

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

Braces

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

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

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

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

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);`

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)

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
};
```

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.

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

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, add works 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 succeeded"boolean. - "PCI device present is 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.

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

Coding Techniques

Example: Module Traffic Light

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

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

```
/** 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);
```

```
#include "traffic_light.h"                      traffic_light.c
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

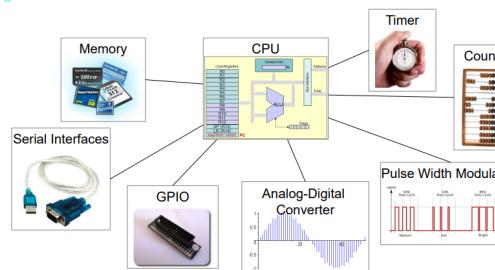
- Microcontroller
- System Bus
- Digital Logic Basics
- Synchronous Bus
- Control and Status Registers
- Address Decoding
- Slow Slaves (Peripherals)
- Bus Hierarchies
- Accessing Control Registers in C
- Conclusions

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

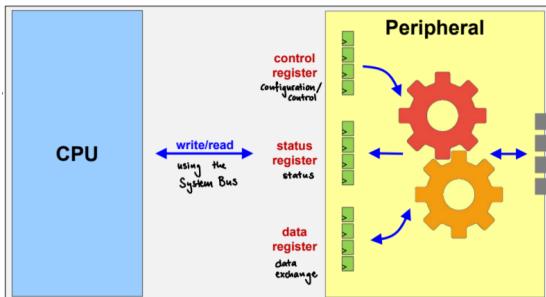
⇒ CPU with integrated memory and peripherals



Microcontroller Architecture

Peripherals and Registers

Peripherals



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

Peripheral Registers

The CPU controls and 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

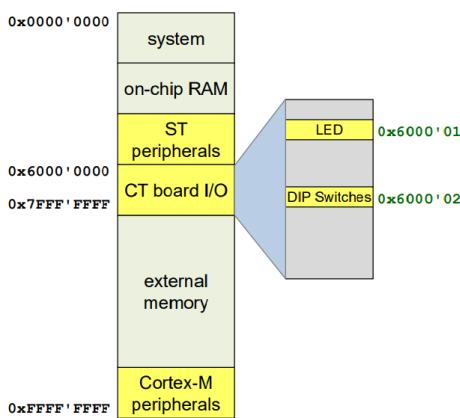
Categories of Registers

A Peripheral typically has multiple registers that can be categorized as:

- **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

- 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)

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	CRGEN	CRCMEXT	DF1	RXONLY	SSM	SSI	LSBFIRST	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																																
	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																																
	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																																
	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

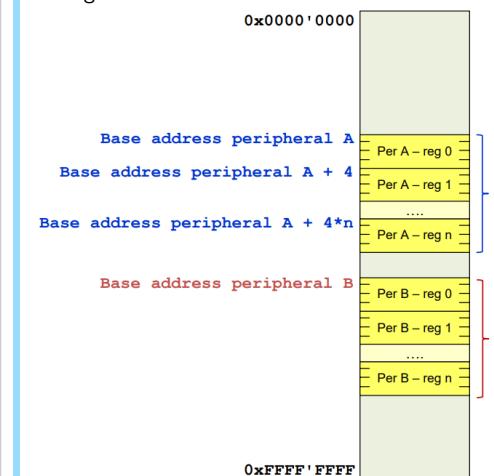
Transmit / Receive

source: STM32F42xxx Reference Manual

Each column shows a single flip-flop

Peripheral Register Addressing

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



Reading/Writing to peripheral registers in C

```
dereference pointer →
#define LED31_0_REG (*((volatile uint32_t *) (0x60000100)))
#define BUTTON_REG (*((volatile uint32_t *) (0x60000210)))

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

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

CPU read/write to peripheral registers

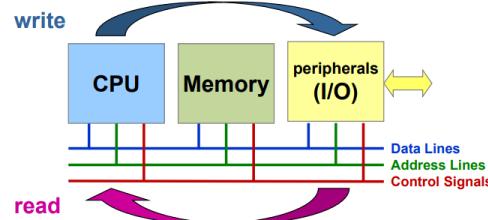
How does the CPU write to and read from peripheral registers?

- 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

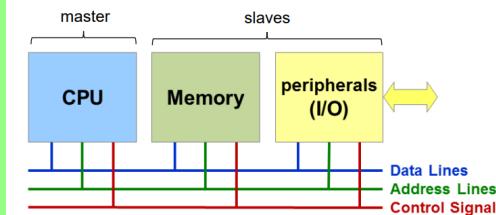


Bus Specification

- 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)

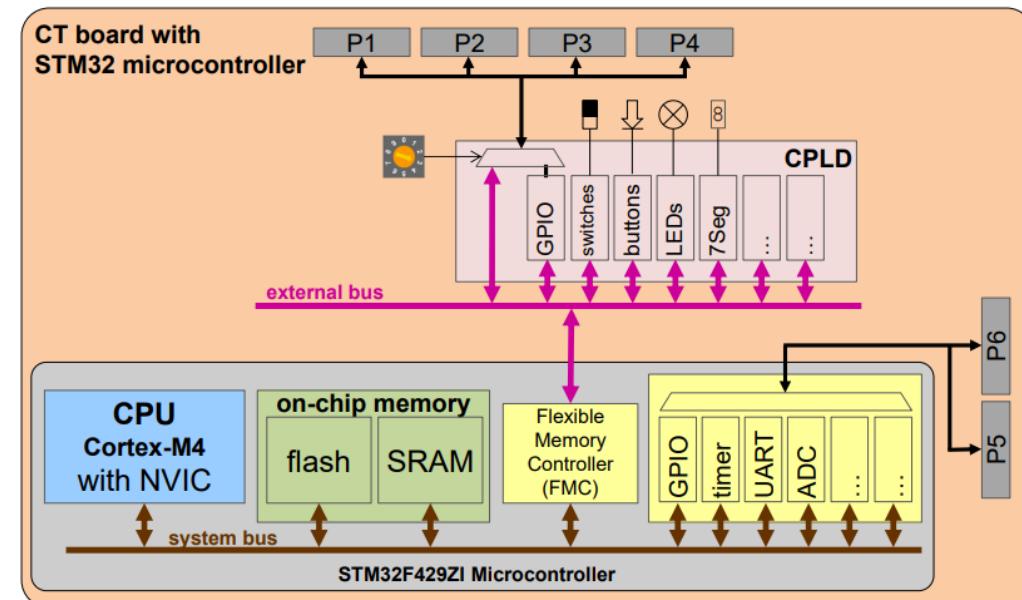
Signal Groups

- **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.



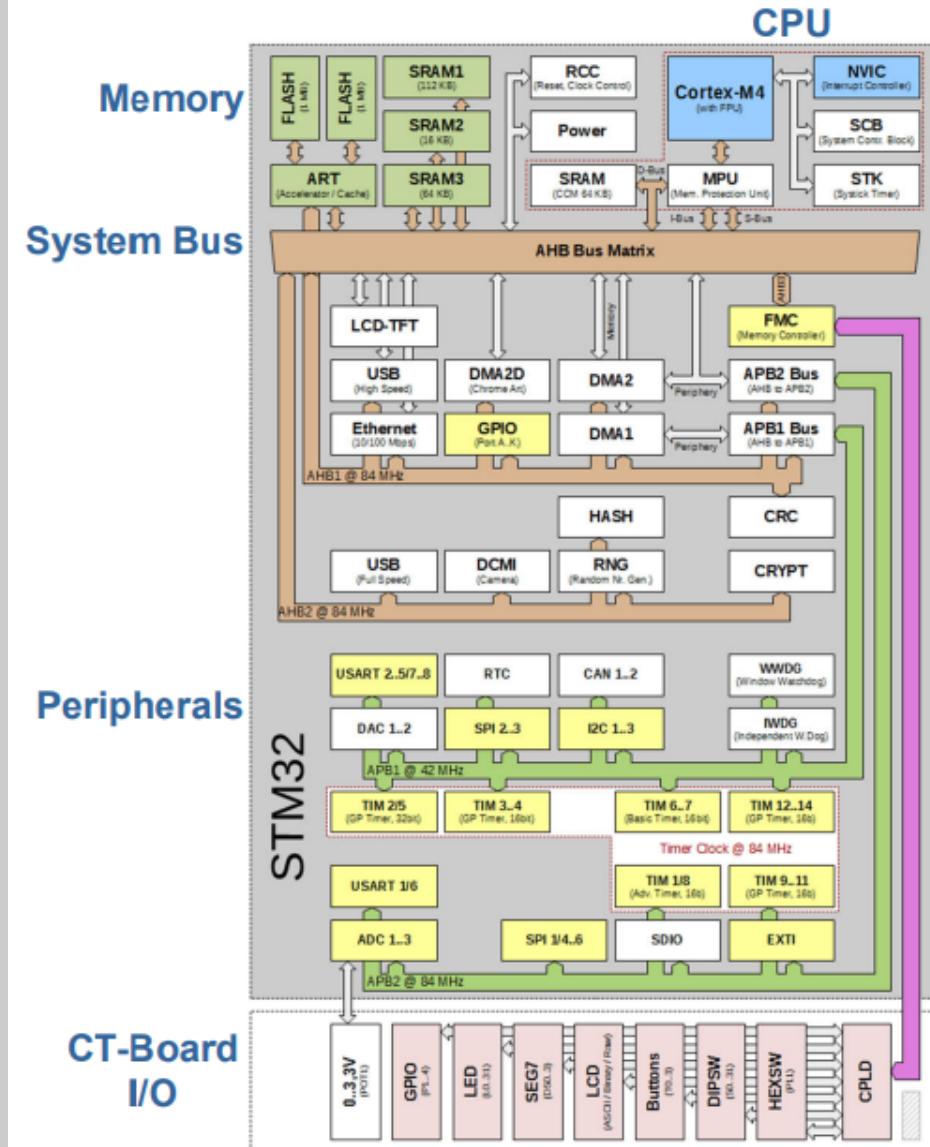
Our System

CT Board with STM32 Microcontroller and Buses



STM32 Microcontroller

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



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.

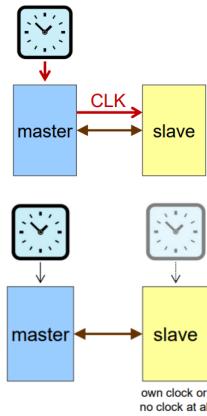
Timing and Access

Synchronous and Asynchronous Buses

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')

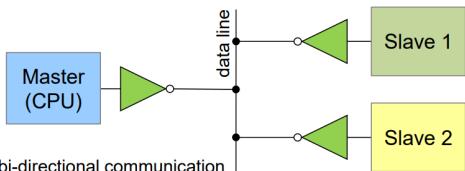


Figure only shows output paths, input paths are not shown.

Bus Contention

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

- write CPU drives bus → 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)

CHECK IF CORRECT

Control and Status Registers

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

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

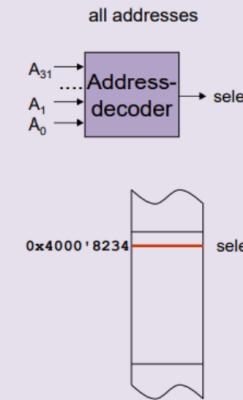
Full Address Decoding

- 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

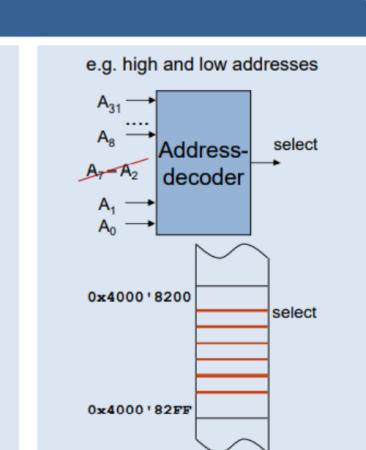
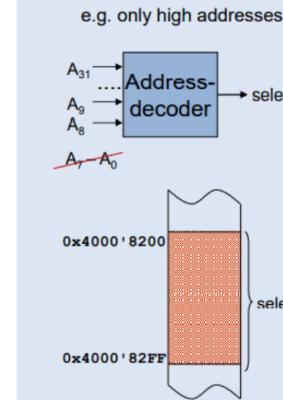
Partial Address Decoding

- 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
- Map a hardware register to several addresses

Full Address Decoding



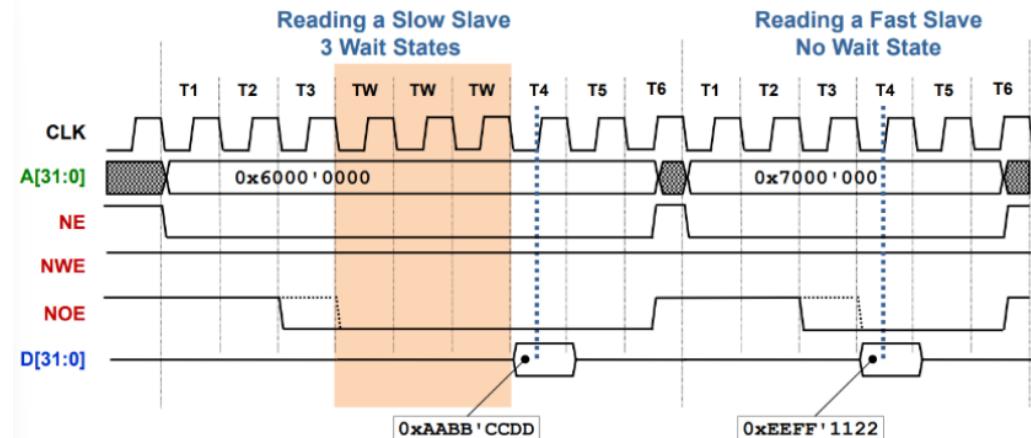
Partial Address Decoding



Slow Slaves (Peripherals)

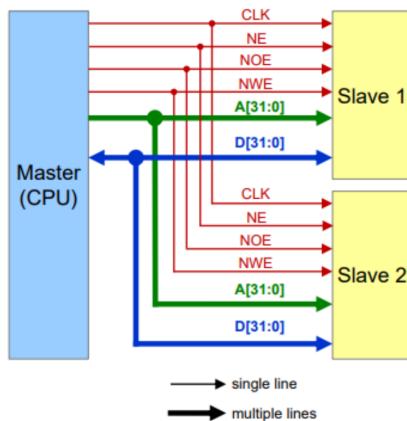
Slow Slaves

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



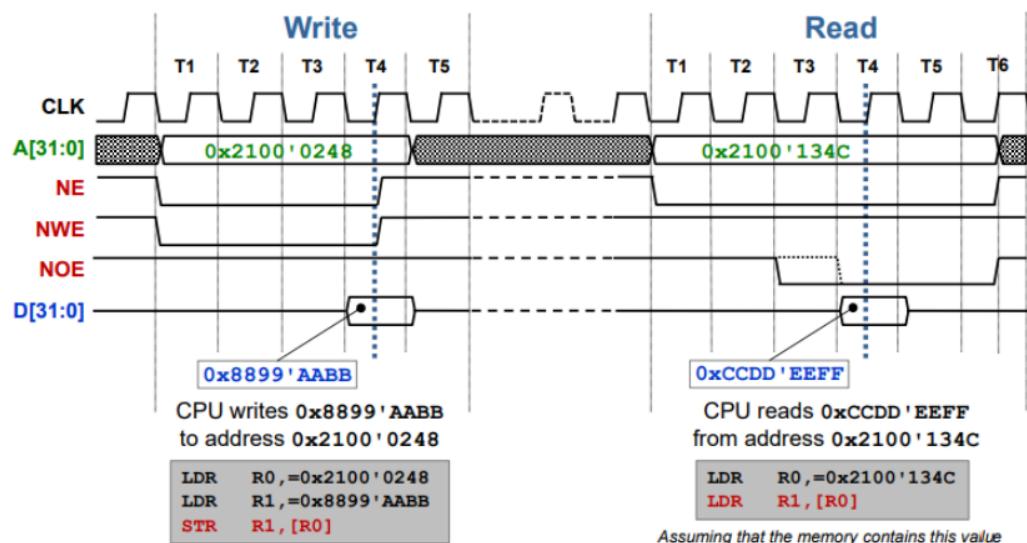
Block Diagram

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



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

CHECK IF CORRECT

GPIO

General Purpose Input Output

Register Address = Base address + Offset

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

GPIO

Situation:

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

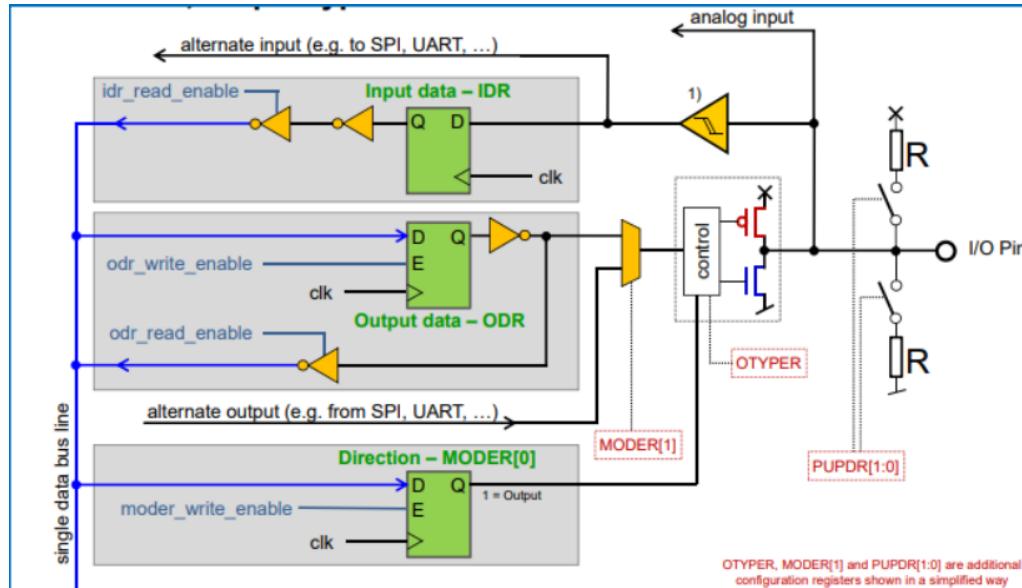
Problem:

- 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

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

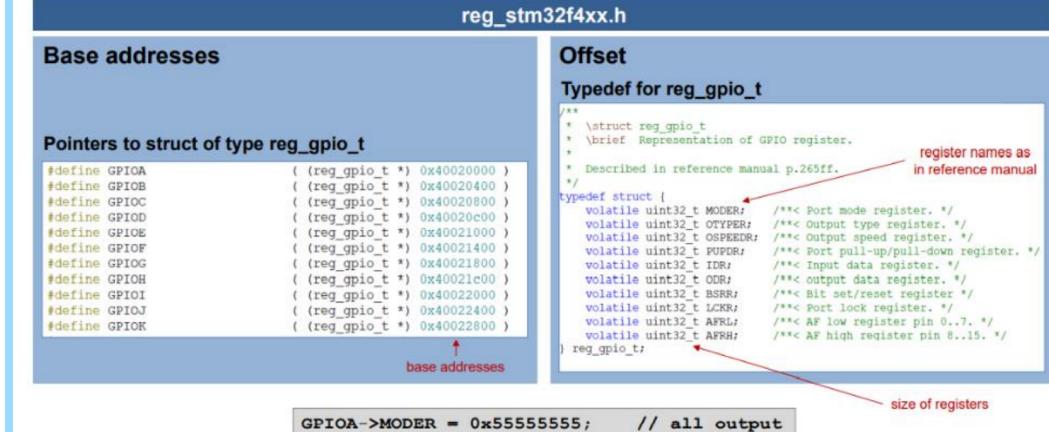
Hardware Abstraction Layer (HAL)

```
#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



Serial Data Transfer

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

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

Overview

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)

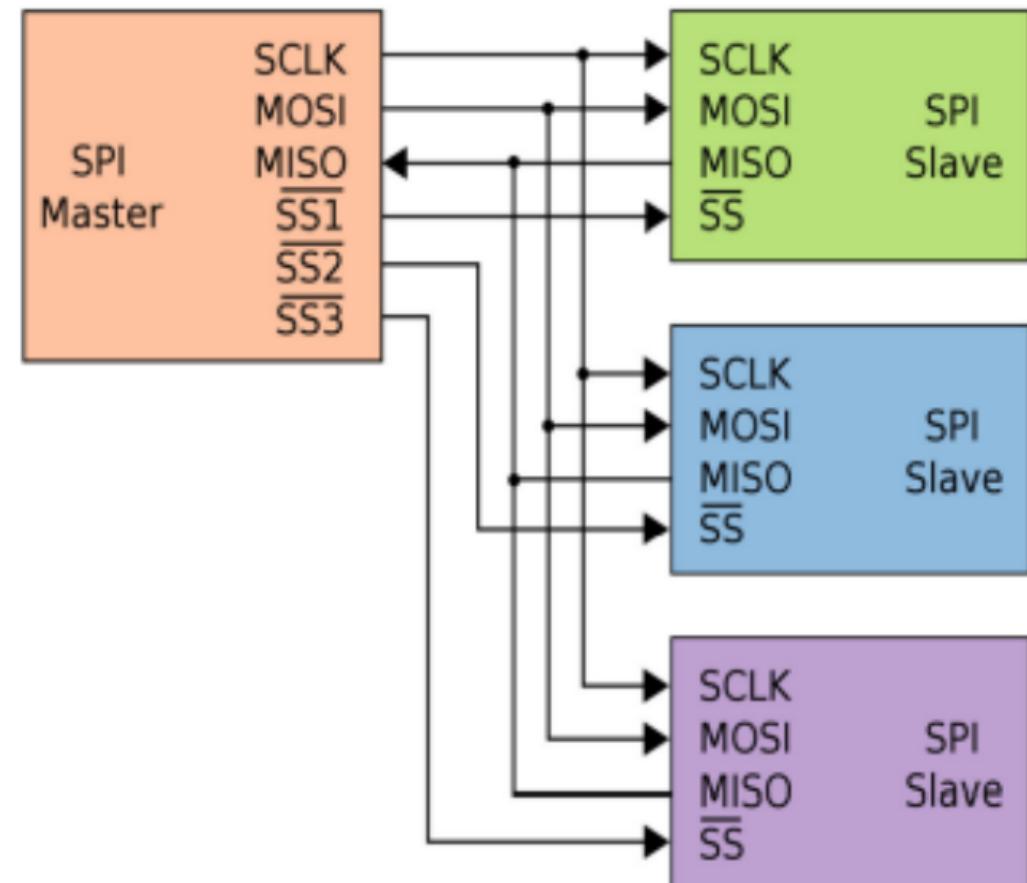
- Transmitter and receiver use diverging clocks
- synchronization using start and stop bits → overhead
- longer connections require line drivers → RS-232/RS-485

I2C - Inter-Integrated Circuit

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

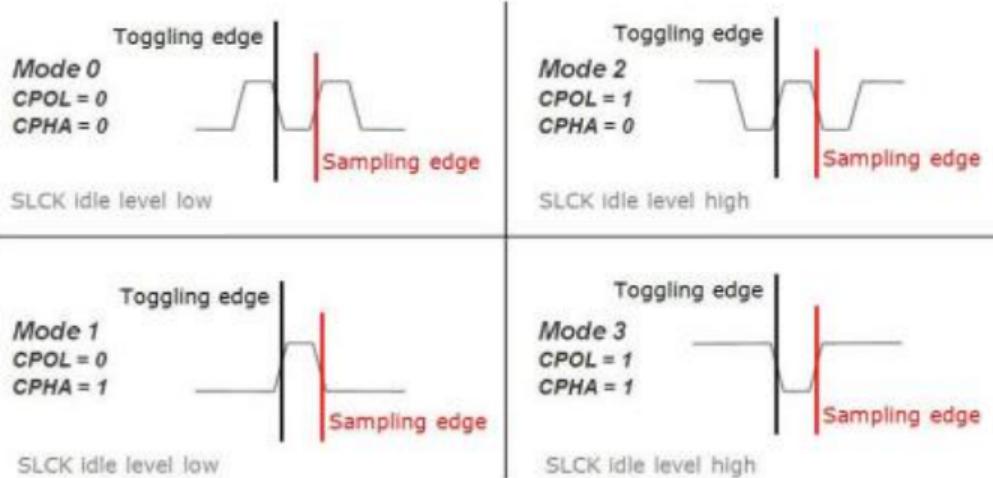
Single Master - Multiple Slaves

- 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 outputs connected to single master input
- Slaves: Selectable by Slave Select (SS) signal
 - Individual Select $\overline{SS1}, \overline{SS2}, \overline{SS3}$
 - $\overline{SSx} = 1 \rightarrow$ slave output $MISO_x$ 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

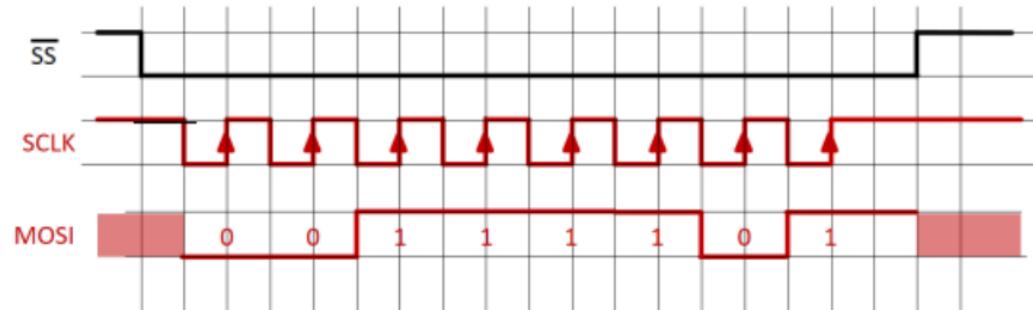
- No defined addressing scheme
 - Use of \overline{SS} instead → KISS (Keep It Simple Stupid)
- Transmission without receive acknowledge and error detection
 - Has to be implemented in higher level protocols
- Originally used only for transmission of single bytes
 - \overline{SS} deactivated after each 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

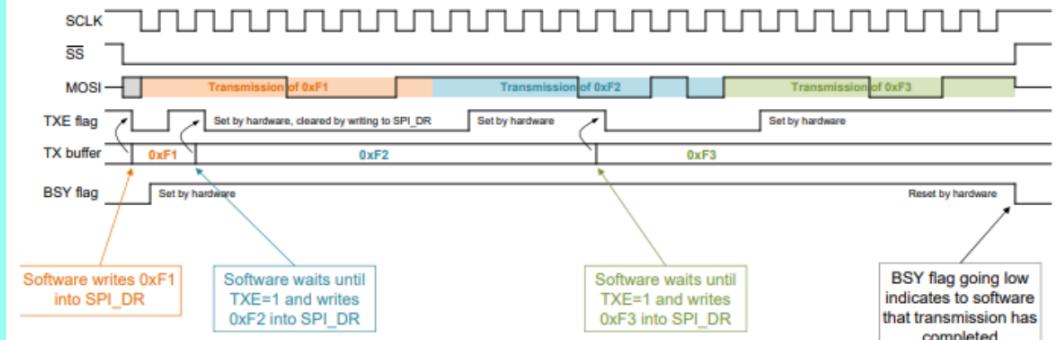


Synchronizing Hardware and Software

- 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

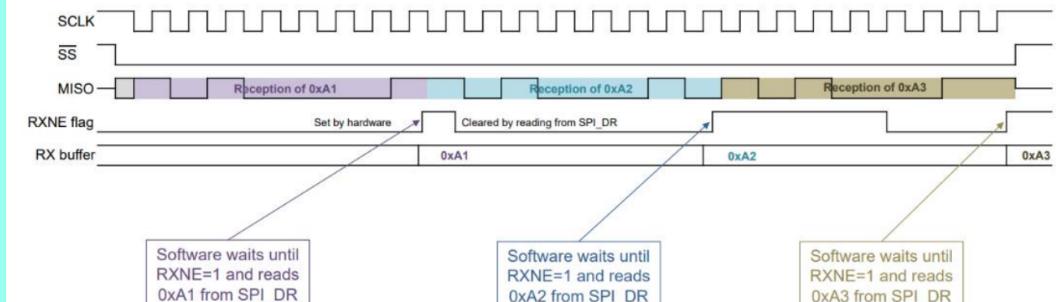
Transmitting in Software

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



Receiving in Software

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

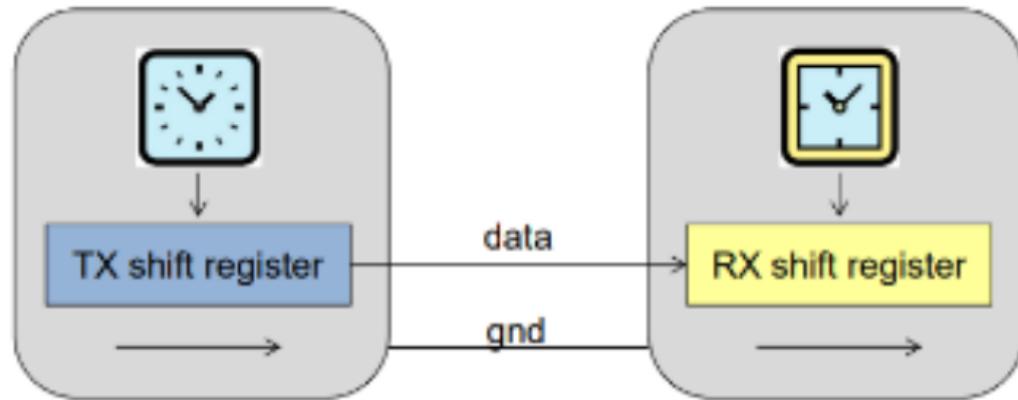


UART/I2C

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