locality, but may increase miss penalty
- Effect of associativity: Higher associativity reduces conflict misses
- Effect of replacement policy: LRU, FIFO, Random - impact on hit rate

### Cache Performance Calculation

Consider a system with the following cache parameters:

- Cache access time: 1 processor cycle
- Main memory access time: 100 processor cycles
- Miss rate for cache configuration A: 3%
- Miss rate for cache configuration B: 1%

1. Calculate the average memory access time (AMAT) for both configurations
2. If configuration B has twice the associativity of A, explain why the miss rate improved
3. If the program executes 1 million memory accesses, how many cycles are saved by using configuration B instead of A?

#### 1. Calculate AMAT for both configurations:

- $AMAT = \text{Hit time} + (\text{Miss rate} \times \text{Miss penalty})$
- For configuration A:
  - $AMAT_A = 1 \text{ cycle} + (0.03 \times 100 \text{ cycles}) = 1 + 3 = 4 \text{ cycles}$
- For configuration B:
  - $AMAT_B = 1 \text{ cycle} + (0.01 \times 100 \text{ cycles}) = 1 + 1 = 2 \text{ cycles}$

#### 2. Effect of increased associativity:

- Higher associativity reduces conflict misses
- In configuration A (lower associativity), more memory blocks map to the same cache set
- With configuration B (doubled associativity), each set can hold twice as many blocks
- This reduces the chance of having to evict a useful block due to set conflicts
- The improved miss rate (from 3% to 1%) is primarily due to the reduction in conflict misses
- Note that capacity misses and compulsory misses would remain the same

#### 3. Cycles saved:

- Total accesses = 1,000,000
- Cycles for configuration A =  $1,000,000 \times 4 = 4,000,000 \text{ cycles}$
- Cycles for configuration B =  $1,000,000 \times 2 = 2,000,000 \text{ cycles}$
- Cycles saved =  $4,000,000 - 2,000,000 = 2,000,000 \text{ cycles}$

Therefore, using configuration B saves 2 million processor cycles for the program, cutting the memory access time in half.

## Replacement and Write Strategies

### Replacement Strategies

When a new block must be loaded into a full cache set, a replacement strategy determines which existing block to evict:

- **Least Recently Used (LRU):**
  - Evict the block that hasn't been accessed for the longest time
  - Good performance but complex to implement for high associativity
- **Least Frequently Used (LFU):**
  - Evict the block that has been accessed least often
  - Requires access counters for each block
- **First-In First-Out (FIFO):**
  - Evict the block that has been in cache longest
  - Simpler to implement than LRU
- **Random:**
  - Evict a random block
  - Simplest to implement
  - Performance often surprisingly close to LRU

### Write Strategies

When a write operation hits in cache, two main strategies exist:

- **Write-Through:**
    - Write data to both cache and main memory
    - Memory always consistent with cache
    - Slower for writes (must wait for memory)
    - Simpler coherence in multiprocessor systems
  - **Write-Back:**
    - Write data only to cache
    - Mark block as "dirty"
    - Write to memory only when block is evicted
    - Faster for repeated writes to same block
    - More complex coherence handling
- For write misses, two approaches:
- **Write-Allocate:**
    - Fetch block into cache, then update
    - Works well with write-back
  - **No-Write-Allocate:**
    - Write directly to memory, don't fetch block
    - Works well with write-through

## Programmer's Perspective

### Cache Performance Optimization

General guidelines for cache-friendly programming:

- **Loop Interchange:** Reorder nested loops to access memory sequentially
- **Loop Blocking/Tiling:** Break large loops into smaller chunks that fit in cache
- **Data Alignment:** Align data structures to cache line boundaries
- **Structure Packing:** Organize structure fields to minimize cache line usage
- **Prefetching:** Load data into cache before it's needed
- **Reduce Working Set Size:** Keep active data small enough to fit in cache

### Optimizing Code for Cache Performance

Analyze array access patterns

- Understand memory layout in your language (e.g., row-major in C, column-major in Fortran)
- Match loop iteration order to memory layout (e.g., for C: outer loop for rows, inner loop for columns)
- Avoid strided access patterns that lead to frequent cache misses

Improve spatial locality

- Place related data together in memory
- Use structures instead of separate arrays for related data
- Pad data structures to align with cache lines when appropriate

Improve temporal locality

- Reuse data while it's still in cache
- Use blocking/tiling for matrix operations to maximize data reuse
- Process data in chunks that fit in cache

Avoid cache conflicts

- Be aware of array sizes that are powers of 2, which can lead to systematic conflicts
- Pad arrays to avoid conflict misses in direct-mapped or set-associative caches
- Consider memory alignment to reduce conflicts

### Cache-Friendly Programming

**Maximize spatial locality:** Access memory in sequential patterns whenever possible.

**Maximize temporal locality:** Reuse recently accessed data before it gets evicted from cache.

**Be aware of cache line size:** Structure data to minimize cache line crossings.

**Consider memory layout:** Organize multidimensional arrays to match access patterns.

```
1 // Cache-unfriendly code (column-major traversal of row-major array)
2 for (j = 0; j < N; j++) {
3     for (i = 0; i < N; i++) {
4         sum += array[i][j]; // Poor spatial locality
5     }
6 }
7 // Cache-friendly code (row-major traversal of row-major array)
8 for (i = 0; i < N; i++) {
9     for (j = 0; j < N; j++) {
10        sum += array[i][j]; // Good spatial locality
11    }
12 }
```

Consider a matrix multiplication function for  $1000 \times 1000$  matrices. Two implementations are shown:

```
1 // Version A - Row by Column
2 for(i = 0; i < 1000; i++) {
3     for(j = 0; j < 1000; j++) {
4         for(k = 0; k < 1000; k++) {
5             C[i][j] += A[i][k] * B[k][j];
6         }
7     }
8 }
9 // Version B - Blocked/Tiled
10 for(i = 0; i < 1000; i += 64) {
11     for(j = 0; j < 1000; j += 64) {
12         for(k = 0; k < 1000; k += 64) {
13             for(ii = i; ii < min(i+64, 1000); ii++) {
14                 for(jj = j; jj < min(j+64, 1000); jj++) {
15                     for(kk = k; kk < min(k+64, 1000); kk++) {
16                         C[ii][jj] += A[ii][kk] * B[kk][jj];
17                     }
18                 }
19             }
20         }
21     }
22 }
```

Explain which version would perform better and why from a cache perspective.

Version B (the blocked/tiled implementation) will perform significantly better for the following cache-related reasons:

1. **Better spatial locality:**
  - Version A accesses matrix B in column-major order ( $B[k][j]$ ), which is inefficient in C where arrays are stored in row-major order
  - This means adjacent accesses to  $B[k][j]$  jump by 1000 elements (entire row), causing frequent cache misses
  - Version B limits this effect by working on small blocks at a time
2. **Better temporal locality:**
  - Version A loads each element of A and B 1000 times
  - Version B loads each element only once within each block, then reuses it multiple times
  - The blocks are sized (64x64) to fit within the cache, increasing reuse before eviction
3. **Reduced cache pressure:**
  - Version A works with the entire matrices at once
  - Version B only needs to keep 3 blocks (A, B, C) in cache at a time
  - For a typical L1 cache (32-64 KiB), the 64x64 blocks (32 KiB at 8 bytes per element) fit much better than the full matrices
4. **Fewer cache misses:**
  - Version A would cause frequent capacity misses as elements are evicted before reuse
  - Version B significantly reduces capacity misses by ensuring blocks fit in cache
  - Version B also reduces conflict misses by working with smaller, better aligned chunks

This tiling technique is a classic cache optimization that can improve performance by orders of magnitude for matrix operations on large matrices.

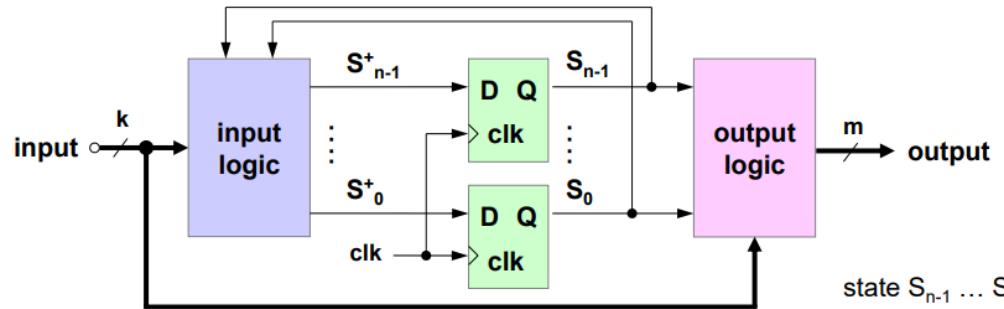
## Software State Machines

### Introduction - FSM in Hardware vs Software

**Finite State Machine (FSM)** A machine with a finite number of states and transitions, which responds to inputs based on its current state. It represents a computational model used to design both computer programs and sequential logic circuits. FSMs are particularly useful in embedded systems for controlling the behavior of the system in response to external events.

#### Hardware FSM

- Flip-Flops store internal state
- Clock-driven: Inputs evaluated at clock edges, state changes only on clock edges



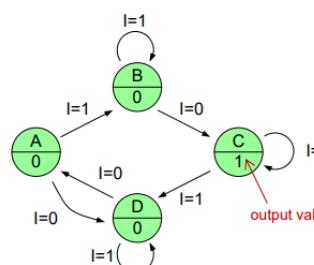
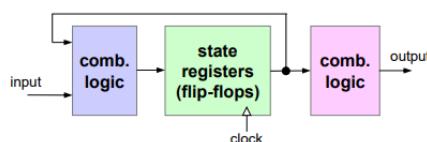
- Intrinsic parallel - multiple FSMs process simultaneously
- Unchanged input signals create no overhead
- Typically implemented as either Moore or Mealy machines

#### Moore vs Mealy FSMs

- Moore FSM: Outputs depend only on current state
- Mealy FSM: Outputs depend on current state and inputs

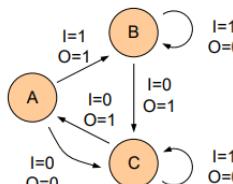
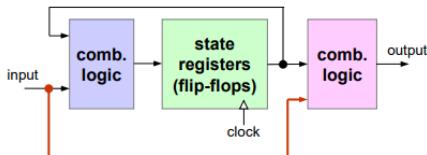
#### Moore

- Input signals influence state only



#### Mealy

- Input signals influence state AND output signals



## Software FSM is different!

### Why software is different

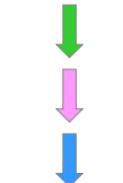
#### Hardware is intrinsically parallel

- Several FSMs can be processed in parallel using the same clock – Large number of flip-flops and gates evaluate simultaneously
- Use of common clock as the only event
- Evaluate signal levels of inputs at clock edges
- Unchanged signal levels of inputs do not create overhead



#### Software is intrinsically sequential

- CPU has to process one FSM after the other
- "HW approach" would require a function call on each clock edge
- All FSM inputs would have to be evaluated on each function call even if they have not changed → creates a large processing load for CPU
- Cooperating FSMs: Using a "synchronous clock approach" as in hardware creates a lot of synchronization issues in sequential system



→ use a different approach for software

### Reactive Systems

**Reactive System (State-Event Model)** A system that responds to external events (inputs) based on its internal state. It is event-driven and processes events as they occur, rather than evaluating all inputs periodically. Reactive systems are particularly well-suited for implementing FSMs in software because they minimize CPU overhead.

#### Components of a Reactive System

- **Events:** External inputs that trigger the system's response
- **Internal state:** Memory of what happened before
- **Actions:** Influence on the outside world (outputs)
- **Transitions:** Rules for how events change state and trigger actions

The key advantage of the reactive approach is that the FSM is evaluated only when an input changes, reducing processing overhead compared to polling all inputs regularly.

#### C common applications of state-event models include:

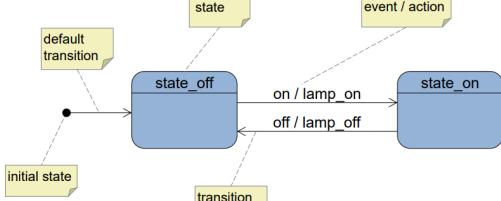
- Communication protocols (connection establishment, data transfer, connection termination)
- Human-machine interfaces (input recognition and validation)
- Parsing of programming languages and text
- Process control systems (washing machines, vending machines, heating systems)
- Embedded control systems (automotive, medical devices)

## Modeling State Machines in UML

**UML State Diagram** A graphical representation based on the Unified Modeling Language (UML) for modeling finite state machines. Based on the notation developed by Prof. David Harel, it describes the reactive behavior of systems. UML state diagrams provide a standard way to document and communicate the behavior of state-based systems.

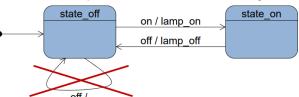
### UML State Diagram Elements

- **State:** Internal condition of the system waiting for the next event
- **Event:** Asynchronous input that may cause a transition
- **Transition:** Reaction to an event, may change state and/or trigger an action
- **Action:** Output associated with a transition (written after a forward slash)
- **Initial state:** Default starting state of the system (indicated by a solid circle)
- **Default transition:** Arrow from initial state to the first active state



In contrast to hardware FSM notations like Mealy, UML state diagrams:

- Treat inputs as asynchronous events rather than signal levels
- Treat outputs as actions
- Omit inputs that have no effect (increasing diagram clarity)
- Can represent both Mealy and Moore behaviors in a unified notation



### UML State Diagram erstellen

#### Grundelemente

- States (Zustände): Rechtecke mit Namen
- Transitions (Übergänge): Pfeile zwischen States
- Initial State: Gefüllter Kreis mit Pfeil
- Events: Auslöser für Transitions
- Actions: Aktionen bei Transitions

#### Transition-Syntax

##### Event / Action

- Event: Was löst den Übergang aus
- Action: Was wird beim Übergang ausgeführt
- Beispiel: start / water\_on, timer\_start

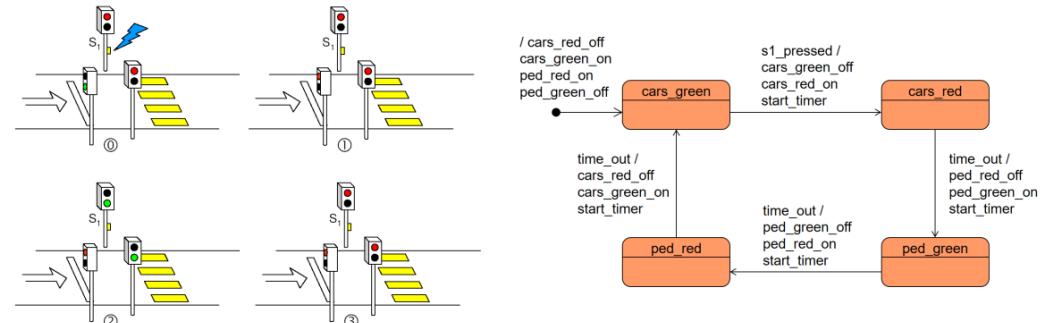
#### Vorgehen

1. Alle Zustände identifizieren
2. Events (Eingaben) definieren
3. Actions (Ausgaben) definieren
4. Übergänge zwischen Zuständen zeichnen
5. Initial State festlegen
6. Entry/Exit Actions definieren (falls nötig)

#### Regeln beachten

- Jeder State muss erreichbar sein
- Determinismus: Eindeutige Transitions
- Vollständigkeit: Alle Events behandeln

### Simple light switch state diagram in UML:



With an initial default transition to state\_off.

Note that the transition öffin state\_off is not shown because it would have no effect and trigger no action.

include SEP\_Handout UML and State Diagram Example ⇒ also includes C implementation of traffic light FSM  
Conclusion and Best Practices

#### Key Differences: Software vs Hardware FSMs

- Software FSMs are event-driven rather than clock-driven
- Software FSMs process one event at a time, while hardware FSMs can be parallel
- Software FSMs often use a state-event model for efficiency
- Software FSMs can use more complex data structures and conditions
- Interaction between software FSMs typically uses message passing

#### FSM Design Best Practices

##### Design Phase

- Start with a clear, well-defined problem statement
- Identify all possible states the system can be in
- Define all events that can occur and how they affect each state
- Use UML state diagrams to visualize and document the FSM
- Review for completeness, consistency, and determinism

##### Implementation Phase

- Keep state handling code separate from event detection
- Use enumerations for states and events to improve readability
- Keep ISRs short and move complex processing to the main loop
- Use event queues for communication between FSMs
- Consider using a table-driven approach for complex FSMs

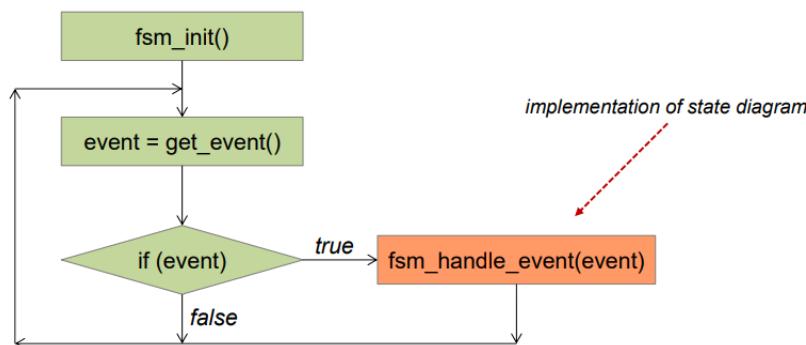
##### Testing and Debugging

- Test each state transition individually
- Verify correct behavior for unexpected or illegal events
- Add debug output for state transitions during development
- Consider adding state history for troubleshooting
- Test boundary conditions and error cases

## Implementation in C

**Basic Implementation Structure** Software implementation of an FSM typically uses two main functions:

- **fsm\_init()**: Initializes the state machine to its default state
- **fsm\_handle\_event(event)**: Processes an incoming event based on current state



The main program continuously polls for events and passes them to the state machine when they occur. This separation of event detection and processing is key to the efficiency of software FSMs.

### State Machine in C implementieren

#### Datenstrukturen

- Enum für States:  
  **typedef enum {STATE1, STATE2} state\_t;**
- Enum für Events:  
  **typedef enum {EVENT1, EVENT2} event\_t;**
- Statische Variable:  
  **static state\_t current\_state;**

#### Hauptschleife

```
1 int main(void) {
2     event_t event;
3     fsm_init();
4     while (1) {
5         event = get_event();
6         if (event != NO_EVENT) {
7             fsm_handle_event(event);
8         }
9     }
10 }
```

#### FSM Handler (Switch-Case)

```
1 void fsm_handle_event(event_t event) {
2     switch (current_state) {
3         case STATE1:
4             switch (event) {
5                 case EVENT1:
6                     action1();
7                     current_state =
8                         STATE2;
9                     break;
10                default:
11                    // unbehandelte Events
12                    break;
13            }
14            break;
15        case STATE2:
16            // ...
17            break;
18        default:
19            current_state = INITIAL_STATE;
20            break;
21    }
22 }
```

#### Event Detection

- Polling: Regelmäßige Abfrage in Hauptschleife
- Interrupt-driven: Events in ISR in Queue einreihen
- Edge Detection: Flanken erkennen (static Variable)

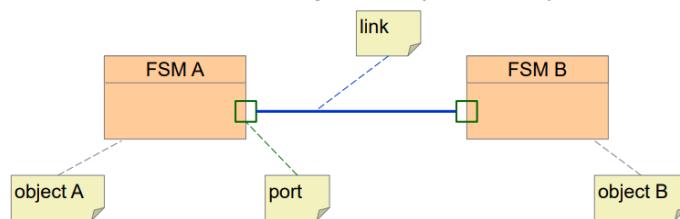
## FSM Implementation Pattern

```
1 // State and event enumerations
2 typedef enum {
3     STATE_A,
4     STATE_B,
5     STATE_C,
6     // ...other states
7 } state_t;
8
9 typedef enum {
10    NO_EVENT,
11    EVENT_X,
12    EVENT_Y,
13    // ...other events
14 } event_t;
15
16 // Current state variable
17 static state_t state;
18
19 // Initialization function
20 void fsm_init(void) {
21     state = STATE_A; // Set initial state
22     // Additional initialization if needed
23 }
24
25 // Event handler function
26 void fsm_handle_event(event_t event) {
27     switch (state) {
28         case STATE_A:
29             switch (event) {
30                 case EVENT_X:
31                     action1();
32                     state = STATE_B;
33                     break;
34                 case EVENT_Y:
35                     action2();
36                     // No state change
37                     break;
38                 default:
39                     // Event ignored in this state
40                     break;
41             }
42             break;
43         case STATE_B:
44             // Handle events for state B
45             break;
46         // Other states
47     }
48 }
49
50 // Main loop
51 int main(void) {
52     event_t event;
53
54     fsm_init();
55
56     while (1) {
57         event = get_event();
58         if (event != NO_EVENT) {
59             fsm_handle_event(event);
60         }
61     }
62 }
```

## Interaction of FSMs

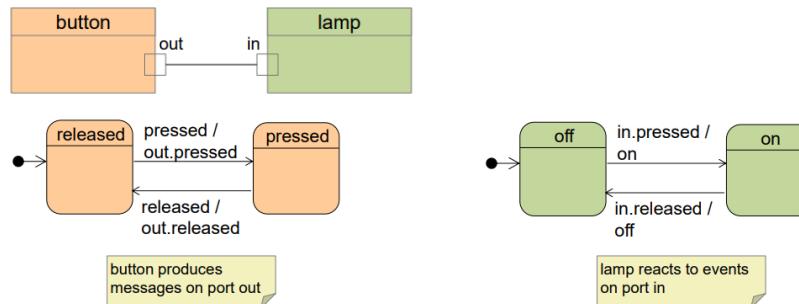
### FSM Interaction Components

- Port:** Defines the messages that can be sent and received by an FSM
  - Output message → action of the FSM
  - Input message → event for the FSM
- Link:** Defines a connection for sending messages between FSMs
- Event Queue:** Buffer for events generated by different objects



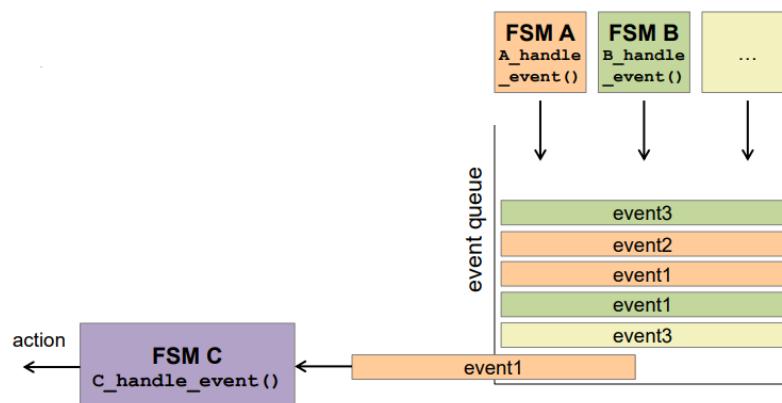
### Reactive System Partitioned into two FSMs

Interaction of FSMs happens through event messages.



### Event Queue for FSM Interaction

- Collects events generated by different objects
- Buffered to avoid losing events (especially important in interrupt-driven systems)
- FSM processes one event after the other
- Events are deleted after processing
- Provides decoupling between event producers and consumers



Actions of one FSM become events for another FSM, allowing complex systems to be constructed from simpler components. This approach enables modular design and clear separation of concerns.

## Building Event-Based FSM Systems

### System Structure

- Divide system into multiple FSMs with well-defined responsibilities
- Define ports and interfaces for each FSM
- Connect FSMs through links and event queues
- Keep individual FSMs simple and focused on specific tasks

### Implementation

- Use interrupt service routines (ISRs) to capture hardware events
- ISRs place events in queue rather than processing directly
- Main program continuously checks queue and dispatches events to appropriate FSMs
- Each FSM processes events according to its current state
- Ensure thread safety if multiple cores or interrupts are involved

### Benefits

- Clear separation of concerns
- Improved maintainability
- Reduced complex interdependencies
- More predictable system behavior
- Easier to test individual components
- Supports incremental development

## Examples of Software State Machines

### Autowaschanlage State Machine Spezifikation:

- Ruhezustand wartet auf Start
- Drei Schritte: wash, rinse, dry (je gleich lang)
- wash: Wasser + Shampoo
- rinse: nur Wasser
- dry: nur Luftstrom
- Stop-Taste bricht ab → alles aus, zurück zu Rest

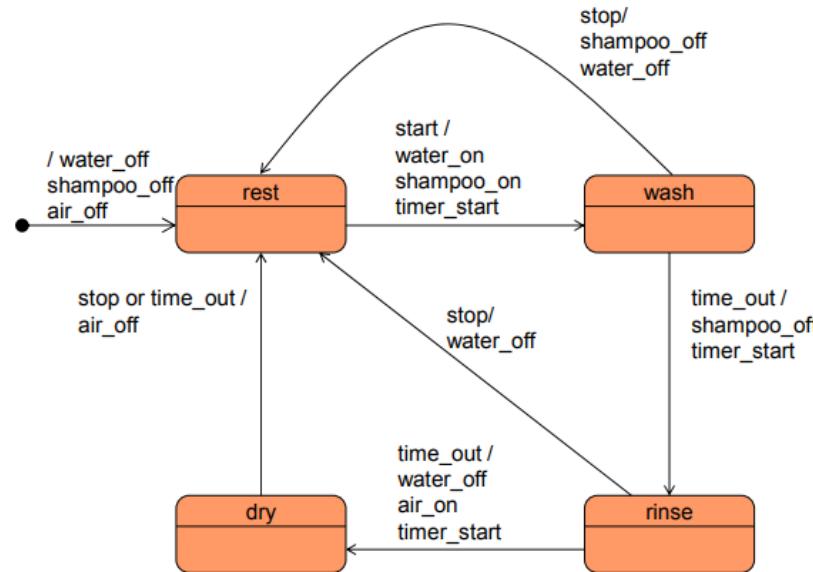
### Events:

- start: Starttaste gedrückt
- stop: Stoptaste gedrückt
- time\_out: Timer abgelaufen

### Actions:

- water\_on/off, shampoo\_on/off, air\_on/off
- timer\_start

### State Diagram:



- rest: Ausgangszustand
- wash: start / water\_on, shampoo\_on, timer\_start
- rinse: time\_out / shampoo\_off, timer\_start
- dry: time\_out / water\_off, air\_on, timer\_start
- Zurück zu rest: stop / water\_off, shampoo\_off, air\_off

## C-Implementation

### LED-Steuerung State Machine

#### Spezifikation:

- WAIT: Beide LEDs aus, warten auf S0
- ALL\_ON: Beide LEDs an bei S0
- S0 oder S1 in ALL\_ON → entsprechende LED aus, zurück zu WAIT

#### Implementierung:

```

1  typedef enum {
2      WAIT,
3      ALL_ON
4  } fsm_state_t;
5
6  typedef enum {
7      NO_SWITCH = 0x00,
8      S0 = 0x01,
9      S1 = 0x02
10 } event_t;
11
12 static fsm_state_t state = WAIT;
13
14 void fsm_handle_event(event_t event) {
15     switch (state) {
16         case WAIT:
17             switch (event) {
18                 case S0:
19                     led_turn_on(LED0_LED1);
20                     state = ALL_ON;
21                     break;
22                 default:
23                     state = WAIT;
24                     break;
25             }
26             break;
27
28         case ALL_ON:
29             switch (event) {
30                 case S0:
31                     led_turn_off(LED0);
32                     state = WAIT;
33                     break;
34                 case S1:
35                     led_turn_off(LED1);
36                     state = WAIT;
37                     break;
38                 default:
39                     state = ALL_ON;
40                     break;
41             }
42             break;
43
44         default:
45             state = WAIT;
46             break;
47     }
48 }
  
```

State Machines eignen sich besonders für reaktive Systeme mit klaren Zuständen und definierten Übergängen. Die Trennung von Event-Detection, State-Logic und Actions macht den Code wartbar und testbar.

## Interrupt Performance

### Event Detection Methods

#### Polling vs. Interrupt-Driven I/O

Two primary methods for detecting events in embedded systems:

- **Polling:** Periodically checking status registers
  - Synchronous with main program
  - CPU actively queries peripherals
  - Predictable timing
  - Simple implementation
- **Interrupt-Driven:** Hardware signals the CPU when events occur
  - Asynchronous with main program
  - CPU notified only when an event happens
  - Event-driven approach
  - More complex implementation

### Polling

#### Polling

Periodische Abfrage von Status-Informationen durch die CPU.

- Synchron mit dem Hauptprogramm
- CPU fragt aktiv nach Status-Aenderungen
- Einfach zu implementieren, deterministisch
- Verschwendet CPU-Zeit durch **Busy Wait**

#### Polling Implementation

In polling, the CPU periodically checks status registers to detect events:

- Main loop continuously or periodically inspects peripheral status
- When an event is detected, appropriate handler executes
- After handling, control returns to polling loop
- CPU always actively checking, even when no events occur

Advantages:

- Simple and straightforward implementation
- Deterministic behavior (predictable timing)
- No need for complex interrupt handling
- Implicit synchronization (operations happen in sequence)

Disadvantages:

- Wastes CPU cycles checking for events that haven't occurred
- Reduced system throughput (CPU busy checking instead of processing)
- Potentially long response times (if many devices must be checked)
- Inefficient for infrequent events

#### Implementing Polling in C

##### Step 1: Identify status registers

Determine which peripheral registers contain status information.

##### Step 2: Create a polling loop

Implement a loop that regularly checks the status flags.

##### Step 3: Check for events

Test specific bits in the status registers to detect events.

##### Step 4: Handle detected events

Process events when their status flags are set.

##### Step 5: Clear status flags

Reset status flags to prepare for next event detection.

```
1 // Main polling loop
2 while (1) {
3     // Check SPI transmit buffer empty flag
4     if (SPI1->SR & SPI_SR_TXE) {
5         // Handle SPI transmission
6         if (spi_tx_count < spi_tx_length) {
7             SPI1->DR = spi_tx_buffer[spi_tx_count++];
8         }
9     }
10
11    // Check UART receive data register not empty flag
12    if (USART2->SR & USART_SR_RXNE) {
13        // Handle UART reception
14        uint8_t data = USART2->DR;
15        process_uart_data(data);
16    }
17
18    // Check ADC end of conversion flag
19    if (ADC1->SR & ADC_SR_EOC) {
20        // Handle ADC conversion complete
21        uint16_t adc_value = ADC1->DR; // Reading DR clears EOC flag
22        process_adc_data(adc_value);
23    }
24
25    // Other system tasks
26    process_system_tasks();
27 }
```

## Interrupt-Driven I/O

### Interrupt-driven I/O

Peripheriegeräte signalisieren Events asynchron an die CPU.

- Hardware setzt Interrupt-Flag bei Event
- CPU wird unterbrochen und führt ISR aus
- Effiziente CPU-Nutzung
- Komplexere Implementierung

### Interrupt-Driven I/O

In interrupt-driven I/O, peripherals notify the CPU when events occur:

- Peripherals assert interrupt signal when they need servicing
- CPU temporarily suspends current execution
- Control transfers to an Interrupt Service Routine (ISR)
- After handling the interrupt, control returns to previous execution

Advantages:

- Efficient CPU utilization (only responds when needed)
- Fast response to events
- Good for infrequent, time-critical events
- Main program can focus on primary tasks

Disadvantages:

- More complex implementation
- Can introduce timing uncertainties (non-deterministic behavior)
- Potential for interrupt conflicts and priority issues
- Overhead for context switching

### Configuring Interrupt-Driven I/O

Step 1: Configure interrupt sources

Enable specific interrupt sources in peripherals.

Step 2: Configure NVIC

Set up the Nested Vectored Interrupt Controller for the interrupts.

Step 3: Implement ISRs

Create Interrupt Service Routines to handle specific events.

Step 4: Enable global interrupts

Enable the global interrupt flag.

```
1 // Step 1: Configure SPI interrupt
2 void configure_spi_interrupt(void) {
3     // Enable SPI peripheral clock
4     RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;
5
6     // Configure SPI parameters
7     SPI1->CR1 = SPI_CR1_MSTR | SPI_CR1_SSI | SPI_CR1_SSM;
8
9     // Enable SPI TX buffer empty interrupt
10    SPI1->CR2 |= SPI_CR2_TXEIE;
11
12    // Step 2: Configure NVIC for SPI1
13    NVIC_SetPriority(SPI1_IRQn, 2); // Set priority level
14    NVIC_EnableIRQ(SPI1_IRQn); // Enable interrupt in NVIC
15 }
16
17 // Step 3: Implement SPI1 ISR
18 void SPI1_IRQHandler(void) {
19     // Check if TX buffer empty interrupt
20     if (SPI1->SR & SPI_SR_TXE) {
21         if (spi_tx_count < spi_tx_length) {
22             // Send next byte
23             SPI1->DR = spi_tx_buffer[spi_tx_count++];
24         } else {
25             // Transfer complete, disable interrupt
26             SPI1->CR2 &= ~SPI_CR2_TXEIE;
27         }
28     }
29 }
30
31 // Main function
32 int main(void) {
33     // Initialize system
34     system_init();
35
36     // Configure SPI interrupt
37     configure_spi_interrupt();
38
39     // Step 4: Enable global interrupts
40     __enable_irq();
41
42     // Main loop - can perform other tasks
43     while (1) {
44         process_system_tasks();
45     }
46 }
```

# Interrupt Performance Analysis

## Interrupt Performance Metriken

Wichtige Kenngrößen zur Bewertung von Interrupt-Systemen:

- **Interrupt-Frequenz**  $f_{INT}$  [Hz]: Wie oft tritt ein Interrupt auf?
- **Interrupt Service Time**  $t_{ISR}$  [s]: Wie lange dauert die ISR-Ausführung?
- **Impact** [%]: Welcher Anteil der CPU-Zeit wird für Interrupts verwendet?

## Impact Berechnung

Der Einfluss von Interrupts auf die Systemleistung:

$$\text{Impact} = f_{INT} \times t_{ISR} \times 100\%$$

Dabei ist:

- $f_{INT}$ : Interrupt-Frequenz in Hz
- $t_{ISR}$ : Interrupt Service Time in Sekunden
- Impact: Prozentuale CPU-Belastung durch Interrupts

## Interrupt Performance Analyse

Schritt 1: Interrupt-Frequenz bestimmen

- Datenrate und Puffergröße analysieren
- $f_{INT} = \frac{\text{Datenrate [bit/s]}}{\text{Puffergröße [bit]}}$

Schritt 2: Service Time berechnen

- Anzahl Clockzyklen für ISR bestimmen
- $t_{ISR} = \frac{\text{Clockzyklen}}{\text{CPU-Frequenz [Hz]}}$

Schritt 3: Impact berechnen

- $\text{Impact} = f_{INT} \times t_{ISR} \times 100\%$
- Prüfung: Impact < 100% für stabilen Betrieb

Schritt 4: Kritische Datenrate bestimmen

- Für 100% CPU-Auslastung:  $f_{INT} \times t_{ISR} = 1$
- Maximale Datenrate:  $\text{Rate}_{max} = \frac{\text{Puffergröße}}{t_{ISR}}$

## Interrupt Performance Berechnung

Ein Prozessorsystem (1 MHz Takt) empfängt Daten mit 16 kbit/s. Das Peripheriegerät kann 32 bit zwischenspeichern. Die ISR benötigt 100 Clockzyklen.

### Loesung:

#### a) Impact berechnen:

- $f_{INT} = \frac{16 \text{ kbit/s}}{32 \text{ bit}} = 500 \text{ Hz}$
- $t_{ISR} = \frac{100 \text{ Zyklen}}{1 \text{ MHz}} = 100 \mu\text{s}$
- $\text{Impact} = 500 \text{ Hz} \times 100 \mu\text{s} \times 100\% = 5\%$

#### b) Kritische Datenrate für 100% CPU-Last:

- $(x/32 \text{ bit}) \times 100 \mu\text{s} = 1$
- $x = \frac{32 \text{ bit}}{100 \mu\text{s}} = 320 \text{ kbit/s}$

#### c) Datenverlust bei 90% Last:

ISR-Zeit schwankt je nach Instruktion und Daten ⇒ Peaks können zu Datenverlust führen.

## Key Performance Metrics

Several metrics characterize interrupt performance:

- **Interrupt Frequency** ( $f_{INT}$ ): How often an interrupt occurs (events per second)
- **Interrupt Service Time** ( $t_{ISR}$ ): Time required to process an interrupt
- **Interrupt Latency**: Time between interrupt event and start of ISR execution
- **System Impact**: Percentage of CPU time spent handling interrupts

## System Impact Calculation

The percentage of CPU time consumed by interrupts can be calculated as:

$$\text{Impact (\%)} = f_{INT} \times t_{ISR} \times 100\% \quad (5)$$

Examples:

- Keyboard interrupt:  $f_{INT} = 20 \text{ Hz}$ ,  $t_{ISR} = 6 \mu\text{s}$

$$\text{Impact} = 20 \text{ Hz} \times 6 \mu\text{s} \times 100\% = 0.012\% \quad (6)$$

- High-speed serial interface:  $f_{INT} = 28,800 \text{ Hz}$ ,  $t_{ISR} = 6 \mu\text{s}$

$$\text{Impact} = 28,800 \text{ Hz} \times 6 \mu\text{s} \times 100\% = 17.3\% \quad (7)$$

## Interrupt Overload Conditions

Interrupt overload occurs when the system cannot keep up with incoming interrupts:

- When  $t_{ISR} >$  time between interrupt events
- Some interrupt events will be missed or delayed
- Data may be lost
- System may become unresponsive to new interrupts

Factors that may lead to overload:

- Too many interrupt sources
- Interrupt sources that trigger too frequently
- ISRs that take too long to execute
- Varying interrupt frequencies causing bursts
- Improper interrupt priority management

## Optimizing ISR Performance

Keep ISRs short

Perform only time-critical operations in the ISR.

Defer processing to main loop

Use flags, queues, or buffers to pass data to main loop.

Use hardware features

Take advantage of DMA and peripheral buffers.

Apply appropriate priorities

Assign higher priorities to more time-critical interrupts.

Disable interrupts judiciously

Disable interrupts only when necessary, for as short a time as possible.

```
1 // Non-optimized ISR
2 void USART2_IRQHandler(void) {
3     if (USART2->SR & USART_SR_RXNE) {
4         // Directly process data in ISR (slow)
5         char c = USART2->DR;
6         process_character(c); // Time-consuming
7         update_display(); // Even more time-consuming
8     }
9 }
10
11 // Optimized ISR
12 volatile uint8_t rx_buffer[256];
13 volatile uint8_t rx_write_idx = 0;
14 volatile uint8_t rx_read_idx = 0;
15 volatile bool new_data_available = false;
16
17 void USART2_IRQHandler(void) {
18     if (USART2->SR & USART_SR_RXNE) {
19         // Only store data in buffer (fast)
20         rx_buffer[rx_write_idx++] = USART2->DR;
21         new_data_available = true;
22     }
23 }
24
25 // Process data in main loop
26 void main(void) {
27     // Initialize
28     system_init();
29
30     while (1) {
31         if (new_data_available) {
32             // Process all received data
33             while (rx_read_idx != rx_write_idx) {
34                 process_character(rx_buffer[rx_read_idx++]);
35             }
36
37             // Update display once after processing all characters
38             update_display();
39
40             // Reset flag
41             new_data_available = false;
42         }
43
44         // Other system tasks
45         process_system_tasks();
46     }
47 }
```

## Interrupt Latency

### Interrupt Latency

Interrupt latency is the time between an interrupt event and the first useful instruction execution in the ISR:

- **Hardware Latency:** Time for hardware to detect and signal the event
- **Arbitration Latency:** Time to determine which interrupt to service (if multiple)
- **CPU Latency:** Time to complete current instruction and save context
- **OS/Software Latency:** Additional delays due to software overhead

Latency is critical for real-time systems, where guaranteed response times are required.

### Sources of Interrupt Latency

Several factors contribute to interrupt latency:

- **Current CPU Instruction:**
  - Multi-cycle instructions may complete before the interrupt is serviced
  - Some instructions may be abandoned and restarted (e.g., SDIV/UDIV on Cortex-M3/M4)
  - Some may be interrupted and resumed (e.g., LDM/STM on Cortex-M3/M4)
- **Disabled Interrupts:**
  - Global interrupts may be disabled (CPSID i / CPSIE i)
  - Specific interrupts may be masked
- **Higher Priority Interrupts:**
  - Lower priority interrupts wait until higher priority ones complete
- **Context Saving:**
  - Pushing registers to stack
  - More registers saved means higher latency
- **Cache and Memory Behavior:**
  - Cache misses when fetching ISR code
  - Memory wait states

## Managing Interrupt Latency

Use interrupt priorities

Assign appropriate priorities based on timing requirements.

Limit interrupt disable periods

Minimize sections where interrupts are disabled.

Optimize context switching

Use processor features that minimize context save/restore.

Preempt lower-priority tasks

Allow high-priority interrupts to preempt less critical ones.

Move waiting loops to main program

Don't block inside ISRs waiting for slow peripherals.

```
1 // Poor: Blocking in ISR
2 void SPI1_IRQHandler(void) {
3     // Read data from input
4     input_data = read_input_source();
5
6     // Process data
7     processed_data = process_data(input_data);
8
9     // Wait for output device to be ready (blocking)
10    while (!(SPI1->SR & SPI_SR_TXE)) { }
11
12    // Write data to output
13    SPI1->DR = processed_data;
14}
15
16 // Better: Queue-based approach
17 void SPI1_IRQHandler(void) {
18     // Read data from input
19     input_data = read_input_source();
20
21     // Process data
22     processed_data = process_data(input_data);
23
24     // Add to output queue and return
25     queue_add(output_queue, processed_data);
26}
27
28 // In main loop
29 void main(void) {
30     // Initialize
31     system_init();
32
33     while (1) {
34         // Check if there's data to send and SPI is ready
35         if (!queue_empty(output_queue) && (SPI1->SR & SPI_SR_TXE)) {
36             // Send next item
37             SPI1->DR = queue_get(output_queue);
38         }
39
40         // Other tasks
41         process_system_tasks();
42     }
43 }
```

## Pre-emption and Priority

### Nested Vector Interrupt Controller (NVIC)

The ARM Cortex-M NVIC provides advanced interrupt handling features:

- Supports up to 240 external interrupt sources (IRQs)
- Each interrupt can be assigned one of 256 priority levels (processor specific)
- Priority-based interrupt preemption
- Automatic context saving and restoring
- Tail-chaining optimization (reduced latency between ISRs)
- Late-arrival handling (higher priority interrupts can overtake pending lower ones)

### Interrupt Preemption

Preemption allows higher-priority interrupts to interrupt lower-priority ones:

- When a higher-priority interrupt occurs during a lower-priority ISR
- Current ISR is suspended (context saved)
- Higher-priority ISR executes
- After completion, lower-priority ISR resumes

This ensures that critical interrupts are handled promptly, regardless of other interrupt activity.

### Priority-Based Interrupt Handling

Key aspects of priority-based interrupt handling:

- **Priority Assignment:**
  - Assign priorities based on time-criticality and importance
  - Lower numerical values usually indicate higher priority
- **Priority Grouping:**
  - Cortex-M supports priority grouping into pre-emption and sub-priority levels
  - Pre-emption priority determines whether an interrupt can pre-empt another
  - Sub-priority determines ordering when multiple interrupts of same pre-emption priority occur
- **Priority Inversion:**
  - Problem where a high-priority task is indirectly delayed by a low-priority task
  - Can be mitigated through proper design and priority inheritance protocols

## Configuring Interrupt Priorities

Step 1: Analyze timing requirements

Determine which interrupts are most time-critical.

Step 2: Group interrupts by importance

Create logical groups based on system criticality.

Step 3: Assign priority levels

Configure NVIC priority registers for each interrupt.

Step 4: Configure priority grouping

Set priority group using NVIC\_SetPriorityGrouping().

```
1 // Configure interrupt priorities
2 void configure_interrupt_priorities(void) {
3     // Set priority grouping (4 bits for pre-emption priority, 0 bits for
4     // sub-priority)
5     NVIC_SetPriorityGrouping(3);
6
7     // Critical interrupts (highest priority)
8     NVIC_SetPriority(EXTIO IRQn, 0);          // Emergency stop button
9     NVIC_SetPriority(TIM1_UP IRQn, 1);        // Critical timing control
10
11    // Medium priority interrupts
12    NVIC_SetPriority(SPI1 IRQn, 4);          // Sensor data acquisition
13    NVIC_SetPriority(USART2 IRQn, 5);         // Communication
14
15    // Lower priority interrupts
16    NVIC_SetPriority(ADC IRQn, 10);           // Regular ADC sampling
17    NVIC_SetPriority(TIM3 IRQn, 11);           // Status LED updates
18
19    // Enable the interrupts
20    NVIC_EnableIRQ(EXTIO IRQn);
21    NVIC_EnableIRQ(TIM1_UP IRQn);
22    NVIC_EnableIRQ(SPI1 IRQn);
23    NVIC_EnableIRQ(USART2 IRQn);
24    NVIC_EnableIRQ(ADC IRQn);
25    NVIC_EnableIRQ(TIM3 IRQn);
}
```

## Interrupt-Driven State Machines

### Interrupt-driven Finite State Machine

Kombination von Interrupt-System und Zustandsautomat:

- ISRs generieren Events und schreiben sie in Queue
- FSM in Main Loop liest Events aus Queue
- Deterministische Event-Verarbeitung
- Saubere Trennung von Interrupt- und Anwendungslogik

### Interrupt-driven FSM Struktur

```
1 // Main Loop
2 int main(void) {
3     event_t event;
4     fsm_init();
5     peripherals_init();
6
7     while (1) {
8         event = get_event_from_queue();
9         if (event != NO_EVENT) {
10             fsm_handle_event(event);
11         }
12     }
13 }
14
15 // Interrupt Service Routines
16 void ISR_Peripheral1(void) {
17     // Event in Queue schreiben
18     write_event_to_queue(EVENT_PERIPHERAL1);
19     // Interrupt Flag löschen
20     clear_interrupt_flag();
21 }
```

**Interrupt Performance Problem** System mit 90% Interrupt-Last verliert trotzdem Daten. Erklären Sie eine plausible Ursache.

### Lösung:

Die Interrupt Service Time ist nicht konstant:

- **Variable Latency:** Je nach aktueller CPU-Instruktion
- **Datenabhängige ISR-Dauer:** Unterschiedliche Verarbeitungszeiten
- **Jitter:** Schwankungen in der Interrupt-Behandlung
- **Preemption:** Andere Interrupts können verzögern

Bei 90% Durchschnittslast können Spitzen > 100% erreichen ⇒ Datenverlust.

### Ashilfe:

- Puffergröße erhöhen
- ISR-Zeit reduzieren und konstanter machen
- Sicherheitsmarge bei Systemauslegung einplanen

### Faustregeln für Interrupt-Systeme:

- Impact sollte < 50% bleiben für stabile Performance
- ISR-Zeit sollte konstant und vorhersagbar sein
- Kritische Daten immer puffern (Hardware oder Software)
- Bei mehreren Interrupt-Quellen: Prioritäten sorgfältig wählen

### Integrating Interrupts and State Machines

Combining interrupt-driven I/O with state machines creates efficient event-driven systems:

- **Event Queue:** Buffer between ISRs and state machine
  - ISRs detect events and add them to queue
  - Main loop pulls events from queue and feeds state machine
- **ISR Responsibilities:**
  - Minimal processing (detect event, capture data)
  - Add event to queue
  - Return quickly
- **Main Loop Responsibilities:**
  - Pull events from queue
  - Process events through state machine
  - Handle longer-duration processing

### Implementing Interrupt-Driven FSM

Step 1: Create event queue

Implement a buffer to store events from ISRs.

Step 2: Implement ISRs

Keep ISRs short, just capturing events and adding to queue.

Step 3: Implement state machine

Create state machine logic to process events from queue.

Step 4: Connect with main loop

Pull events from queue and feed to state machine in main loop.

```
1 // Define event types and queue
2 typedef enum {
3     EVENT_BUTTON_PRESS,
4     EVENT_TIMEOUT,
5     EVENT_SENSOR_TRIGGER,
6     EVENT_NO_EVENT
7 } event_t;
8
9 #define EVENT_QUEUE_SIZE 16
10 event_t event_queue[EVENT_QUEUE_SIZE];
11 int queue_head = 0;
12 int queue_tail = 0;
13 int queue_count = 0;
14
15 // Add event to queue (called from ISRs)
16 void queue_add_event(event_t event) {
17     if (queue_count < EVENT_QUEUE_SIZE) {
18         event_queue[queue_tail] = event;
19         queue_tail = (queue_tail + 1) % EVENT_QUEUE_SIZE;
20         queue_count++;
21     }
22 }
23
24 // Get event from queue (called from main loop)
25 event_t queue_get_event(void) {
26     event_t event = EVENT_NO_EVENT;
27
28     if (queue_count > 0) {
29         event = event_queue[queue_head];
30         queue_head = (queue_head + 1) % EVENT_QUEUE_SIZE;
31         queue_count--;
32     }
33
34     return event;
35 }
```

## Part 2: Implementing the State Machine

```

1 // Button interrupt handler
2 void EXTI0_IRQHandler(void) {
3     // Check if EXTI0 interrupt occurred
4     if (EXTI->PR & EXTI_PR_PRO) {
5         // Add button press event to queue
6         queue_add_event(EVENT_BUTTON_PRESS);
7
8         // Clear pending bit
9         EXTI->PR = EXTI_PR_PRO;
10    }
11 }
12
13
14 // Timer interrupt handler
15 void TIM2_IRQHandler(void) {
16     if (TIM2->SR & TIM_SR UIF) {
17         // Add timeout event to queue
18         queue_add_event(EVENT_TIMEOUT);
19
20         // Clear update interrupt flag
21         TIM2->SR &= ~TIM_SR UIF;
22    }
23 }
24
25 // Main loop with state machine
26 int main(void) {
27     // Initialize system and interrupts
28     system_init();
29     __enable_irq();
30
31     // Initialize state machine
32     fsm_init();
33
34     while (1) {
35         // Get event from queue
36         event_t event = queue_get_event();
37
38         // Process event with state machine
39         if (event != EVENT_NO_EVENT) {
40             fsm_handle_event(event);
41         }
42
43         // Background tasks
44         process_system_tasks();
45     }
46 }
```

## Complete Interrupt-Driven FSM Example

Implement a button-controlled LED system with debouncing using timer interrupts.

The system has three states: - OFF: LED is off - ON: LED is on at full brightness - BLINK: LED is blinking

Events: - SHORT\_PRESS: Button pressed briefly - LONG\_PRESS: Button held for >1 second - BLINK\_TIMER: Timer for blinking

```

1 // Event definitions
2 typedef enum {
3     EVENT_NONE,
4     EVENT_SHORT_PRESS,
5     EVENT_LONG_PRESS,
6     EVENT_BLINK_TIMER
7 } event_t;
8
9 // State definitions
10 typedef enum {
11     STATE_OFF,
12     STATE_ON,
13     STATE_BLINK
14 } state_t;
15
16 // Global variables
17 volatile event_t event_queue[10];
18 volatile uint8_t queue_head = 0;
19 volatile uint8_t queue_tail = 0;
20 volatile uint8_t queue_count = 0;
21 volatile state_t current_state = STATE_OFF;
22 volatile uint32_t button_press_time = 0;
23 volatile uint8_t button_released = 1;
24
25 // Button interrupt handler (EXTI0)
26 void EXTI0_IRQHandler(void) {
27     uint32_t current_time = HAL_GetTick();
28
29     if (EXTI->PR & EXTI_PR_PRO) {
30         if (GPIOA->IDR & GPIO_PIN_0) {
31             // Button released
32             if (!button_released) {
33                 uint32_t press_duration = current_time - button_press_time;
34                 if (press_duration > 1000) {
35                     // Long press (>1s)
36                     if (queue_count < 10) {
37                         event_queue[queue_tail] = EVENT_LONG_PRESS;
38                         queue_tail = (queue_tail + 1) % 10;
39                         queue_count++;
40                     }
41                 } else if (press_duration > 50) {
42                     // Short press (debounced)
43                     if (queue_count < 10) {
44                         event_queue[queue_tail] = EVENT_SHORT_PRESS;
45                         queue_tail = (queue_tail + 1) % 10;
46                         queue_count++;
47                     }
48                     button_released = 1;
49                 }
50             } else {
51                 // Button pressed
52                 button_press_time = current_time;
53                 button_released = 0;
54             }
55
56             // Clear pending bit
57             EXTI->PR = EXTI_PR_PRO;
58         }
59     }
60 }
```

### Complete Interrupt-Driven FSM Example (continued)

```

1 // Timer interrupt for blinking (TIM3)
2 void TIM3_IRQHandler(void) {
3     if (TIM3->SR & TIM_SR UIF) {
4         // Add blink timer event to queue
5         if (queue_count < 10) {
6             event_queue[queue_tail] = EVENT_BLINK_TIMER;
7             queue_tail = (queue_tail + 1) % 10;
8             queue_count++;
9         }
10
11     // Clear update interrupt flag
12     TIM3->SR &= ~TIM_SR UIF;
13 }
14
15 // State machine handler
16 void fsm_handle_event(event_t event) {
17     switch (current_state) {
18         case STATE_OFF:
19             if (event == EVENT_SHORT_PRESS) {
20                 // Turn on LED
21                 GPIOC->BSRR = GPIO_PIN_13;
22                 current_state = STATE_ON;
23
24                 // Disable blink timer
25                 TIM3->CR1 &= ~TIM_CR1_CEN;
26             } else if (event == EVENT_LONG_PRESS) {
27                 // Start blinking
28                 GPIOC->BSRR = GPIO_PIN_13;
29                 current_state = STATE_BLINK;
30
31                 // Enable blink timer
32                 TIM3->CR1 |= TIM_CR1_CEN;
33             }
34             break;
35
36         case STATE_ON:
37             if (event == EVENT_SHORT_PRESS) {
38                 // Turn off LED
39                 GPIOC->BSRR = (GPIO_PIN_13 << 16);
40                 current_state = STATE_OFF;
41             } else if (event == EVENT_LONG_PRESS) {
42                 // Start blinking
43                 current_state = STATE_BLINK;
44
45                 // Enable blink timer
46                 TIM3->CR1 |= TIM_CR1_CEN;
47             }
48             break;
49
50         case STATE_BLINK:
51             if (event == EVENT_BLINK_TIMER) {
52                 // Toggle LED
53                 GPIOC->ODR ^= GPIO_PIN_13;
54             } else if (event == EVENT_SHORT_PRESS) {
55                 // Turn off LED and blinking
56                 GPIOC->BSRR = (GPIO_PIN_13 << 16);
57                 current_state = STATE_OFF;
58
59                 // Disable blink timer
60                 TIM3->CR1 &= ~TIM_CR1_CEN;
61             }
62             break;
63     }
64 }
65

```

### Complete Interrupt-Driven FSM Example (continued)

```

1 // Main function
2 int main(void) {
3     // Initialize hardware
4     system_init();
5
6     // Configure EXTI for button
7     // Configure TIM3 for blinking (500ms period)
8     // Enable interrupts
9
10    while (1) {
11        // Get and process events
12        if (queue_count > 0) {
13            event_t event;
14
15            // Disable interrupts to access queue
16            __disable_irq();
17            event = event_queue[queue_head];
18            queue_head = (queue_head + 1) % 10;
19            queue_count--;
20            __enable_irq();
21
22            // Handle event
23            fsm_handle_event(event);
24        }
25
26        // System can enter low-power mode here
27        // when queue is empty
28    }
29 }
30

```

## Interrupt Performance Exercises

### Interrupt Performance Analysis

#### Analyzing Interrupt System Performance

##### Calculate interrupt frequency

- Identify the source and rate of interrupts
- For periodic interrupts, calculate frequency directly
- For data-driven interrupts (e.g., serial interface), calculate:
  - $f_{INT} = \text{data\_rate} / \text{data\_size\_per\_interrupt}$
  - Example: 9600 baud UART with 8 data bits =  $9600/8 = 1200 \text{ Hz}$

##### Determine interrupt service time

- Estimate execution time of the Interrupt Service Routine (ISR)
- Include:
  - Context switching time (register saving/restoring)
  - Instruction execution time of the routine itself
  - Memory access time
- Measure or calculate in CPU clock cycles, then convert to time
- $t_{ISR} = \text{cycles} / f_{CPU}$

##### Calculate system impact

- Percentage of CPU time used by interrupts:
- Impact =  $f_{INT} \times t_{ISR} \times 100\%$
- Assess if the impact is acceptable:
  - < 10%: Minimal impact
  - 10-50%: Moderate impact
  - > 50%: Significant impact, may need optimization
  - > 90%: System likely unable to perform other tasks
- Check if  $t_{ISR} > 1/f_{INT}$  - indicates missed interrupts

##### Optimize interrupt handling

- Reduce ISR execution time:
  - Keep ISRs short
  - Move non-critical processing to main loop
  - Use efficient algorithms and data structures
- Manage interrupt frequency:
  - Use buffering to reduce interrupt rate
  - Adjust hardware configurations if possible
- Implement interrupt prioritization for critical tasks

### Interrupt System Analysis

A processor system running at 1 MHz receives data through a peripheral interface at a rate of 16 kbit/s. The peripheral can buffer 32 bits of data and signals the processor via an interrupt line when the buffer is ready to be read. If the data is not read before the next interrupt, it is lost.

The Interrupt Service Routine (ISR) requires 100 clock cycles on average, including call and return overhead. The system uses no other interrupts.

- Calculate the impact of interrupts on the system performance.
- Determine the maximum data rate the interface could handle before the processor spends 100% of its time handling interrupts.
- If the data rate is increased such that the system spends 90% of its time handling interrupts, occasional data loss still occurs. Explain a possible cause.

#### Solution:

##### 1. Calculate interrupt impact:

- Interrupt frequency:  $f_{INT} = 16 \text{ kbit/s} \div 32 \text{ bits} = 500 \text{ Hz}$
- Interrupt service time:  $t_{ISR} = 100 \text{ cycles} \div 1 \text{ MHz} = 100 \mu\text{s}$
- Impact =  $f_{INT} \times t_{ISR} \times 100\% = 500 \text{ Hz} \times 100 \mu\text{s} \times 100\% = 5\%$

Therefore, interrupts consume 5% of the CPU's processing time.

##### 2. Maximum data rate calculation:

- For 100% CPU usage:  $1 = f_{INT} \times t_{ISR}$
- $f_{INT} = 1 \div t_{ISR} = 1 \div 100 \mu\text{s} = 10,000 \text{ Hz}$
- Data rate =  $f_{INT} \times 32 \text{ bits} = 10,000 \times 32 = 320 \text{ kbit/s}$

##### 3. Cause of occasional data loss:

- At 90% CPU utilization, there should theoretically be enough time to process all interrupts
- However, the service time ( $t_{ISR}$ ) is not constant but an average
- Some interrupt instances may take longer than the average 100 cycles
- When  $t_{ISR}$  varies, occasional peaks can exceed the time between interrupts
- The variation may come from:
  - Different code paths within the ISR depending on the data
  - Memory access times that vary due to cache effects
  - Context switching overhead that varies based on CPU state
  - Interrupt latency variations based on what instruction was executing
- When a single interrupt takes too long, the next interrupt arrives before processing is complete
- With high CPU utilization (90%), there's little margin for these variations
- Result: occasional buffer overflows and data loss

# Interrupt Latency Optimization

## Managing Interrupt Latency

### Understand latency components

- **Hardware latency:**

- Time for interrupt controller to recognize interrupt
- Time to finish current instruction (multi-cycle instructions)
- Time to push registers onto stack

- **Software latency:**

- Interrupt disabled periods
- Higher priority interrupt processing
- Cache misses during ISR execution

### Measure interrupt latency

- Use an oscilloscope to measure time between:
  - Interrupt trigger signal
  - GPIO pin toggle in ISR
- Alternatively, use a timer to capture timestamps:
  - Configure timer to capture on interrupt signal
  - Read timer value at ISR entry
  - Calculate difference
- Measure best-case, worst-case, and average latency

### Optimize for consistent latency

- Minimize interrupt disable periods
  - Keep critical sections short
  - Use selective interrupt masking instead of global disable
- Optimize ISR code
  - Minimize stack usage to reduce push/pop operations
  - Keep ISRs short and efficient
  - Ensure frequently used data and code are in cache
- Use appropriate priorities
  - Assign higher priorities to time-critical interrupts
  - Group interrupts with similar timing requirements

### Implement deterministic response

- For time-critical operations, consider:
  - Polling instead of interrupts if determinism is crucial
  - Hardware solutions (DMA, dedicated controllers)
  - Real-time operating system with predictable scheduling
- Move processing from ISRs to main loop
  - ISR only captures essential data and signals main loop
  - Use flags, semaphores, or message queues for communication

## Interrupt Latency Optimization

A microcontroller-based system needs to sample an analog signal at precise 100 µs intervals. The current implementation uses Timer1 to generate interrupts every 100 µs, and the ISR reads the ADC. Measurements show that the interrupt latency varies between 5-20 µs, causing jitter in the sampling.

Design an improved solution to minimize sampling jitter, explaining your approach and implementation.

### Solution:

The goal is to reduce sampling jitter by minimizing variations in the time between the timer interrupt and ADC sampling. Here's an improved solution:

#### Hardware-triggered ADC approach:

1. Use timer to trigger ADC directly:

- Configure Timer1 to generate a trigger signal every 100 µs
- Connect this trigger to the ADC hardware trigger input
- Configure the ADC to start conversion automatically on timer trigger
- Use DMA to transfer ADC results to memory without CPU intervention

2. Implementation details:

```
1 // Configure Timer1
2 void configure_timer(void) {
3     // Enable clock for Timer1
4     RCC->APB2ENR |= RCC_APB2ENR_TIM1EN;
5
6     // Set prescaler and period for 100 microsecond interval
7     TIM1->PSC = (SystemCoreClock / 1000000) - 1; // 1 microsecond timer ticks
8     TIM1->ARR = 100 - 1; // 100 microsecond period
9
10    // Configure timer to generate trigger output
11    TIM1->CR2 &= ~TIM_CR2_MMS;
12    TIM1->CR2 |= TIM_CR2_MMS_1; // Update event as trigger output
13
14    // Enable timer
15    TIM1->CR1 |= TIM_CR1_CEN;
16 }
17
18 // Configure ADC with timer trigger
19 void configure_adc(void) {
20     // Enable clocks for ADC and GPIO
21     RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;
22     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
23
24     // Configure GPIO pin as analog input
25     GPIOA->MODER |= GPIO_MODER_MODE0; // PA0 as analog
26
27     // Configure ADC for timer trigger
28     ADC1->CR2 |= ADC_CR2_EXTEN_0; // Enable external trigger on rising edge
29     ADC1->CR2 |= ADC_CR2_EXTSEL_3; // Select Timer1 TRGO as trigger
30
31     // Configure single conversion on channel 0
32     ADC1->SQR3 = 0; // Channel 0 as first conversion
33     ADC1->SQR1 = 0; // 1 conversion
34
35     // Enable DMA for ADC
36     ADC1->CR2 |= ADC_CR2_DMA;
37
38     // Enable ADC
39     ADC1->CR2 |= ADC_CR2_ADON;
40 }
```

## Complete Interrupt-Driven FSM Example (continued)

```
1 // Configure DMA for ADC
2 void configure_dma(uint16_t* buffer, uint32_t buffer_size) {
3     // Enable DMA clock
4     RCC->AHB1ENR |= RCC_AHB1ENR_DMA2EN;
5
6     // Configure DMA stream
7     DMA2_Stream0->CR &= ~DMA_SxCR_EN; // Disable DMA during config
8
9     DMA2_Stream0->PAR = (uint32_t)&ADC1->DR; // Source: ADC data register
10    DMA2_Stream0->MOAR = (uint32_t)buffer; // Destination: buffer
11    DMA2_Stream0->NDTR = buffer_size; // Number of data items
12
13    DMA2_Stream0->CR &= ~DMA_SxCR_DIR; // Peripheral to memory
14    DMA2_Stream0->CR |= DMA_SxCR_CIRC; // Circular mode
15    DMA2_Stream0->CR |= DMA_SxCR_PSIZE_0; // Peripheral size: 16 bits
16    DMA2_Stream0->CR |= DMA_SxCR_MSIZE_0; // Memory size: 16 bits
17    DMA2_Stream0->CR |= DMA_SxCR_PL_1; // Priority: high
18
19    // Enable DMA
20    DMA2_Stream0->CR |= DMA_SxCR_EN;
21 }
22
23 // Main application
24 int main(void) {
25     // Sample buffer
26     uint16_t adc_buffer[1000];
27
28     // Configure peripherals
29     configure_timer();
30     configure_adc();
31     configure_dma(adc_buffer, 1000);
32
33     // Process samples in background
34     while (1) {
35         // Process captured samples from adc_buffer
36         // Buffer is continuously updated by DMA
37     }
38 }
```

### 3. Advantages of this approach:

- Eliminates interrupt latency completely from the sampling process
- Hardware triggers ADC directly with precise timing
- DMA transfers data without CPU intervention
- CPU is free to process samples when available
- Sampling jitter reduced to hardware timing accuracy (typically sub- $\mu$ s)

### 4. Alternative approaches:

- If hardware triggering is not available, optimize the ISR:
  - Set ISR to highest priority
  - Disable all other interrupts during the ISR
  - Make the ISR as short as possible (just start ADC)
  - Use a state variable to track if previous conversion completed
- Use a high-precision external timer dedicated to ADC triggering

This solution eliminates software-induced jitter by leveraging hardware synchronization between the timer and ADC, providing the precise 100  $\mu$ s sampling interval required.

## Interrupt-Driven System Design

### Designing Efficient Interrupt-Driven Systems

Choose between polling and interrupts

#### • Use interrupts when:

- Events occur infrequently or at unpredictable times
- System needs to respond to external events quickly
- Power efficiency is important (can sleep between events)
- Multiple event sources need to be monitored simultaneously

#### • Use polling when:

- Events occur at very high frequency
- Absolute determinism is required
- Interrupt overhead would be excessive
- Extremely low latency is required

Design efficient ISRs

#### • Keep ISRs as short as possible

- Only handle time-critical operations in the ISR
- Defer processing to main loop using flags or queues
- Avoid complex calculations or I/O operations

#### • Manage shared resources

- Use volatile for variables shared between ISR and main code
- Consider atomic operations for simple updates
- Implement proper synchronization for complex data structures

#### • Avoid nesting or recursion in ISRs

Implement event queues

#### • Use queue to transfer events from ISRs to main loop

- ISR detects event and enqueues it
- Main loop dequeues and processes events

#### • Design queue to be interrupt-safe

- Use atomic operations if available
- Consider implementing lock-free algorithms
- Size queue appropriately to avoid overflow

#### • Process events in order of importance or arrival

Implement event-driven architecture

#### • Structure code as state machines responding to events

#### • Separate event detection (ISRs) from processing (handlers)

#### • Design for different operating modes

- Normal operation
- Power-saving modes
- Error handling and recovery

#### • Consider using an RTOS for complex systems

## Interrupt-Driven UART Communication System

Design an interrupt-driven communication system with the following requirements:

- UART interface operating at 115200 baud, 8N1 format
- Must handle both transmit and receive operations
- System should never lose incoming data
- Main application should not be blocked by I/O operations
- CPU runs at 72 MHz and has other tasks to perform

Solution:

The system will use an interrupt-driven approach with queues to handle UART communication efficiently:

### 1. System architecture:

- Receive interrupt (RXNE) to handle incoming data
- Transmit interrupt (TXE) to send outgoing data
- Circular buffers for both TX and RX data
- Non-blocking API for the main application

### 2. Implementation:

```
1 #include <stdint.h>
2 #include <stdbool.h>
3
4 // Buffer size must be power of 2 for efficient wrap-around
5 #define UART_BUFFER_SIZE 256
6 #define UART_BUFFER_MASK (UART_BUFFER_SIZE - 1)
7
8 // Circular buffer structure
9 typedef struct {
10     volatile uint8_t data[UART_BUFFER_SIZE];
11     volatile uint16_t head;
12     volatile uint16_t tail;
13 } circular_buffer_t;
14
15 // Global buffers
16 static circular_buffer_t rx_buffer = {0};
17 static circular_buffer_t tx_buffer = {0};
18 static volatile bool tx_busy = false;
19
20 // Function prototypes
21 void uart_init(uint32_t baudrate);
22 bool uart_send_byte(uint8_t data);
23 bool uart_send_data(const uint8_t* data, uint16_t length);
24 bool uart_read_byte(uint8_t* data);
25 uint16_t uart_available(void);
26 void uart_flush(void);
27
28 // External hardware functions
29 extern void uart_hw_init(uint32_t baudrate);
30 extern void uart_hw_send_byte(uint8_t data);
31 extern uint8_t uart_hw_read_byte(void);
32 extern bool uart_hw_tx_empty(void);
33 extern bool uart_hw_rx_ready(void);
34 extern void uart_hw_enable_tx_interrupt(bool enable);
35 extern void uart_hw_enable_rx_interrupt(bool enable);
```

## Complete Interrupt-Driven FSM Example (continued)

```
1 // Buffer operations
2 static bool buffer_is_full(const circular_buffer_t* buffer) {
3     return ((buffer->head + 1) & UART_BUFFER_MASK) == buffer->tail;
4 }
5
6 static bool buffer_is_empty(const circular_buffer_t* buffer) {
7     return buffer->head == buffer->tail;
8 }
9
10 static bool buffer_push(circular_buffer_t* buffer, uint8_t data) {
11     uint16_t next_head = (buffer->head + 1) & UART_BUFFER_MASK;
12
13     if (next_head == buffer->tail) {
14         return false; // Buffer full
15     }
16
17     buffer->data[buffer->head] = data;
18     buffer->head = next_head;
19     return true;
20 }
21
22 static bool buffer_pop(circular_buffer_t* buffer, uint8_t* data) {
23     if (buffer->head == buffer->tail) {
24         return false; // Buffer empty
25     }
26
27     *data = buffer->data[buffer->tail];
28     buffer->tail = (buffer->tail + 1) & UART_BUFFER_MASK;
29     return true;
30 }
31
32 static uint16_t buffer_count(const circular_buffer_t* buffer) {
33     return (buffer->head - buffer->tail) & UART_BUFFER_MASK;
34 }
35
36 // UART initialization
37 void uart_init(uint32_t baudrate) {
38     // Initialize hardware
39     uart_hw_init(baudrate);
40
41     // Reset buffers
42     rx_buffer.head = rx_buffer.tail = 0;
43     tx_buffer.head = tx_buffer.tail = 0;
44     tx_busy = false;
45
46     // Enable receive interrupt
47     uart_hw_enable_rx_interrupt(true);
48     // Transmit interrupt will be enabled when data is available
49 }
50 }
```

## Complete Interrupt-Driven FSM Example (continued)

```

1 // Send a single byte (non-blocking)
2 bool uart_send_byte(uint8_t data) {
3     bool success;
4     bool start_transmission = false;
5
6     // Critical section: disable interrupts
7     __disable_irq();
8
9     success = buffer_push(&tx_buffer, data);
10
11    // If TX is not busy and we successfully added data, start transmission
12    if (success && !tx_busy) {
13        start_transmission = true;
14        tx_busy = true;
15    }
16
17    // End critical section
18    __enable_irq();
19
20    if (start_transmission) {
21        uart_hw_enable_tx_interrupt(true);
22    }
23
24    return success;
25}
26
27 // Send multiple bytes (non-blocking)
28 bool uart_send_data(const uint8_t* data, uint16_t length) {
29    for (uint16_t i = 0; i < length; i++) {
30        if (!uart_send_byte(data[i])) {
31            return false; // Buffer full
32        }
33    }
34    return true;
35}
36
37 // Read a single byte (non-blocking)
38 bool uart_read_byte(uint8_t* data) {
39    return buffer_pop(&rx_buffer, data);
40}
41
42 // Get number of bytes available to read
43 uint16_t uart_available(void) {
44    return buffer_count(&rx_buffer);
45}
46
47 // Flush TX buffer
48 void uart_flush(void) {
49    // Wait until TX buffer is empty
50    while (!buffer_is_empty(&tx_buffer)) {
51        __NOP(); // No operation - just wait
52    }
53
54    // Wait until last byte is transmitted
55    while (tx_busy) {
56        __NOP();
57    }
58}
59
60 // RX interrupt handler
61 void UART_RX_IRQHandler(void) {
62    if (uart_hw_rx_ready()) {
63        uint8_t data = uart_hw_read_byte();
64
65        // If buffer full, data will be lost
66        buffer_push(&rx_buffer, data);
67    }
68}

```

## Complete Interrupt-Driven FSM Example (continued)

```

1 // TX interrupt handler
2 void UART_TX_IRQHandler(void) {
3    if (uart_hw_tx_empty()) {
4        uint8_t data;
5
6        if (buffer_pop(&tx_buffer, &data)) {
7            // Send next byte
8            uart_hw_send_byte(data);
9        } else {
10            // No more data, disable TX interrupt
11            uart_hw_enable_tx_interrupt(false);
12            tx_busy = false;
13        }
14    }
15}
16
17 // Example usage in main application
18 void main(void) {
19    // Initialize UART
20    uart_init(115200);
21
22    // Main loop
23    while (1) {
24        // Check for received data
25        if (uart_available() > 0) {
26            uint8_t data;
27            uart_read_byte(&data);
28
29            // Process received data
30            process_data(data);
31
32            // Echo back
33            uart_send_byte(data);
34        }
35
36        // Perform other tasks
37        other_application_tasks();
38    }
39}
40

```

### 3. Key design elements:

- **Circular buffers:** Efficient for handling serial data; no memory copying required
- **Non-blocking API:** Main application can send/receive without waiting
- **Interrupt-driven:** Minimal CPU overhead, responds quickly to data
- **Buffer size is power of 2:** Enables efficient masking for wrap-around
- **Buffer management:** Checks for full/empty conditions to prevent data loss
- **Critical sections:** Protect shared buffer access between ISR and main code

### 4. Performance analysis:

- At 115200 baud, 8N1:
  - 1 start bit + 8 data bits + 1 stop bit = 10 bits per byte
  - Byte rate = 115200/10 = 11520 bytes per second
  - Time between bytes = 1/11520 ≈ 87 μs
- With a 256-byte buffer:
  - Buffer can hold  $256 \times 87 \mu s \approx 22$  ms of data
  - Sufficient for main loop to process data even with other tasks
- ISR execution time:
  - At 72 MHz, reading a byte and storing it takes 10 cycles
  - ISR overhead (entry/exit) 20 cycles
  - Total 30 cycles = 0.42 μs
  - Impact =  $0.42 \mu s / 87 \mu s \approx 0.5\%$  of CPU time

This design provides an efficient, interrupt-driven UART interface that meets all requirements with minimal CPU overhead.