Computertechnik 2

Jil Zerndt
FS 2025

Coding Guidelines

- reduce the number of bugs
- robustness
- correctness
- maintainability
- facilitate code reading within a team
- takes less time to understand another team member's code
- improve portability
- reuse of code on other HW platforms

Enforce by

- automated scans (part of static code checking)
- peer reviews

Appearance

- Indentation: 4 spaces (no tabs)
- Line length: 80 characters
- Only one statement per line (readability)
- No trailing whitespace (no spaces at the end of a line)
- Parentheses to aid clarity (do not rely on operator precedence)
- No magic numbers (use constants instead)

Braces

Non-function statement blocks `if, else, while, for, do, do-while, switch`

```
if (x == y) {
    p = q;
}
```

~~`if (x == y)`~~
~~`    p = q;`~~

```
do {
    body of do-loop
} while (condition);
```

```
if (x == y) {
    ...
} else if (x > y) {
    ...
} else {
    ...
}
```

Function statement blocks `function, class, namespace`

```
int32_t function(int32_t x)
{
    body of function
}
```

`struct, enum:`

```
typedef enum {
    RED,
    GREEN
} colors;
```

```
struct entry {
    uint32_t index,
    uint32_t value
};
```

Spaces

- Mostly function-versus-keyword usage Use space after keywords `if, switch, case, for, do, while`
- No space with sizeof, typeof, alignof, or attribute (as they look somewhat like functions)
  `s = sizeof(struct file);`
- Pointer declaration: * adjacent to data name or function name
  `uint8_t *ptr;`
  `uint32_t parse(uint8_t *ptr, uint8_t **retptr);`
  `uint8_t *match(uint8_t *s);`

Operators

- No space before or after unary operators: ++, –, +, -, !, ~, (type), (cast), sizeof, typeof, alignof, _attribute_, defined
- One space before or after binary/ternary operators: *, /, %, +, -, «, », <, <=, >, >=, ==, !=, ?, :
- No space around struct member operator: . and ->
- No trailing whitespaces!

Functions

- short and sweet (less than 50 lines)
- do just one thing
- no more than 5-10 local variables
- no more than 3 parameters
- function prototypes shall include parameter names with their data types
- no more than 3 levels of indentation (if, for, while, do-while)
- no more than 2 nested loops (for, while, do-while)

Function Parameters Use const to define call-by-reference function parameters that should not be modified - int32_t strlen (const int8_t s[]); strlen() does not modify any character of character array s - void display(mystruct const *param) ;

Function Definition Just one exit point and it shall be at the bottom of the function - keyword return shall appear only once - All 'private' functions shall be defined static - 'private' → Functions that are only used within the module itself. The function is an implementation detail and not accessible from other modules A prototype shall be defined for each 'public' function in the module header file module.h - 'public' → Functions that are called by other modules. The function prototypes are part of the module interface.

Return Values Shall be checked by the caller If the name of a function is an action or an imperative command - Function should return an error-code integer i.e. 0 for success and -Exxx for failure. - If possible error codes shall be based on the Posix Errorcode - If self-defined error codes are being used they shall be properly documented. In the header file for public functions or in the .c file for private functions - For example, ädd workïs a command, and the add_work () function returns 0 for success or -EBUSY for failure.
If the name of a function is a predicate - Function should return a ßucceeded"boolean. - "PCI device presentïs a predicate, and the pci_dev_present () function returns 1 if it succeeds in finding a matching device or 0 if it doesn't. - Functions whose return value is the actual result of a computation, rather than an indication of whether the computation succeeded, are not subject to this rule. - Generally they indicate failure by returning some out-of-range result. - Typical examples would be functions that return pointers; they use NULL or the ERR_PTR mechanism to report failure.

General Rules

Naming No macro name (#define) shall contain any lowercase letters - Function and variable names shall not contain uppercase letters - Use descriptive names for functions, global variables and important local variables - Underscores shall be used to separate words in names e.g. count_active_users() Use short names e.g. i for auxiliary local variables like loop counters - Do not encode types in names. Let the compiler do the type checking

Comments - All comments shall be in English - C99 comments // are allowed - Explain WHAT your code does not HOW - Don't repeat what the statement says in a comment. - Assume that the reader is familiar with C - Comments shall never be nested - All assumptions shall be spelled out in comments - or even better in a set of design-by-contract tests or assertions - The interface of a public function shall be commented next to the function prototype in the header file. - The comment shall not be repeated next to the function definition in the .c file

Types - Use fixed width C99 data types from stdint. h - e.g. uint8_t or int32_t rather than unsigned char or int - Type char shall be restricted to declarations and operations on strings - Bit-fields shall not be defined within signed integer types - None of the bit-wise operators shall be used to manipulate signed integer data
Signed integers shall not be combined with unsigned integers in comparisons or expressions - Decimal constants meant to be unsigned should be declared with an ' U ' at the end Casts shall be done explicitly and accompanied by a comment - Use just one data declaration or one data definition per line - Allows a comment for each item.

Header Files There shall be precisely one header file for each module Each header file shall contain a preprocessor guard against multiple inclusion
`#ifndef _ADC_H`
`#define _ADC_H`
`#endif /* _ADC_H */`
Only add #includes that are immediately needed for this header file; do not add #includes for convenience of others Do not define or declare variables - i.e. uint32_t count / extern uint32_t count

Example: Module Traffic Light

traffic_light.h

```
typedef enum {
    DARK    = 0x00,
    RED     = 0x01,
    YELLOW  = 0x02,
    GREEN   = 0x03
} tl_state_type;

/** Set-up and initializes the traffic light */
void traffic_light_init(void);

/** Sets the specified state on the traffic light */
void traffic_light_set_state(tl_state_type state);

/** Returns the current state of the traffic light */
tl_state_type traffic_light_get_state(void);
```

traffic_light.h contains only those function declarations (prototypes) and type definitions that are strictly necessary for another module to know.

traffic_light.c

```
#include "traffic_light.h"

static tl_state_type traffic_light_state;
static void lamps_set(tl_state_type color);

/** See description in header file */
void traffic_light_init(void){
    traffic_light_state = DARK;
    lamps_set(DARK);
}

/** See description in header file */
void traffic_light_set_state(tl_state_type state){
    traffic_light_state = state;
    lamps_set(state);
}

/** See description in header file */
tl_state_type traffic_light_get_state (void){
    return traffic_light_state;
}

/** Turns the individual lamps on and off */
static void lamps_set(tl_state_type state){
    // drive the lamps
}
```

Variable **traffic_light_state** and function **lamps_set()** are declared **static**
→ visible only inside module traffic_light

Encapsulation:
- Interface: .h file
- Implementation: .c file

maybe finish but not rn

Microcontroller

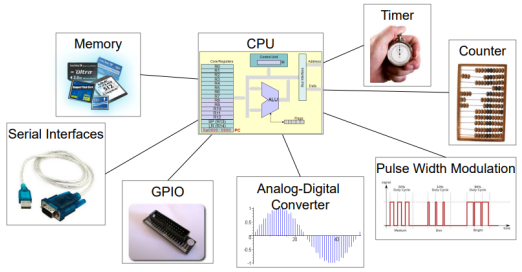
- Microcontroller
- System Bus
- Digital Logic Basics
- Synchronous Bus
- Control and Status Registers
- Address Decoding
- Slow Slaves (Peripherals)
- Bus Hierarchies
- Accessing Control Registers in C
- Conclusions

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

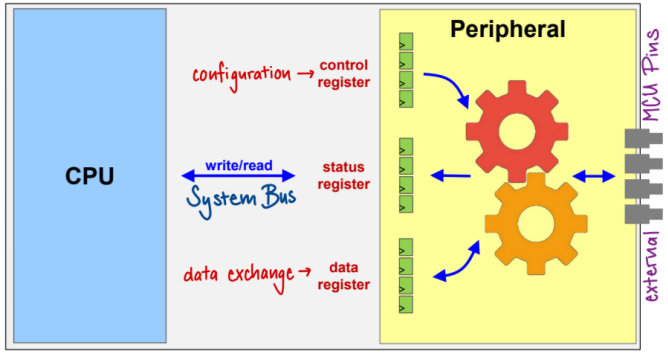


Microcontroller Architecture

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



Peripheral Registers CPU controls/monitors Peripherals through 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

Categories of Registers

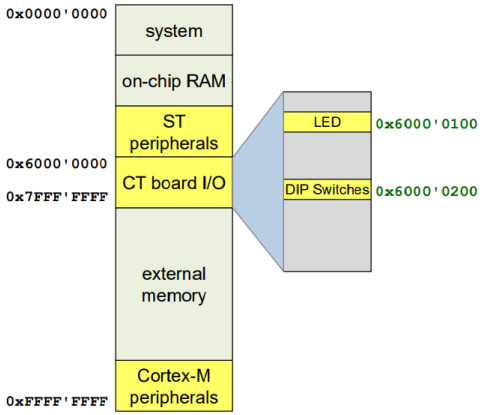
- **Control Registers**
  enable CPU to configure the peripheral
- **Status Registers**
  enable CPU to monitor the peripheral
- **Data Registers**
  enable CPU to exchange data with the peripheral

Memory-mapped Peripheral Registers

```
#define ADDR_LED_31_0        0x60000100
#define ADDR_DIP_SWITCH_31_0 0x60000200

uint32_t value = read_word(ADDR_DIP_SWITCH_31_0);
write_word(ADDR_LED_31_0, value);
```

Memory-mapped Peripheral Registers

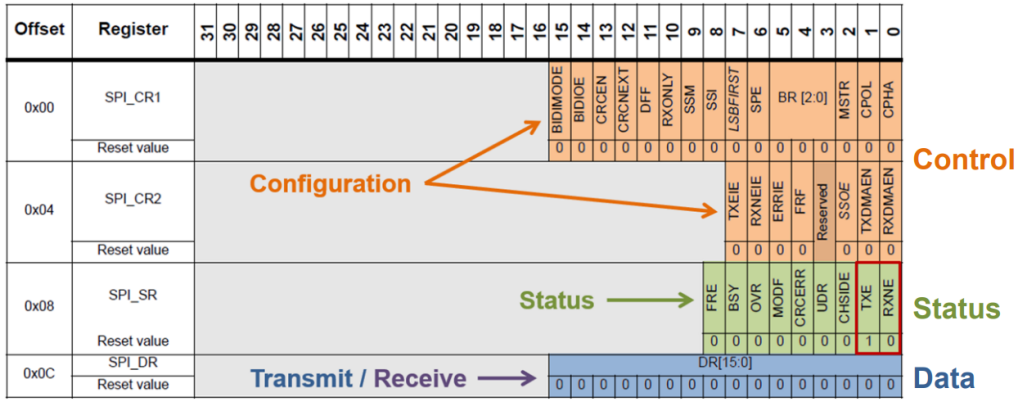
- Control register: controls states of LEDs
- Status register: monitors states of DIP switches



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)

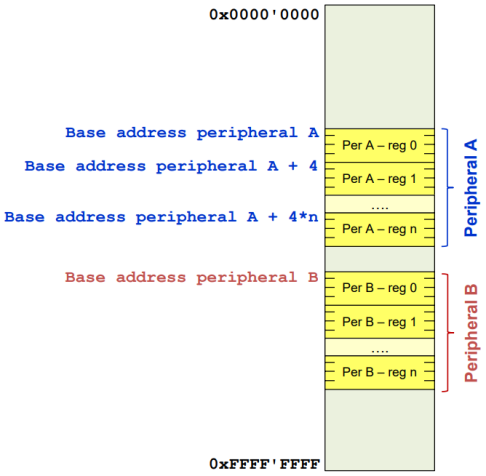


Each column shows a single flip-flop

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



CPU read/write to peripheral registers

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

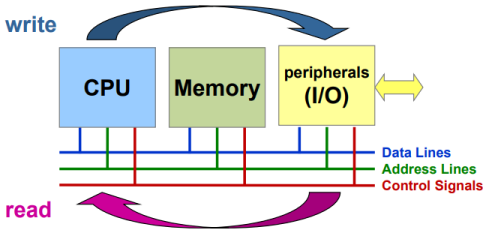
- CPU reads/writes to peripheral registers
- CPU uses memory-mapped I/O to access peripheral registers
- CPU uses load/store instructions to access peripheral registers

⇒ System Buses

System Bus

System Bus

- Interconnects CPU with memory and peripherals
- 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

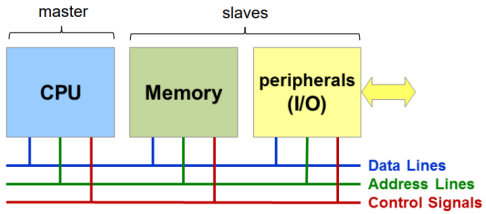


Bus Specification

- **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)

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
- **Control signals**
  - Control read/write direction
  - Provide timing information
  - Chip select, read/write, etc.



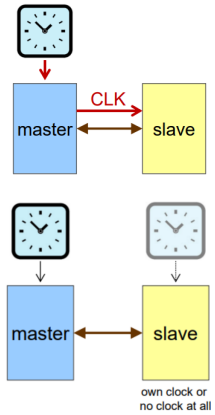
Bus Timing Options

Synchronous

- Master and slaves use a common clock
  Often a dedicated clock 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



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')

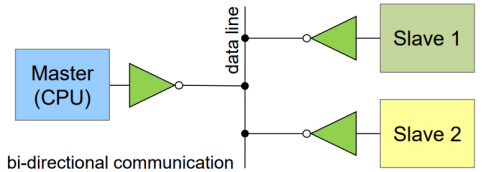


Figure only shows output paths, input paths are not shown.

Bus Contention

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

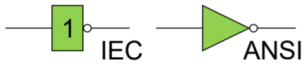
- `write` CPU drives bus $rightarrow$ all slave drivers disconnected
- `read` CPU releases bus $rightarrow$ one slave drives bus (selected through values on address lines, other slave drivers disconnected)

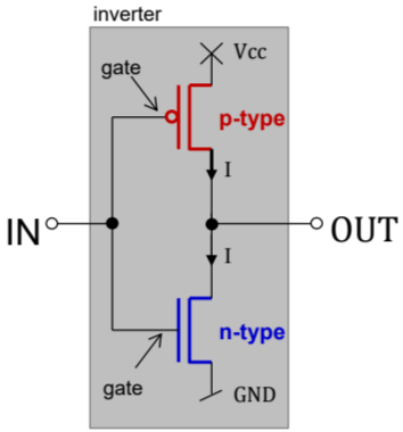
Electrically disconnecting a driver is called **tri-state** or **high-impedance** (Hi-Z) state. (switch)
CHECK IF CORRECT

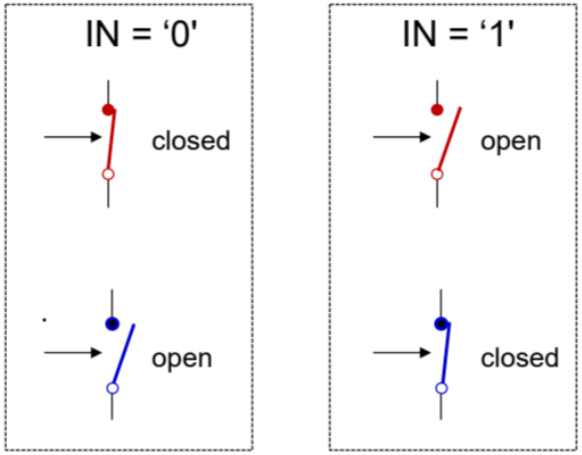
But how can a driver be disconnected electrically?

Digital Logic Basics

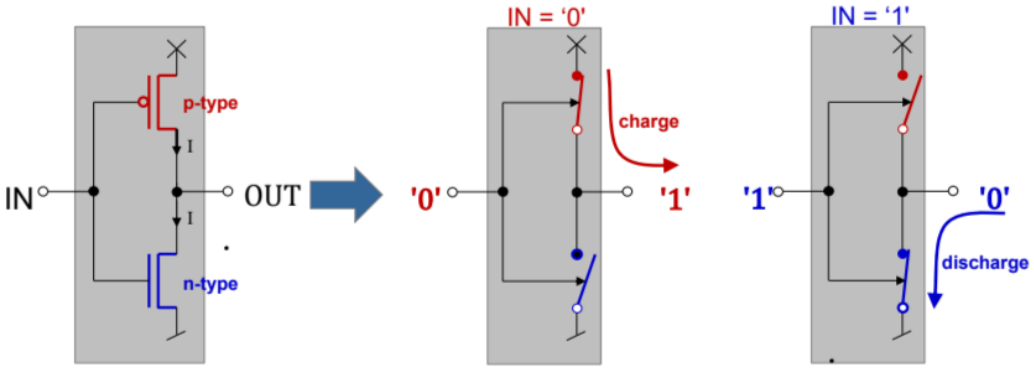
CMOS Inverter Complementary switches (transistors)
→ p-type and n-type have opposite open-close behaviour





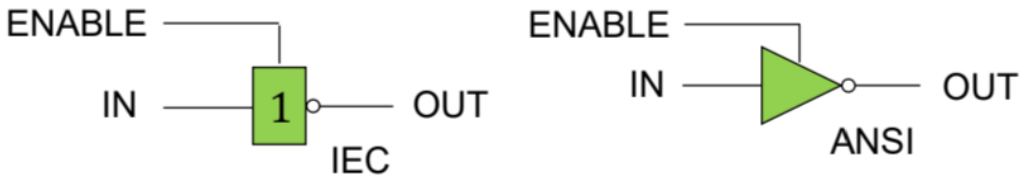


e.g. Vcc = 3V = '1', Gnd = 0V = '0'
Vcc is the supply voltage of the circuit/chip



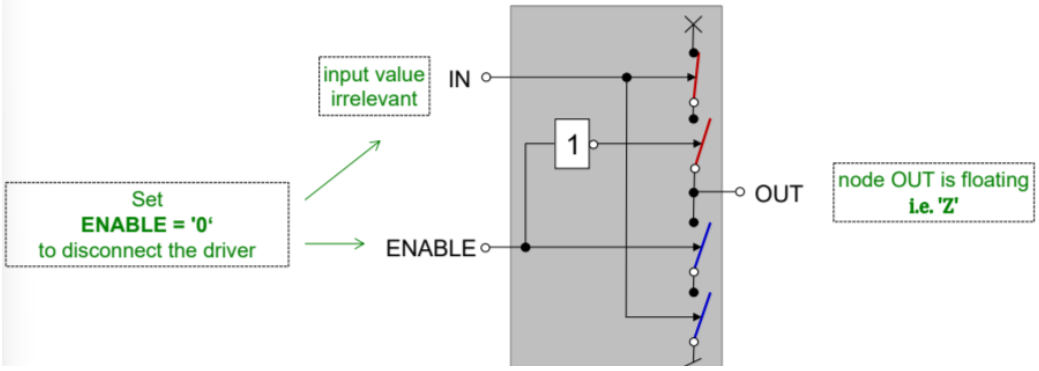
A buffer is built by connecting two inverters in series

CMOS Tri-State Inverter

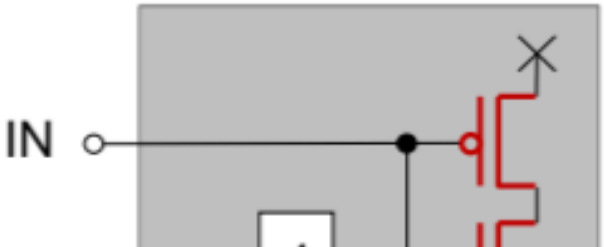


ENABLE	OUT
'1'	! IN
'0'	'Z'

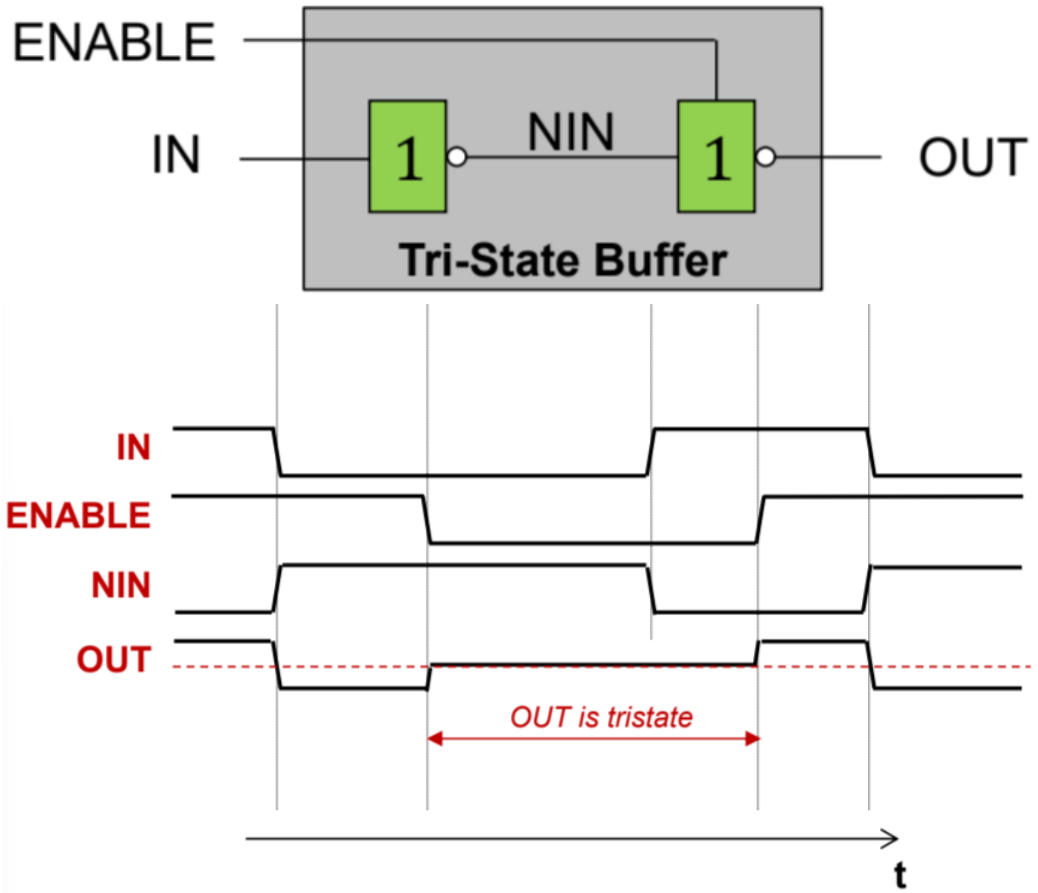
'Z' = high impedance



Implementation:

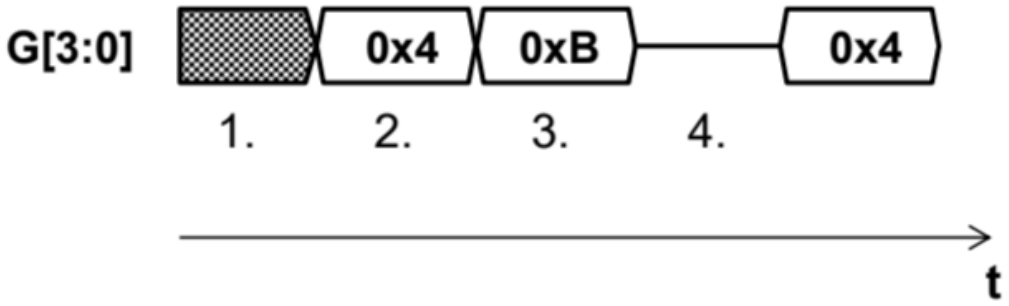


Tri-State Buffer Timing Diagram



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 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 0×4

$$\text{i.e. } G[3] = \text{ ' 0 ', G[2] = '1', G[1] = '0', G[0] = '0'}$$

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

$$\text{i.e. } G[3] = \text{ ' 1 ', G[2] = '0', G[1] = '1', G[0] = '1'}$$

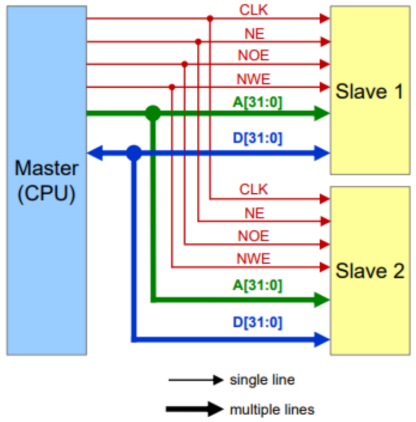
4. Tri-state
All signals $G[3:0]$ are tri-state (i.e. ' Z ' or high-impedance). "No one is driving the bus"

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 ($Nxxx$) means active-low signal - E.g. NOE means 'NOT OUTPUT ENABLE'
NOE = ' 0 ' → output enabled
NOE = ' 1 ' → output disabled

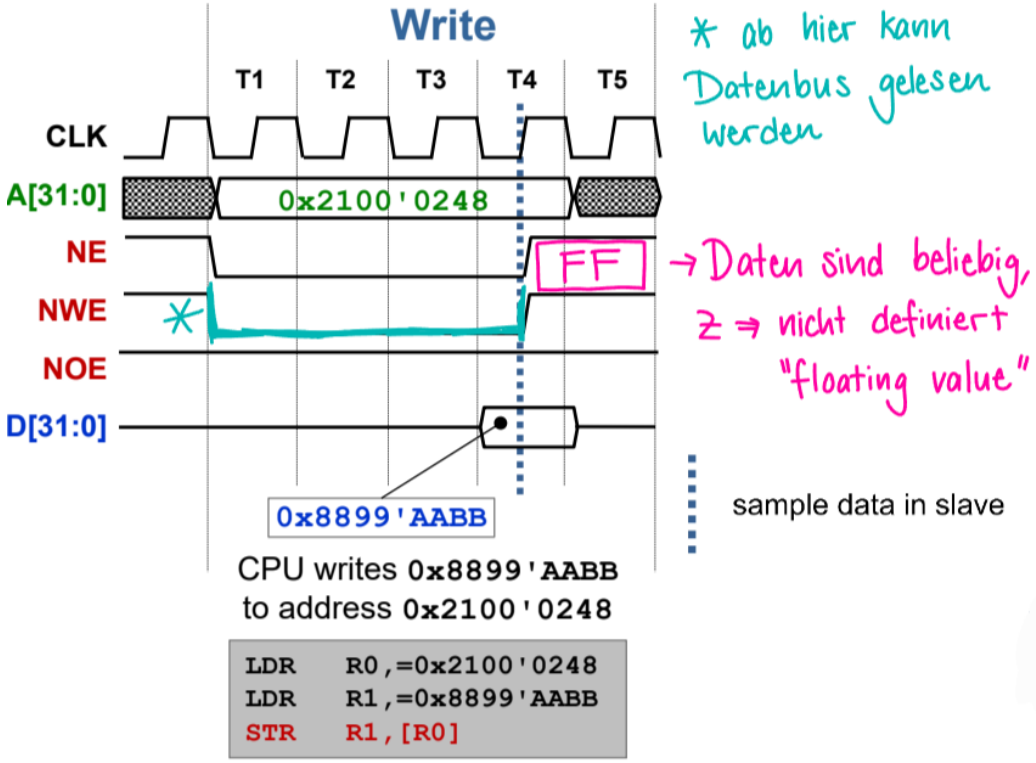
Block Diagram

- Address lines [31:0]
- Data lines [31:0]
- Control signals
  - CLK → clock
  - NE → Not enable
  - NWE → Not write enable
  - NOE → Not output enable

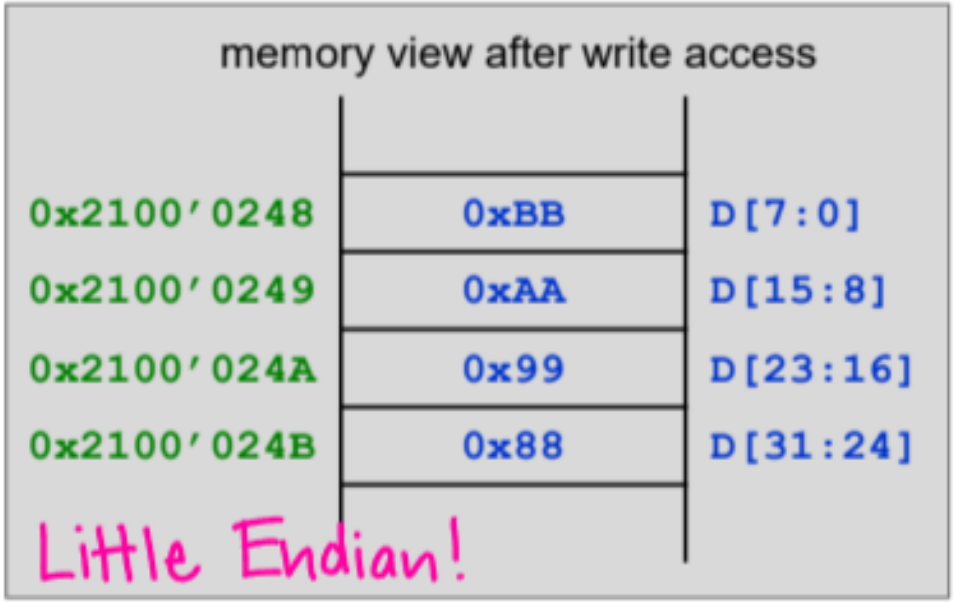


Bus Timing Diagram

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

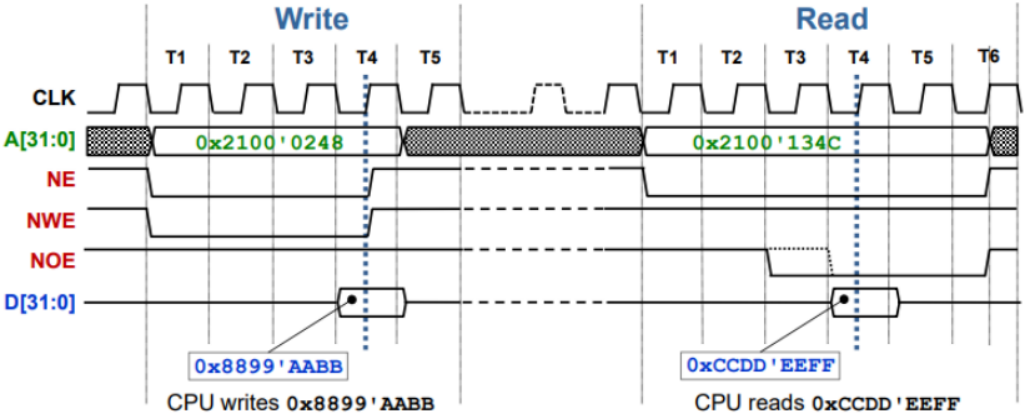



```
LDR   R0,=0x2100'0248
LDR   R1,=0x8899'AABB
STR   R1,[R0]
```



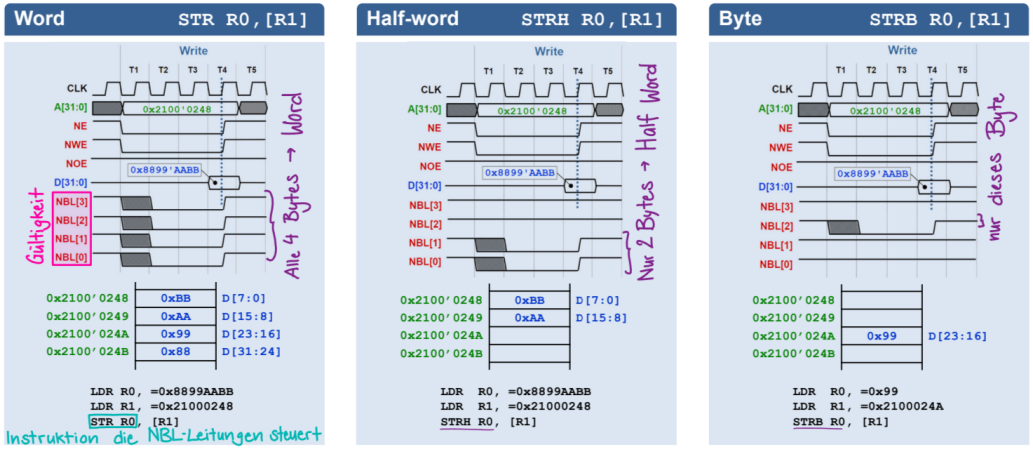
Timing Diagram

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



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

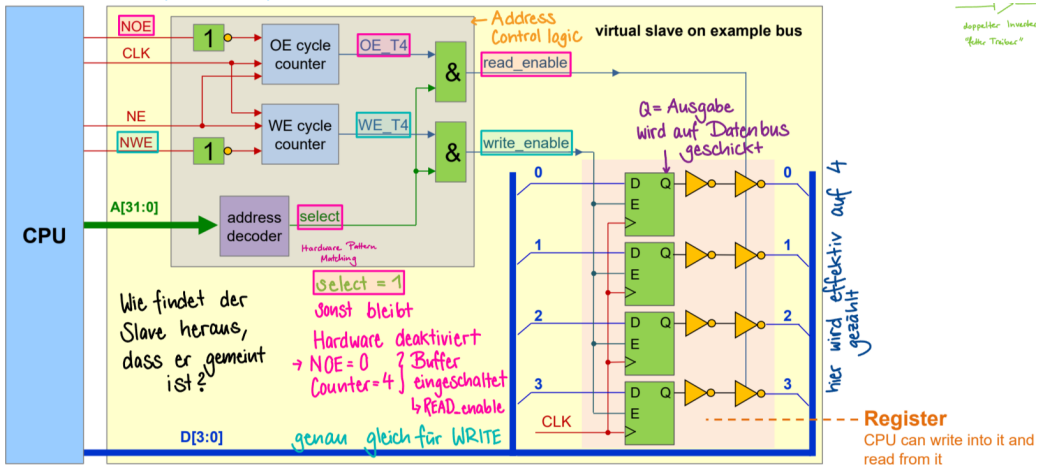


Gültigkeit: damit zeigt die CPU an, welche der 4 Bytes übertragen werden sollen (gültig = 0 (unten))

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

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
	r					rw	rw					r	r	r	r

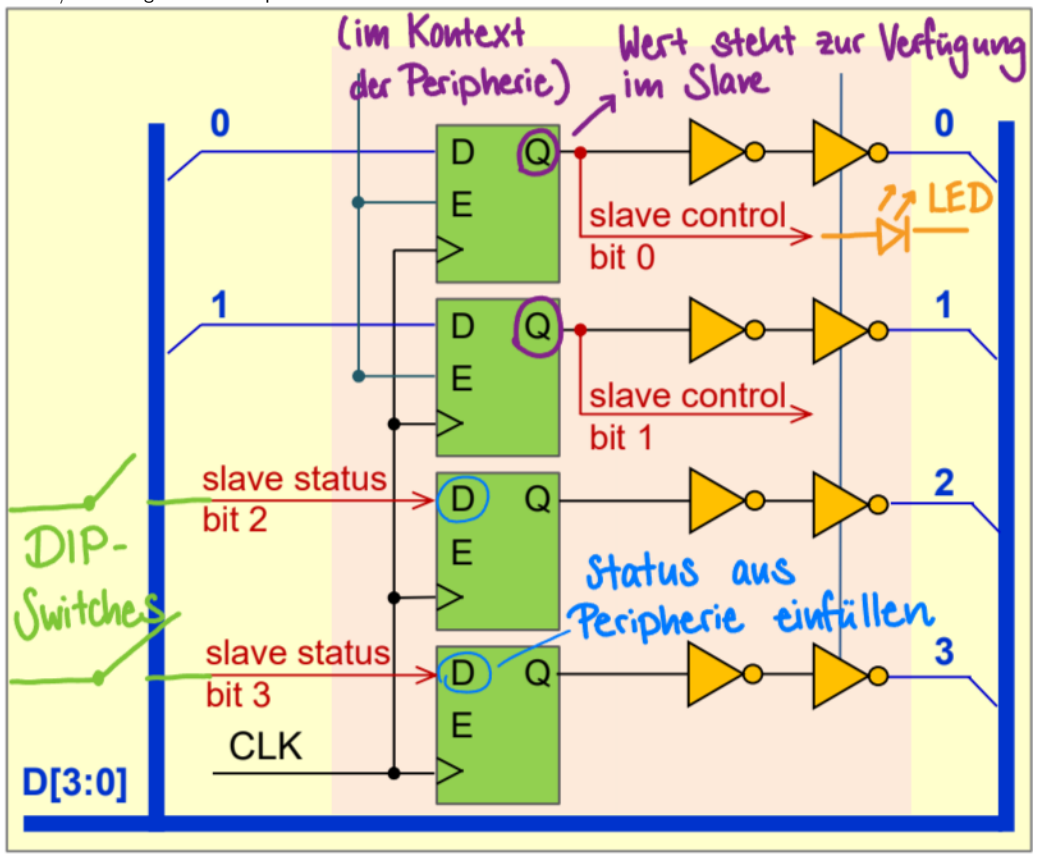
Control Bits

BRE: Backup regulator enable
EWUP: Enable WKUP pin

Status Bits

BRR: Backup regulator ready
PVDO: PVD output
SBF: Standby flag
WUF: Wakeup flag

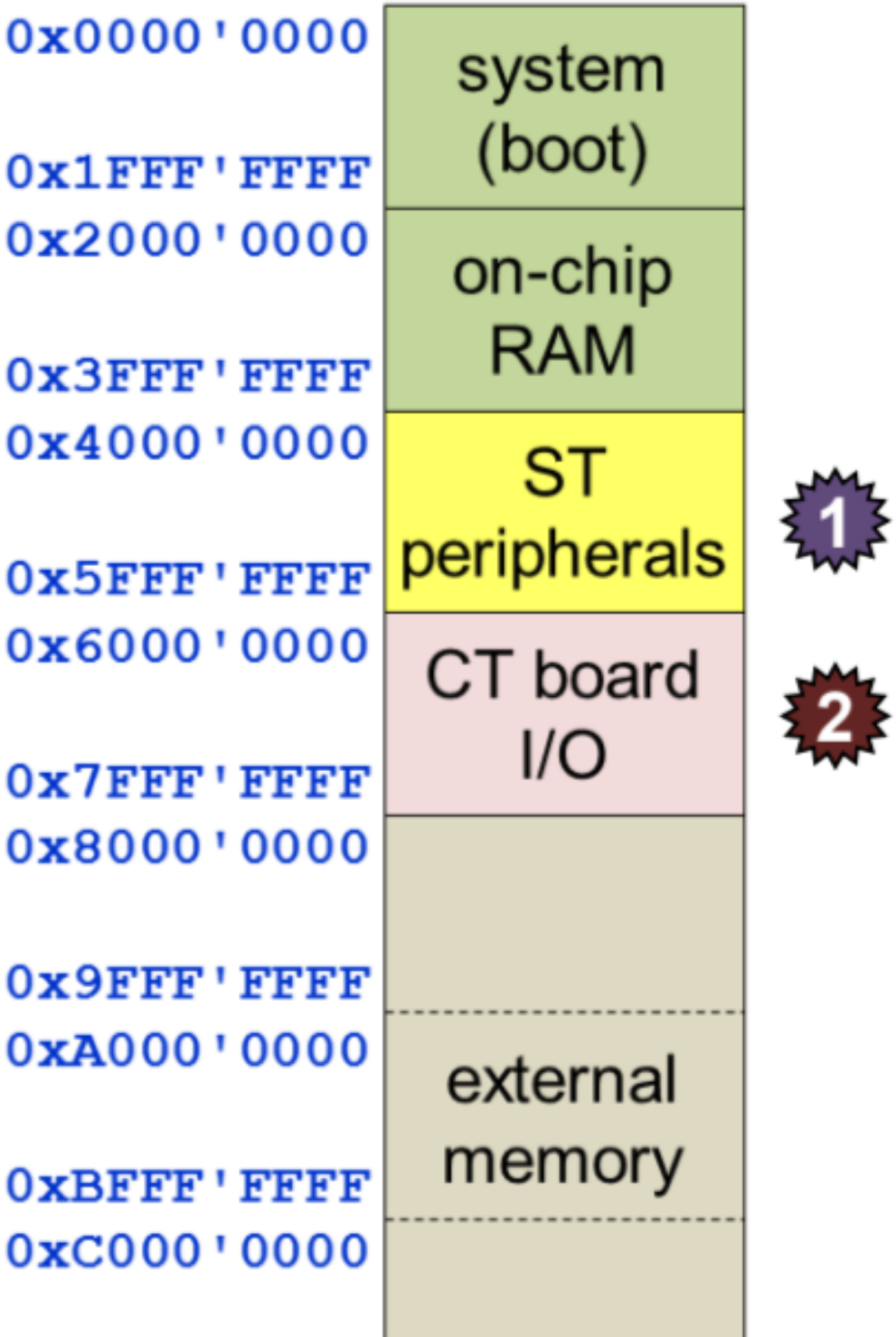
Control/Status register on example bus:



control bits: 0 and 1
status bits: 2 and 3
The same register may contain both control and status bits!

Control and Status Registers on CT Board

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



Address Decoding

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

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

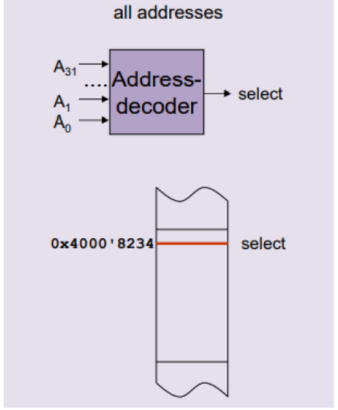
Full Address Decoding

- 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

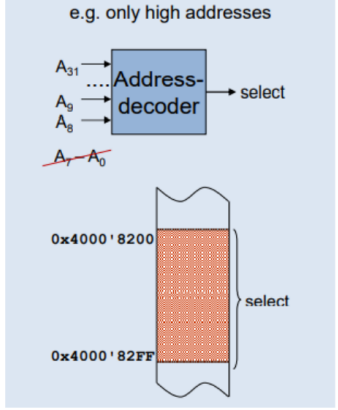
Partial Address Decoding

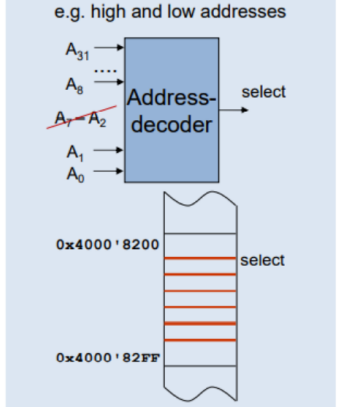
- Only a subset of address lines are decoded
- A control register can be accessed at multiple locations
- 1:n mapping: Multiple addresses map to the same hardware register
- Motivation: Simpler and possible Aliasing (Map a hardware register to several addresses)

Full Address Decoding



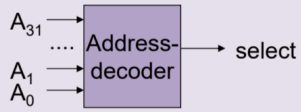
Partial Address Decoding





Full Address Decoding

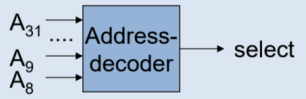
- all addresses from A31 to A0



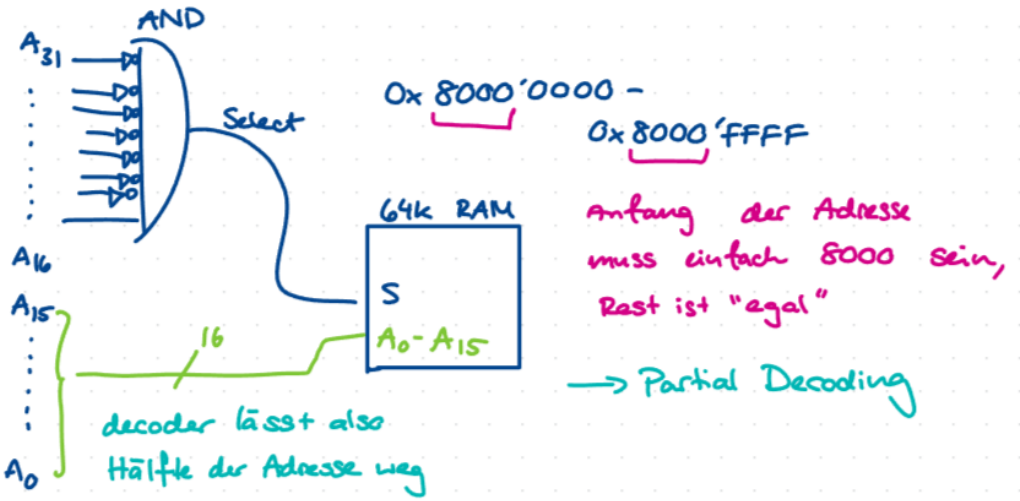
- select is active for exactly one address
- E.g. at `0x4000'8234`

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`



Slow Slaves (Peripherals)

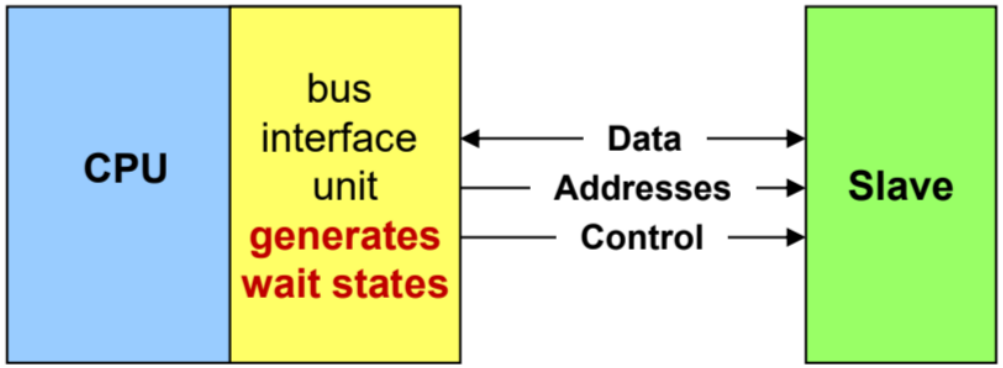
Slow Slaves

Problem: Individual Slave Access times

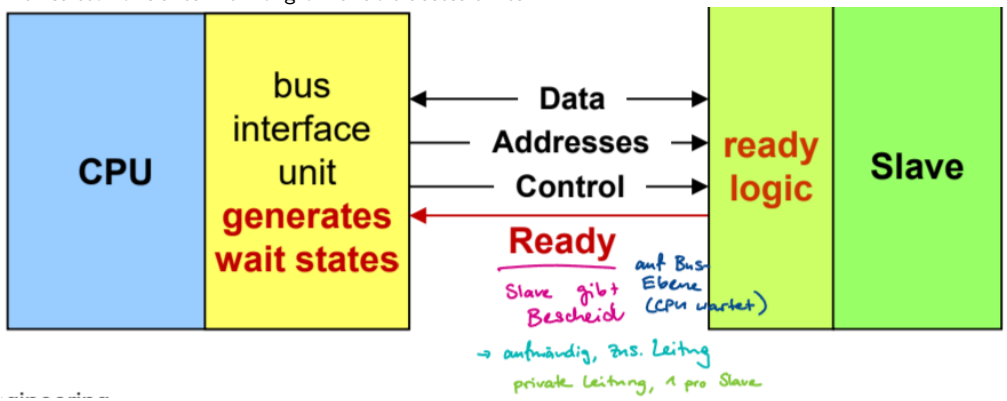
- If slowest slave defines bus cycle time → reduced bus performance
- How can we get an individual bus cycle time for each slave?

Solutions for slow slaves two possibilities:

- **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)

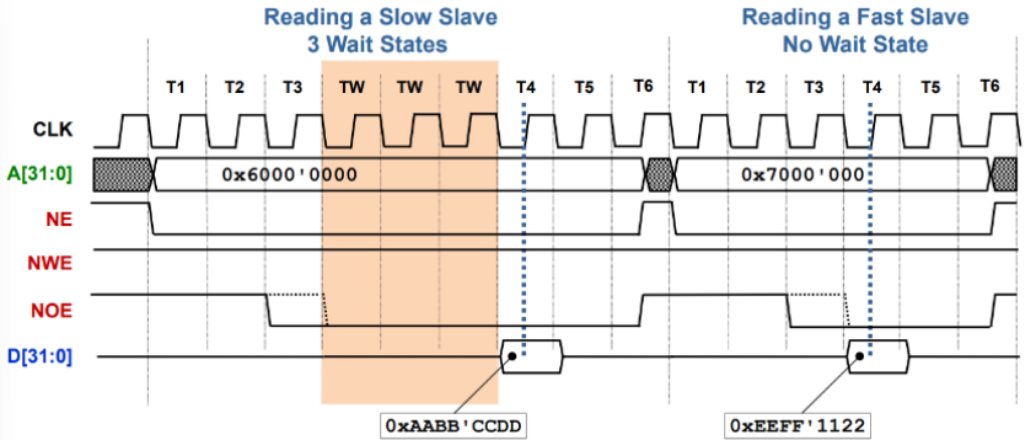


- **Bus Mastering** Slave tells bus interface unit when it is ready
  Allow a peripheral to take control of the bus and perform its own accesses (e.g. DMA)
  Well suited for slaves with long or variable access times



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)

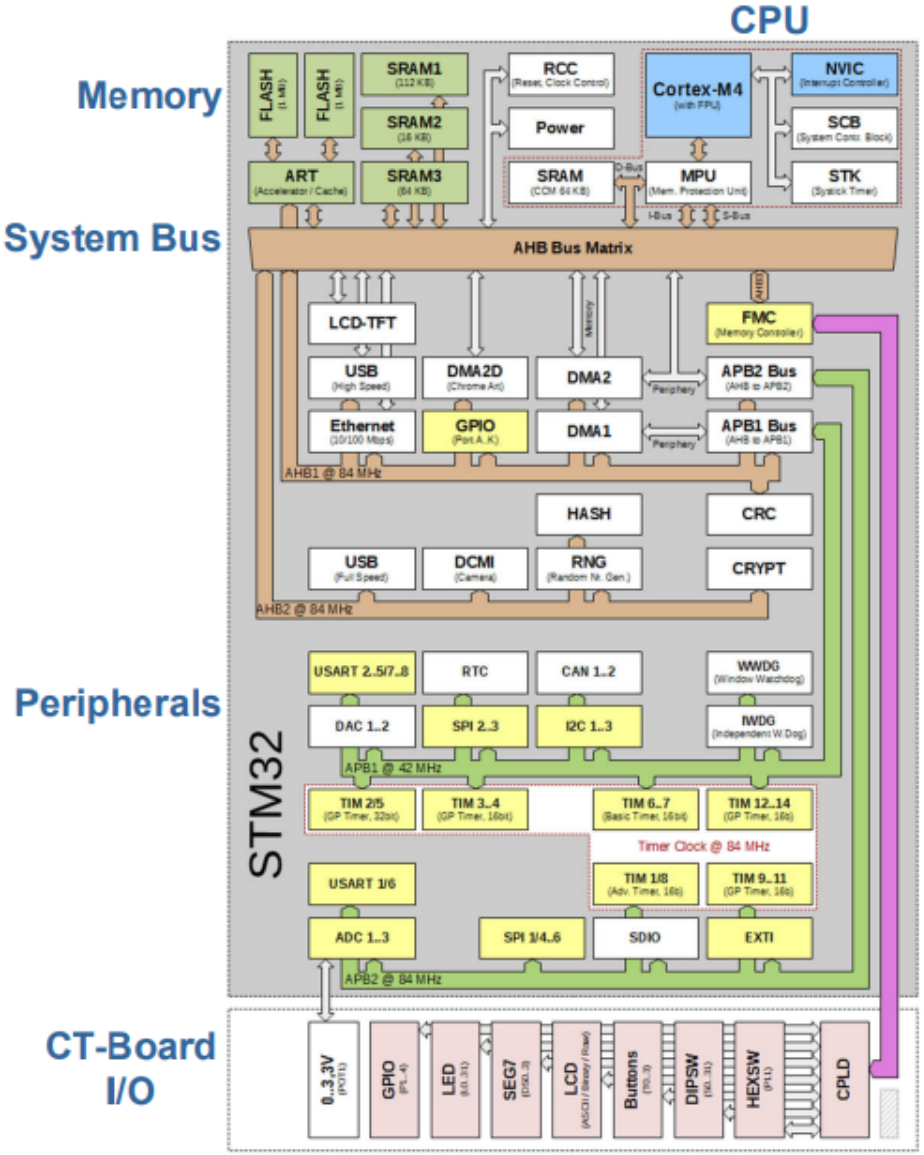


Bus Hierarchies

STM32 Microcontroller

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

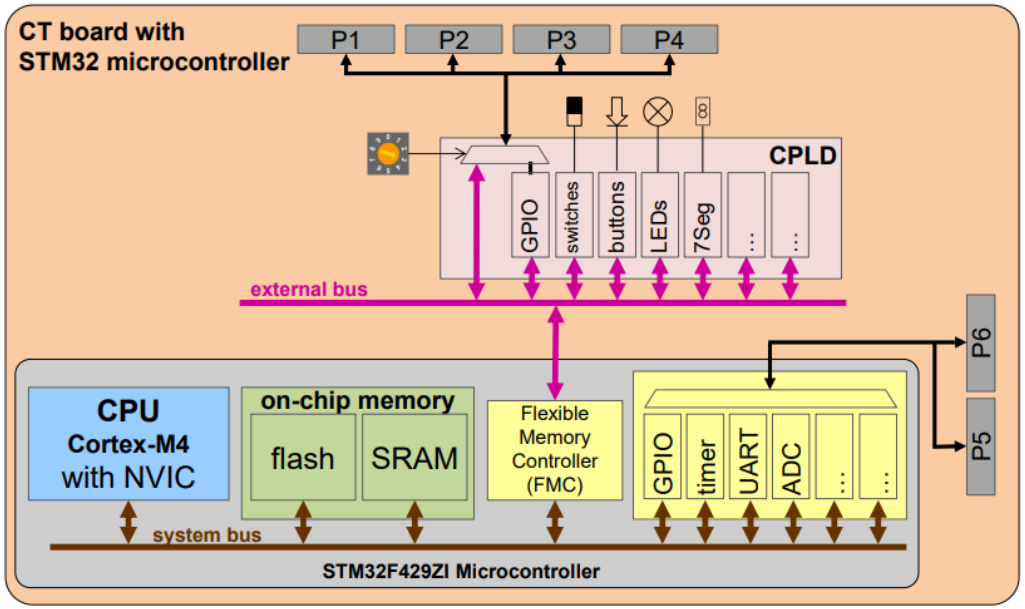
- On-chip system bus: 32 data lines, 32 address lines and control signals
- Off-chip external bus: 16 data lines, 24 address lines and control signals



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.

CT Board with STM32 Microcontroller and Buses



Real-world Systems are partitioned into multiple buses.

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 = 0xBBBBBBB;
    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 = 0xBBBBBBB;
    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

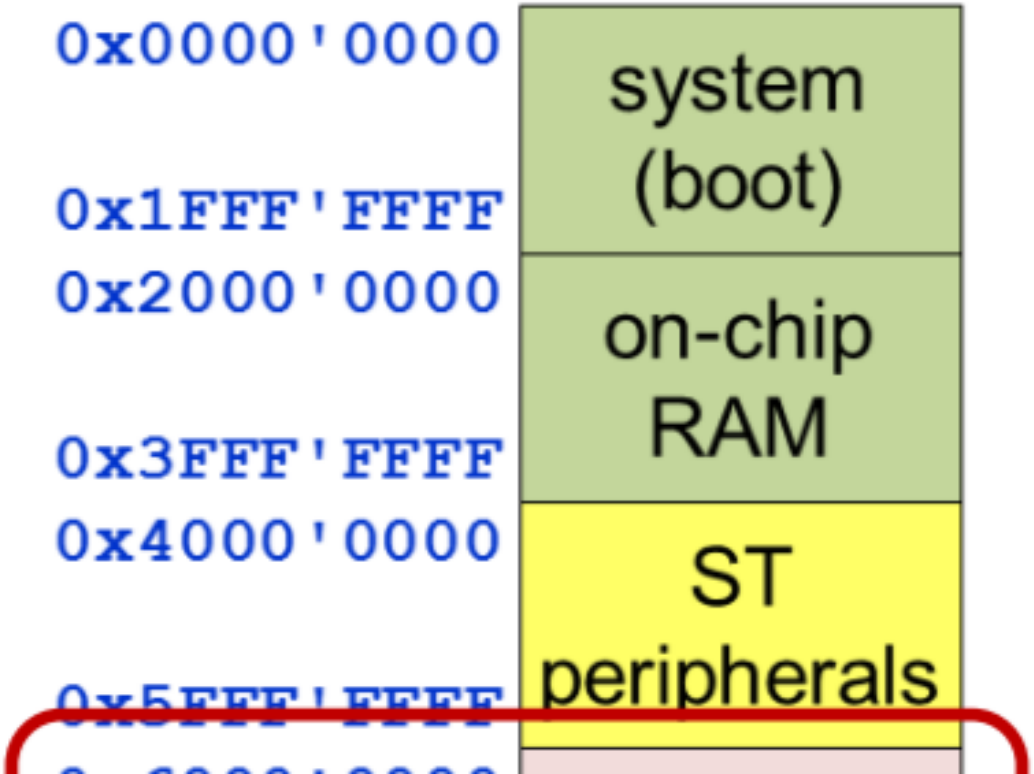
Access through Pointers e.g. writing to and reading from CT Board I/O

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


// set LEDs
p_reg = (volatile uint32_t *) (0x60000100);
*p_reg = 0xAA55AA55;


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

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

write 0xAA55'AA55 to LEDs

read dip-switches



Using Preprocessor Macros → #define

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

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

//Write LED register to 0xBBCC'DDEE
LED31_0_REG = 0xBBCCDDEE;
//Read Button register to aux_var
aux_var = BUTTON_REG;
```

`(*((volatile uint32_t *) 0x60000100))`
→ dereference the pointer to the register address
→ cast the address to a pointer to a 32-bit register

Conclusions

Conclusions Microcontrollers → Embedded Systems - Low cost, real time, low power, extreme environments
System Bus - Address, data and control lines - Synchronous or asynchronous - CPU (master) reads from or writes to slave - Timing sequences - Wait states
Address Decoding - Who is the CPU talking to? - Full vs. partial address decoding
Accessing Control Registers in C - Qualifier volatile - Use of pointers for memory accesses

Learning Objectives:
At the end of this lesson, you will be able - to enumerate the signal groups of a system bus - to distinguish between 'synchronous' and 'asynchronous' bus timing - to know what the term 'tri-state' means - to interpret simple bus timing diagrams - to describe the concept of a bus and how data is transferred on a bus - to describe the function and purpose of control and status registers - to explain the terms 'full address decoding' and 'partial address decoding' - to access control registers from C - to explain the meaning of the qualifier 'volatile' in C - to analyze address decoding logic and to derive the applicable addresses or address ranges - to derive an address decoding logic for a given address (range) - to explain the function of wait states

GPIO

General Purpose Input Output

- Working with documents
- General Purpose Input Output (GPIO)
- GPIO Structure
- Configuration Registers
  Direction, Output Type, Output Speed, Pull-Up/Pull-Down
- Data Registers
  Reading Input Data, Writing Output Data
- GPIO Cookbook
- Hardware Abstraction Layer (HAL)

GPIO

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

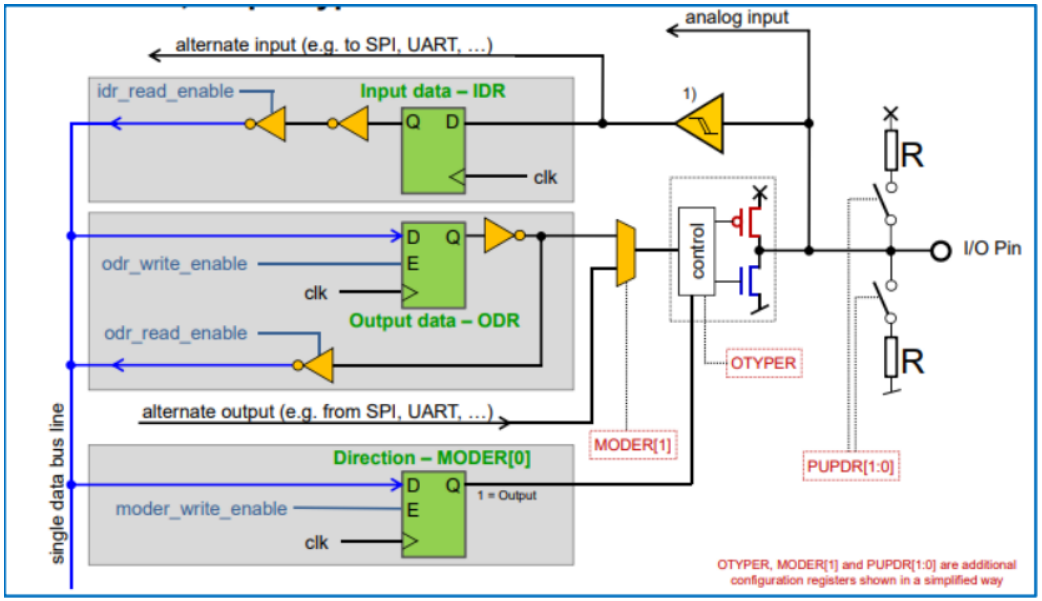
Problem:
- Limited number of pins
- Many functions to be implemented (many functional blocks included)

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

Register Access

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

Structure



Configuration Registers
- **GPIOx_MODER** (Mode Register)
- **GPIOx_OTYPER** (Output Type Register)
- **GPIOx_OSPEEDR** (Output Speed Register)
- **GPIOx_PUPDR** (Pull-up/Pull-down Register)

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)

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)

Configuration
- **Mode:** Input, Output, Alternate Function, Analog
- **Type:** Push-Pull, Open Drain
- **Speed:** Low, Medium, High, Very High
- **Pull-Up/Pull-Down:** No, Pull-Up, Pull-Down, Reserved

Hardware Abstraction Layer (HAL)

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

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

Accessing a register:
- each GPIO port has the same 10 registers
- there are 11 GPIO ports → GPIOA to GPIOI

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

size of registers

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

Conclusion

Conclusion

Learning Objectives: At the end of this lesson, you will be able - to work with register descriptions in reference manuals - to explain the concept and implementation of GPIOs - to explain the differences between open-drain and push-pull - to use GPIOs in your own programs - to explain the idea of a HAL

Serial Data Transfer

Overview

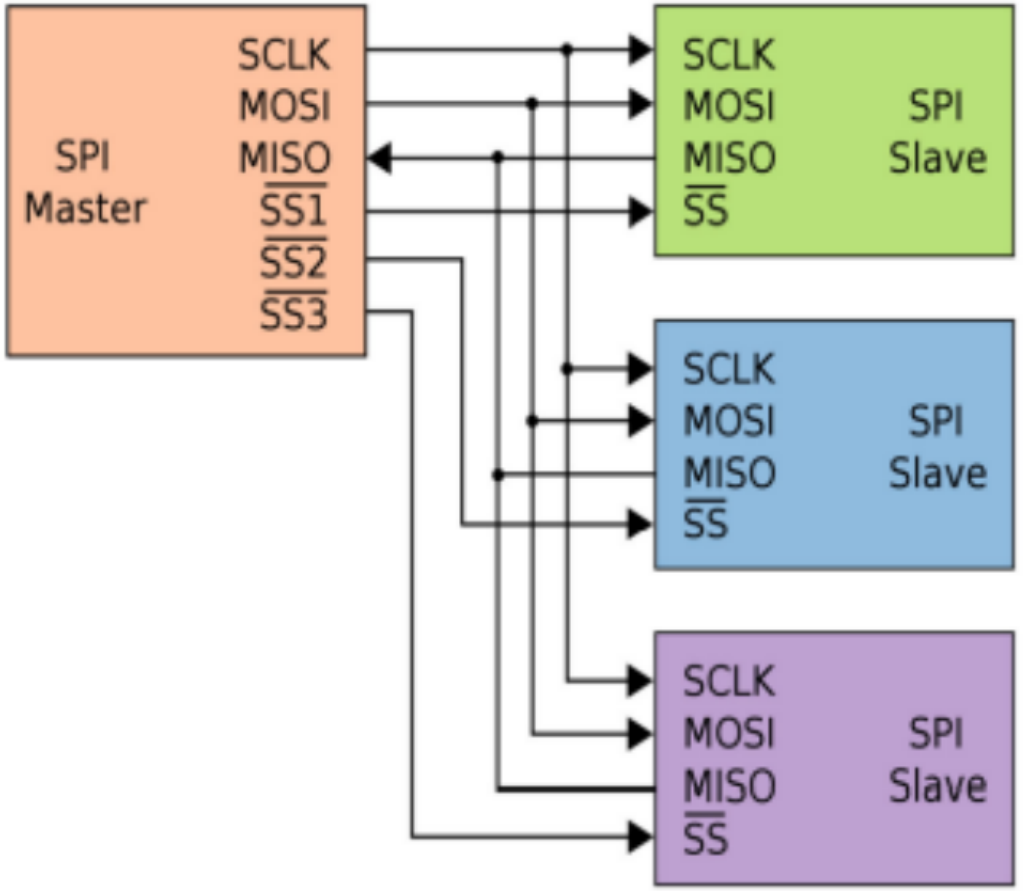
UART	SPI	I2C
serial ports (RS-232)	4-wire bus	2-wire bus
TX, RX opt. control signals	*MOSI, MISO, SCLK, SS*	*SCL, SDA*
point-to-point	point-to-multipoint	(multi-) point-to-multi-point
full-duplex	full-duplex	half-duplex
asynchronous	synchronous	synchronous
only higher layer addressing	slave selection through $\overline{SS}$ signal	7/10-bit slave address
parity bit possible	no error detection	no error detection
chip-to-chip, PC terminal program	chip-to-chip, on-board connections	chip-to-chip, board-to-board connections
The three interfaces provide the lowest layer of communication and require higher level protocols to provi interpret the transferred data.		

SPI - Serial Peripheral Interface
- Master/slave
- Synchronous full-duplex transmissions (MOSI, MISO)
- Selection of device through Slave Select (SS)
- no acknowledge, no error detection
- Clock signal (SCK) for synchronization: four mode → CPOL (clock polarity), CPHA (clock phase)

UART - Universal Asynchronous Receiver Transmitter (Serial Interface)
- Transmitter and receiver use diverging clocks
- synchronization using start and stop bits → overhead
- longer connections require line drivers → RS-232/RS-485

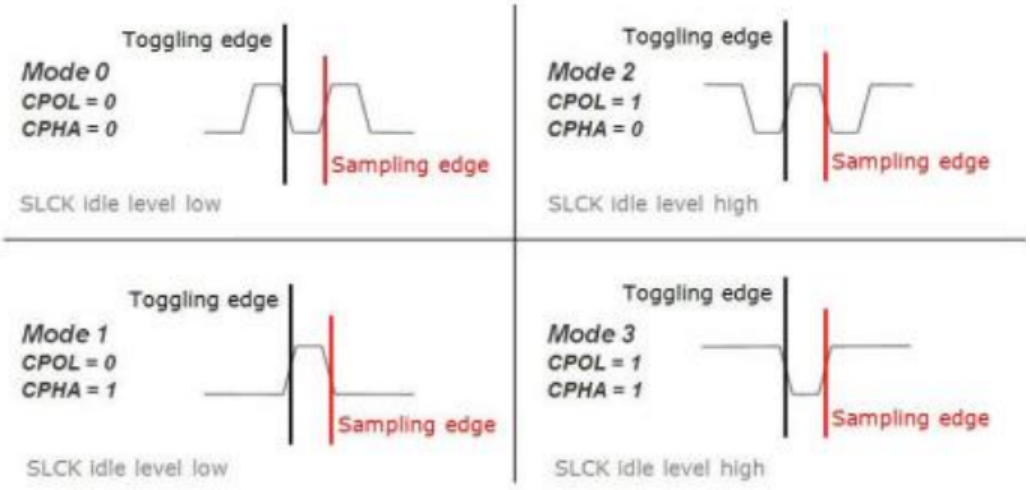
I2C - Inter-Integrated Circuit
- Synchronous half-duplex transmission (SCL, SDA)
- 7-bit slave addresses
- Acknowledge, error detection
- Clock signal (SCL) for synchronization

Single Master - Multiple Slaves
- Master generates a common clock signal for all Slaves
- MOSI: From Master Output to all Slave Inputs
- MISO: From Master Input to all Slave Outputs → all slave outpuzs connected to single master input
- Slaves: Selectable by Slave Select (SS) signal
  - Individual Select $\overline{SS1}$, $\overline{SS2}$, $\overline{SS3}$
  - $\overline{SSx} = 1$ → slave output MISOx is tri-state



Clock Polarity (CPOL) Clock Phase (CPHA)

- **CPOL:** Clock Polarity
  - 0: Clock is low in idle state
  - 1: Clock is high in idle state
- **CPHA:** Clock Phase
  - 0: Data is sampled on the first edge of the clock
  - 1: Data is sampled on the second edge of the clock



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

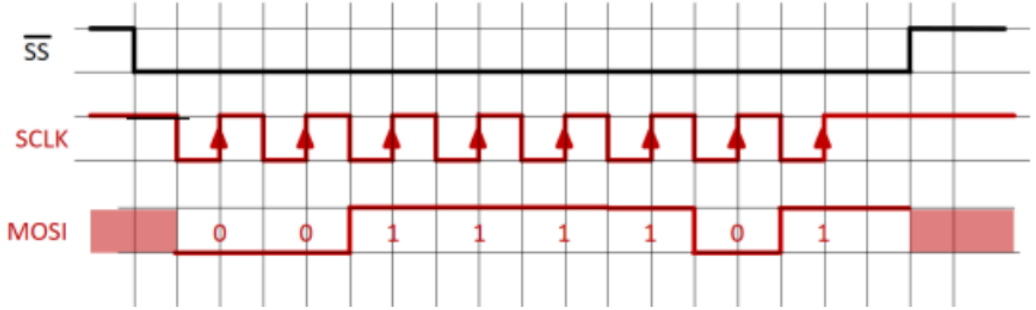
Serial Connection - SPI

Properties of SPI

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

SPI - Serial Peripheral Interface

Ein Prozessor (SPI Master) sendet das Byte 0x3D = 0011 1101
Die Schnittstelle ist wie folgt konfiguriert:
Mode = 3, CPOL = 1, CPHA = 1, MSB first

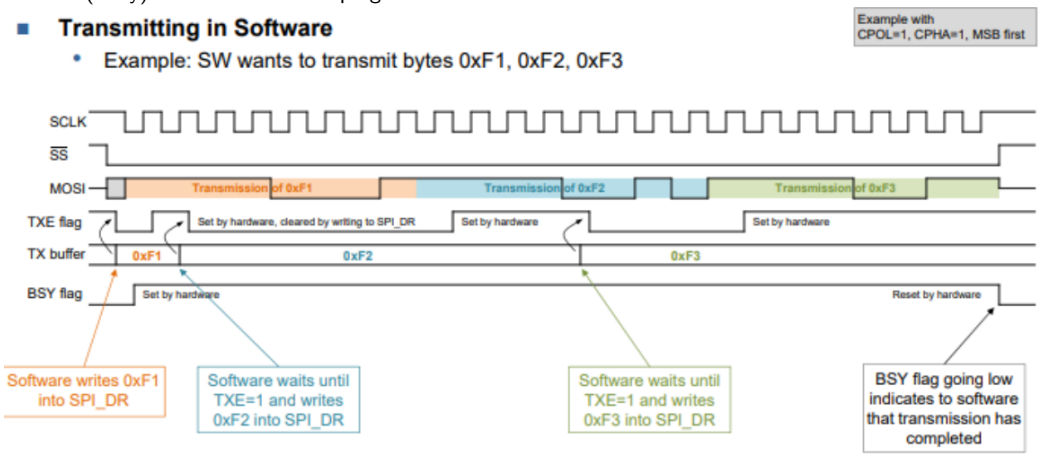


Synchronizing Hardware and Software

- 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

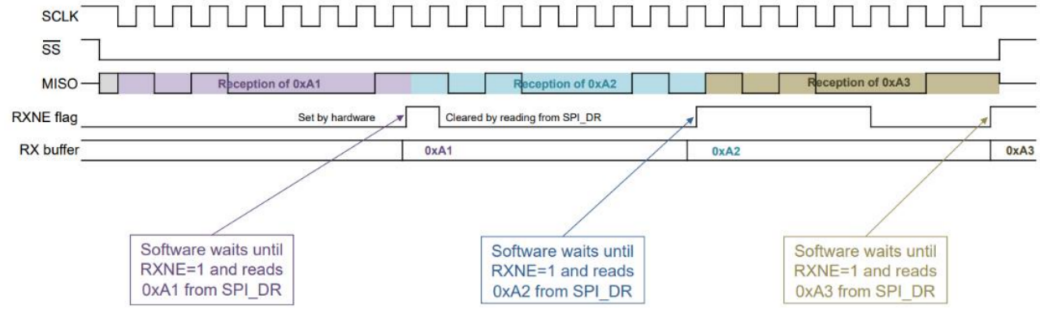
Transmitting in Software

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



Receiving in Software

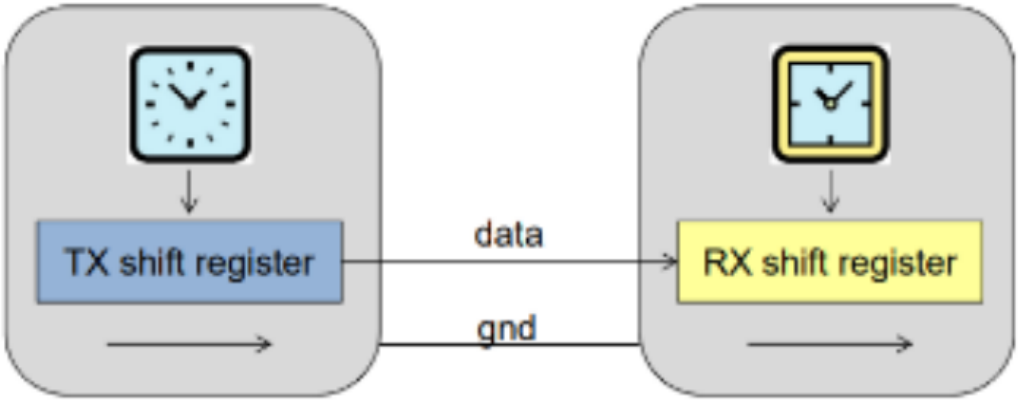
- Example: SW receives bytes 0xA1, 0xA2, 0xA3



UART - Universal Asynchronous Receiver Transmitter

Connecting shift registers with diverging clock sources

- same target frequency
- different tolerances and divider ratios
- requires synchronization at start of each data item receiver



UART Characteristics

synchronization

- Each data item (5-8 bits) requires synchronization

Asynchronous data transfer

- mismatch of clock frequencies in TX and RX
- requires overhead for synchronization → additional Bits
- requires effort for synchronization → additional Hardware

Advantages

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

on-board connections

- signal levels are 3V or 5V with reference to ground
- off-board connections require strong output drivers