Computertechnik 2

Jil Zerndt
FS 2025

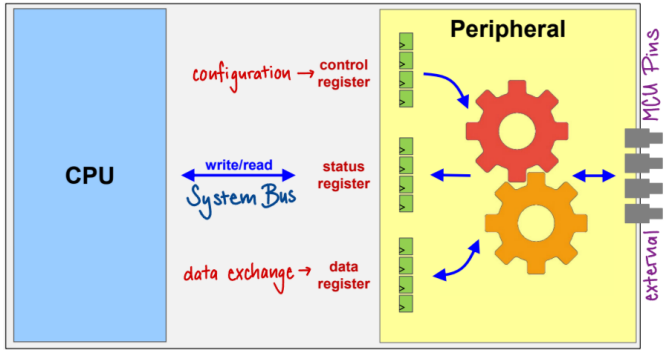
Microcontroller Basics

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



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

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 read/write to peripheral registers

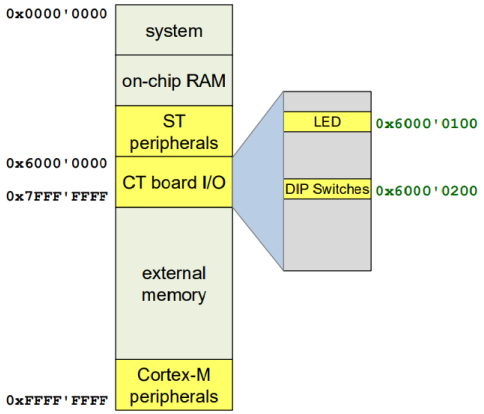
How does the CPU write to and read from peripheral registers?

- 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

Memory-mapped Peripheral Registers

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



Memory-Mapped Registers

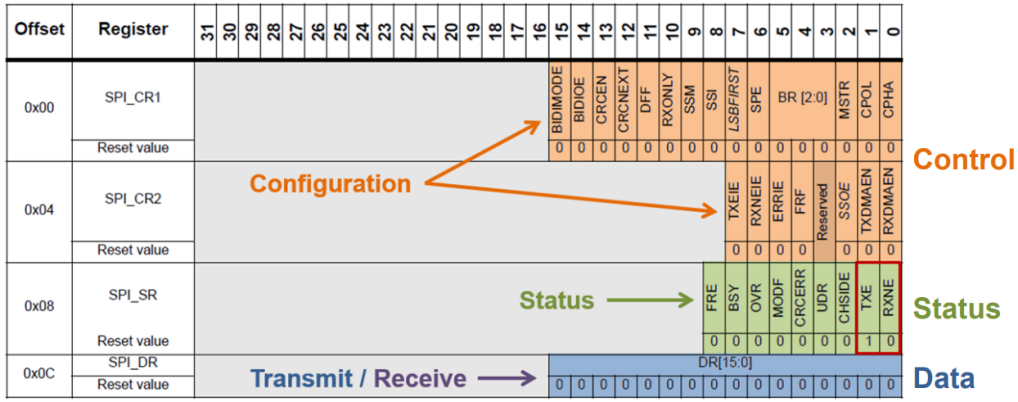
Registers are mapped into the memory address range - each has a specific address:

```
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);
```

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)

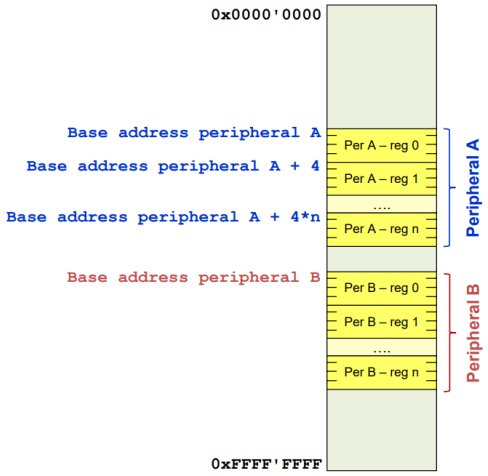


Each column shows a single flip-flop

source: STM32F42xxx Reference Manual

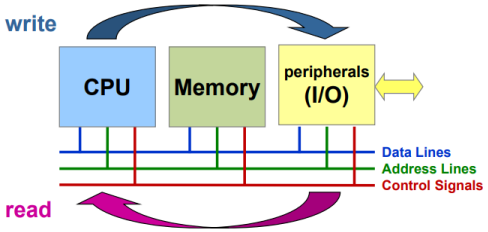
Memory mapping of Peripheral Registers

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



System Bus

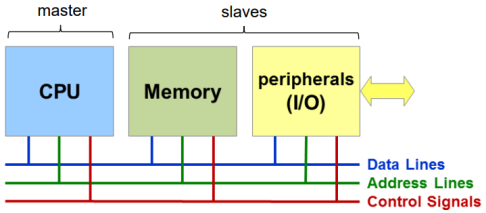
- Interconnects CPU with memory and peripherals, allowing data transfer between components.
- CPU acts as master: initiating and controlling all transfers
- Peripherals and memory act as slaves: responding to requests from the CPU
- System bus is a shared resource



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.



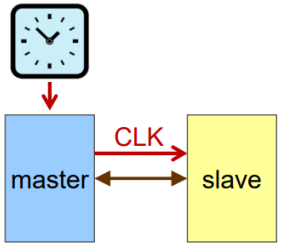
Bus Specification

- **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

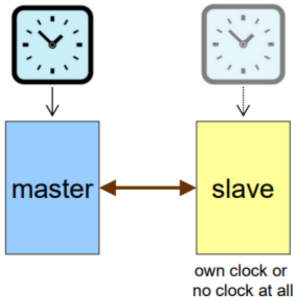
Synchronous

- 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')

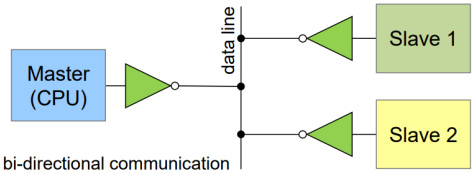
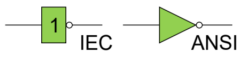
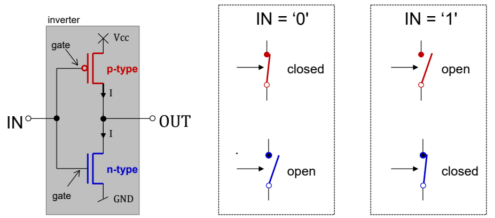


Figure only shows output paths, input paths are not shown.

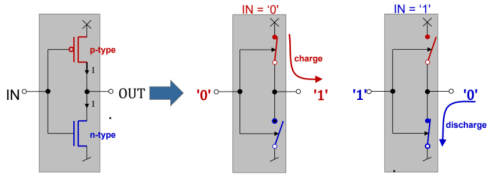
CMOS Inverter



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

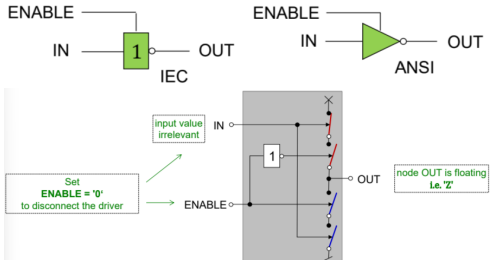


e.g. Vcc = 3V = '1', Gnd = 0V = '0'
Vcc is the supply voltage of the circuit/chip

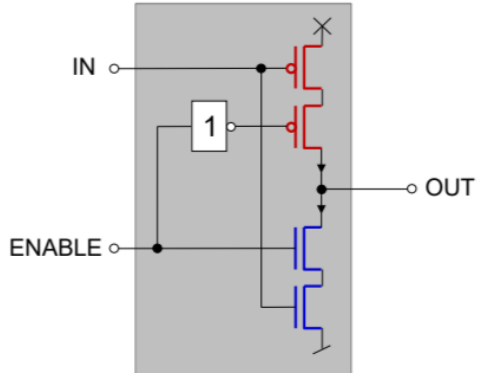


A buffer is built by connecting two inverters in series

CMOS Tri-State Inverter



Implementation:



Tri-State Logic

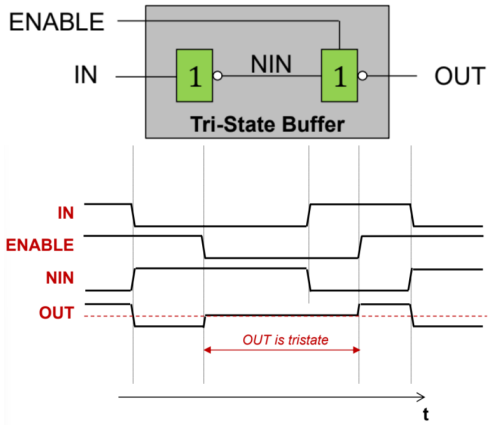
Multiple devices can drive the same data line thanks to tri-state capability:

- 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 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 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

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

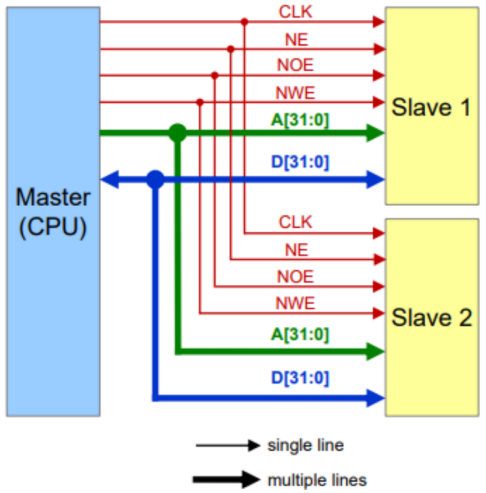
Synchronous Bus Timing

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)

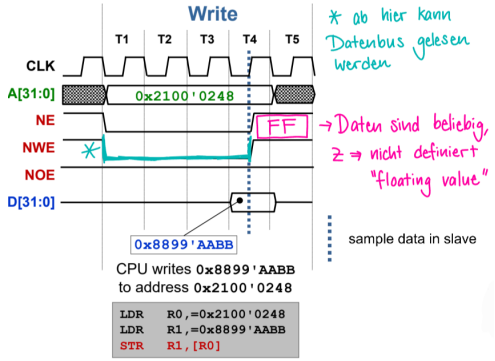
Note that 'N' prefix indicates active-low signals.

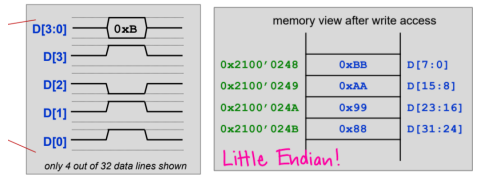
Block Diagram



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)

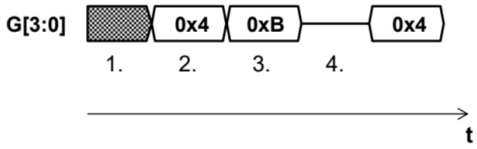




WICHTIG: nicht genau mit Flanke schreiben, Daten brauchen eine gewisse Zeit um stabil zu werden (keine genaue Definition, muss einfach 'genug' sein)
READ dauert länger als WRITE, da die Daten erst stabil werden müssen bevor sie gelesen werden können

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 0×4

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

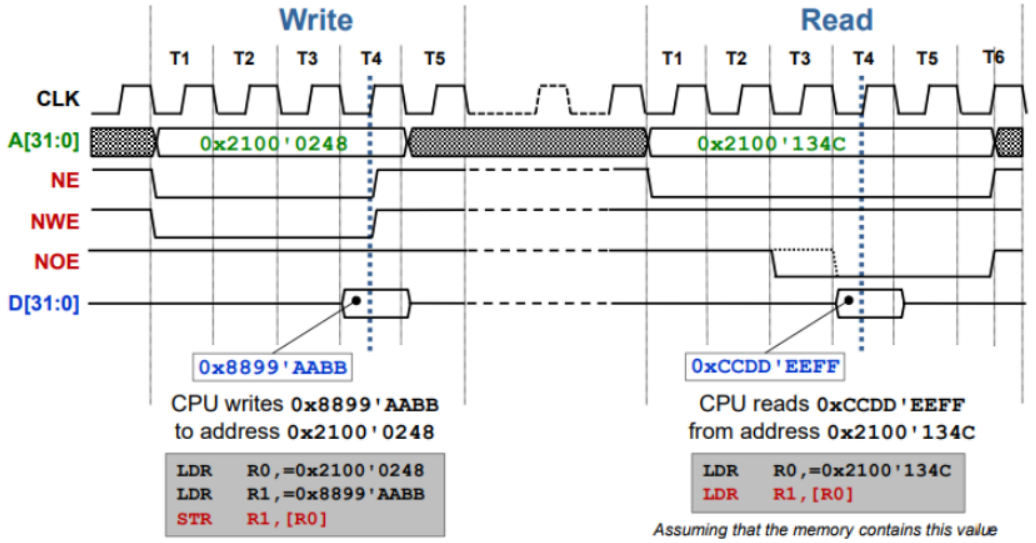
3. The bus holds the value $0 \times B$

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"

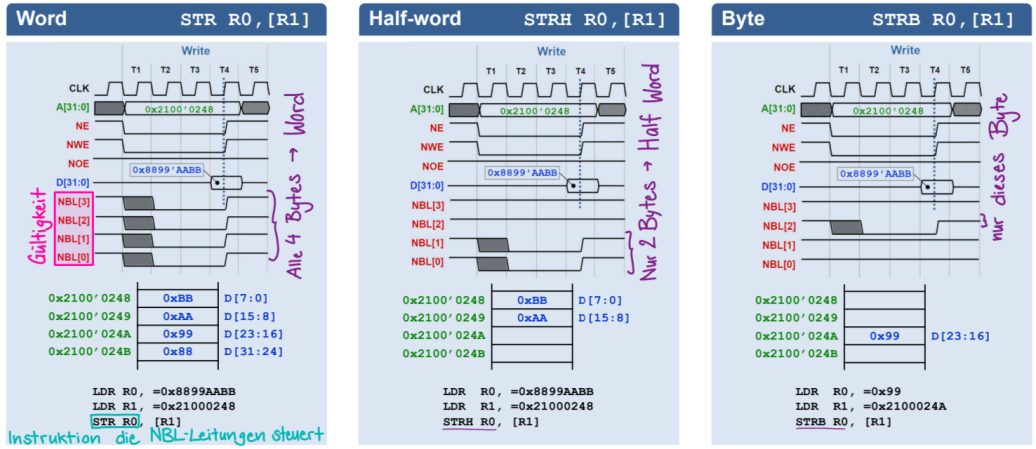
Timing Diagram

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



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

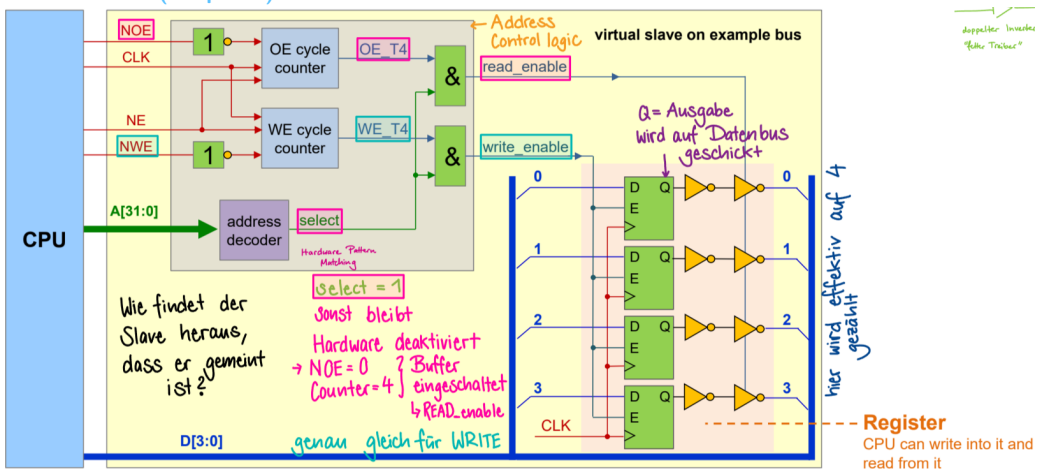


Gültigkeit: damit zeigt die CPU an, welche der 4 Bytes übertragen werden sollen (gültig = 0 (unten))

- 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

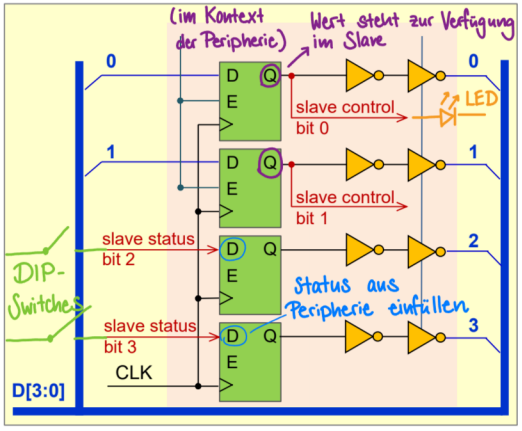
Example STM32 Power Control/Status Register PWR_CSR

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

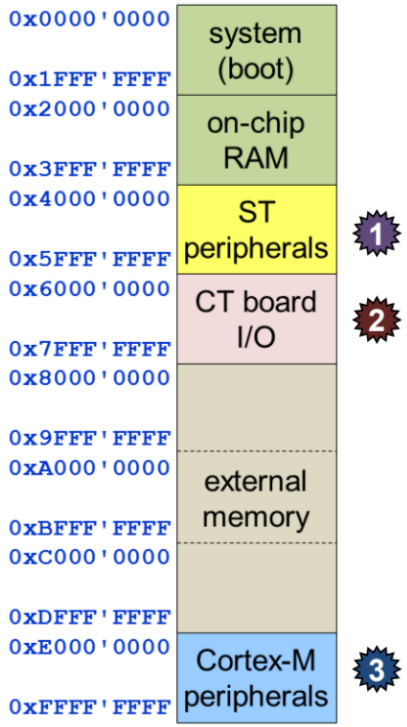
Control/Status register on example bus:



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)



1. ST peripherals
   e.g. Timers, ADC, UART, SPI, …
2. CT board I/O
   LEDs, dip switches, LCD, …
3. ARM Cortex-M
   NVIC, …

Zwiebelbild:

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

STM : ST-Peripherals
→ Timers, SPI, IRC, 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

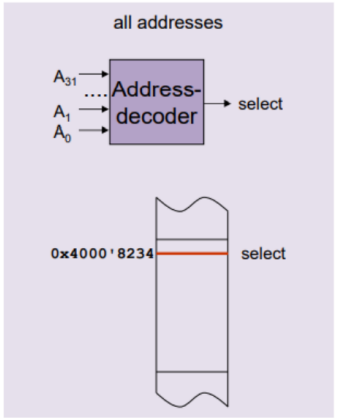
Full Address Decoding

- 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

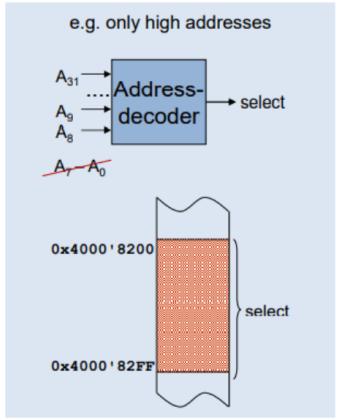
Partial Address Decoding

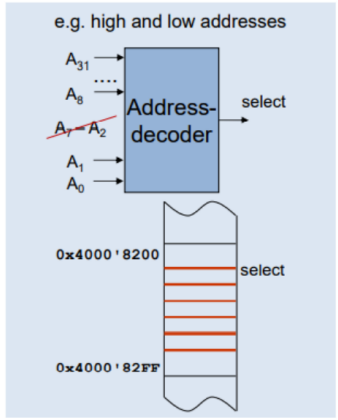
- Only a subset of address lines are decoded
- A control register can be accessed at multiple locations
- 1:n mapping: Multiple addresses map to the same hardware register
- Motivation: Simpler and possible Aliasing (Map a hardware register to several addresses)

Full Address Decoding



Partial Address Decoding





Simple Address Decoder

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

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

Address Decoding Exercise

Given a system bus with 6 address lines A[5:0], determine the address ranges if only bits A[5:4] are decoded.

This means the lower 4 bits (A[3:0]) are not decoded, resulting in a partial address decoding.
Each decoded address represents a range of $2^4 = 16$ addresses.
If A[5:4] = 01, the corresponding address range is: - Start: 0x10 (binary: 010000) - End: 0x1F (binary: 011111)
All addresses in this range (0x10 through 0x1F) will select the same device.

Address Range Calculation

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

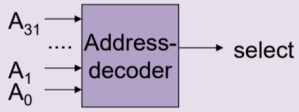
- Each decoded address represents a range of 2^m addresses
- The size of this address range is 2^m bytes
- The start address has all lower bits set to 0
- The end address has all lower bits set to 1

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

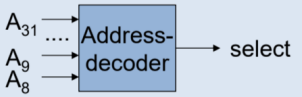
- all addresses from A31 to A0



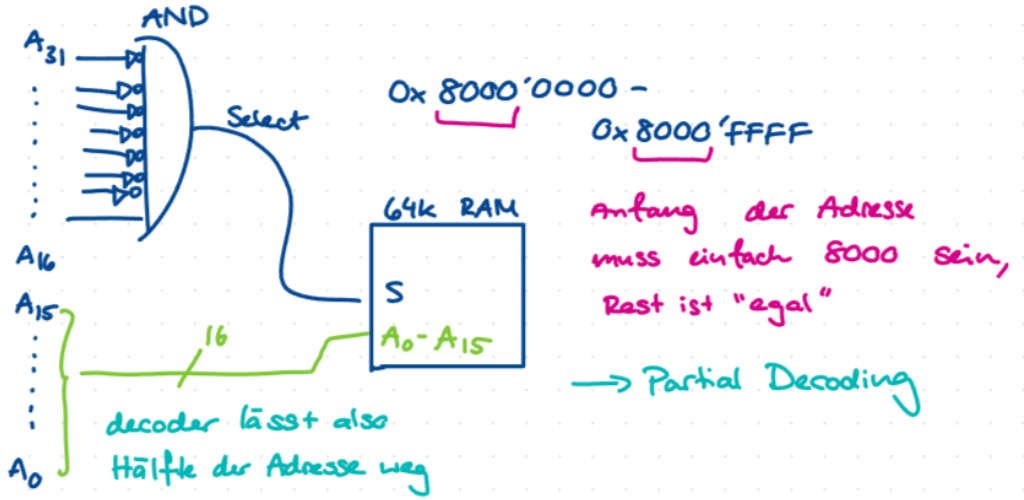
- select is active for exactly one address
- E.g. at `0x4000'8234`

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`



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 "readyßignal (for long or variable access times)

Slow Slaves

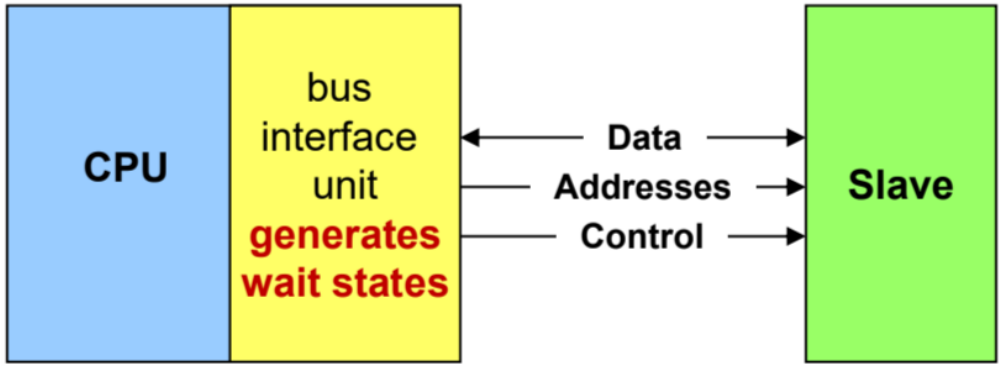
Problem: Individual Slave Access times

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

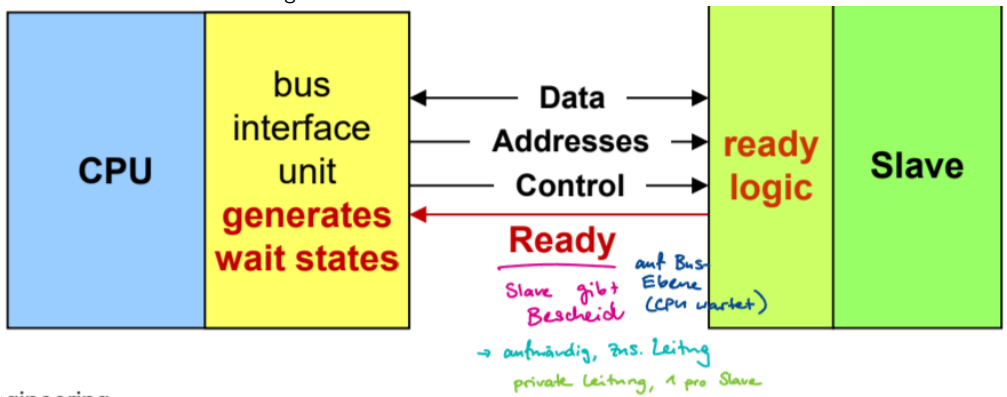
Solutions for slow slaves

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)

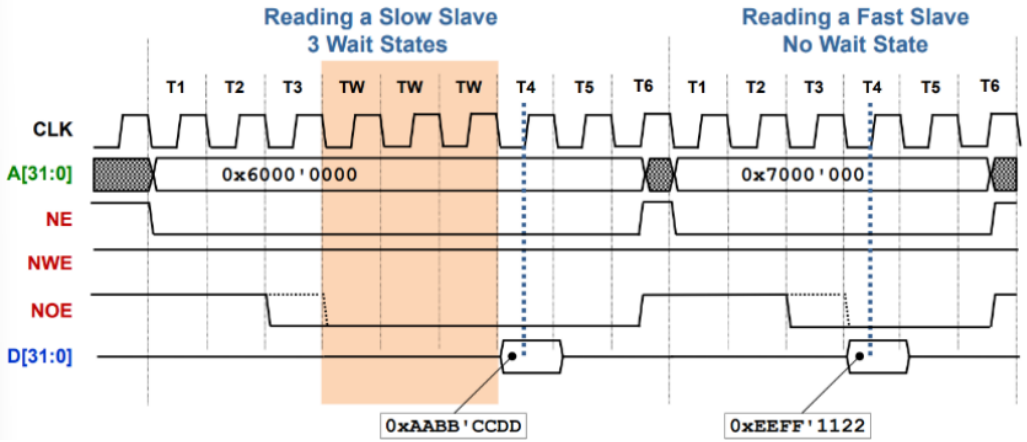


- **Bus Mastering** Slave tells bus interface unit when it is ready
  Allow a peripheral to take control of the bus and perform its own accesses (e.g. DMA)
  Well suited for slaves with long or variable access times



Individual Wait States

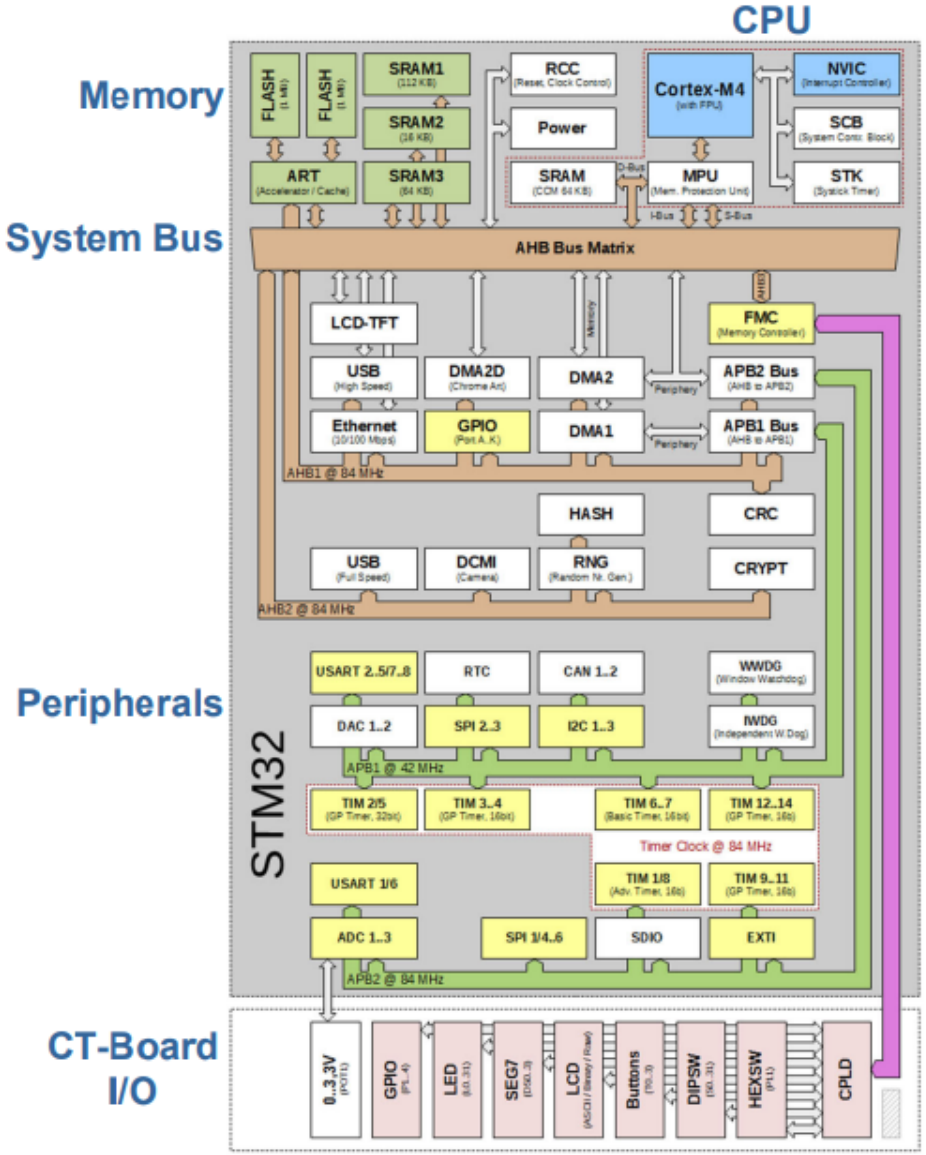
Wait states are inserted to slow down the CPU to match the speed of the slowest peripheral (depending on the address of an access)



STM32 Microcontroller

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

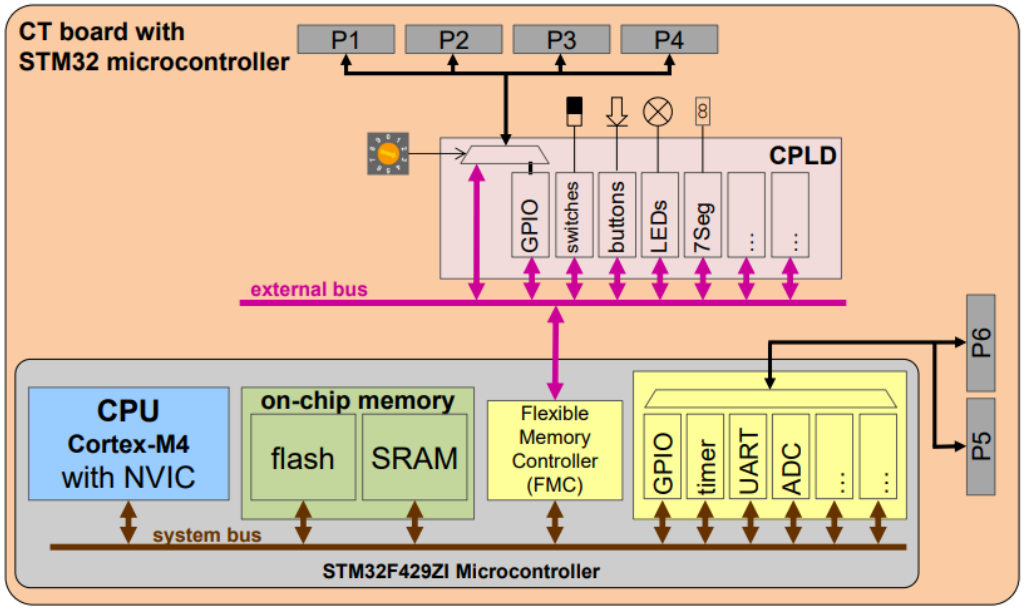
- On-chip system bus: 32 data lines, 32 address lines and control signals
- Off-chip external bus: 16 data lines, 24 address lines and control signals



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

Hardware View

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

Software View

- Accessing control and status registers in C

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

Problem

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

```
uint32_t ui;

void ex_func(void)
{
    ui = 0xAAAAAAAA;
    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 = 0xAAAAAAAA;
    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

Using volatile for register access

```
// Without volatile, compiler might remove these statements
uint32_t ui;
void ex_func(void) {
    ui = 0xAAAAAAAA;  // Appears to have no effect
    ui = 0xBBBBBBBB;  // Appears to have no effect
    while (ui == 0) {
        // ...
    }
}

// With volatile, all accesses are preserved
volatile uint32_t ui;
void ex_func(void) {
    ui = 0xAAAAAAAA;  // Will be executed
    ui = 0xBBBBBBBB;  // Will be executed
    while (ui == 0) {
        // ...
    }
}
```

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.

```
// Create a pointer to volatile uint32_t
volatile uint32_t *p_reg;

// Set LEDs
p_reg = (volatile uint32_t *)(0x60000100);
*p_reg = 0xAA55AA55;  // Write pattern to LEDs

// Wait for DIP switches to be non-zero
p_reg = (volatile uint32_t *)(0x60000200);
while (*p_reg == 0) {
    // Wait until any switch is pressed
}
```

Access through Pointers

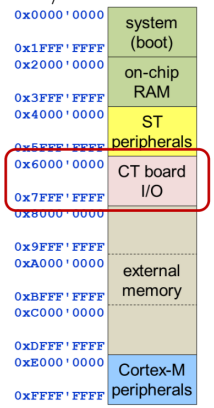
e.g. writing to and reading from CT Board I/O

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

// 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 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.

```
// Define register macros
#define LED31_0_REG     (*((volatile uint32_t *)(0x60000100)))
#define BUTTON_REG      (*((volatile uint32_t *)(0x60000210)))

// Write to LED register
LED31_0_REG = 0xBBCCDDEE;

// Read button register
uint32_t aux_var = BUTTON_REG;
```

Using Preprocessor Macros → #define

```
#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

GPIO (General Purpose I/O)

GPIO Fundamentals

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.)

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

GPIO Hardware Structure

Each GPIO pin contains several hardware components:

- Input data register (IDR) - reads the pin state
- Output data register (ODR) - sets the output state
- Direction control (MODER) - configures pin as input or output
- Output type control (OTYPER) - push-pull or open-drain
- Pull-up/pull-down resistors (PUPDR)
- Speed configuration (OSPEEDR)
- Alternate function selection (AFRL/AFRH)

GPIO Registers

Each GPIO port has several configuration registers:

- **GPIOx_MODER**: Mode register (input, output, alternate function, analog)
- **GPIOx_OTYPER**: Output type register (push-pull or open-drain)
- **GPIOx_OSPEEDR**: Output speed register (low, medium, high, very high)
- **GPIOx_PUPDR**: Pull-up/pull-down register
- **GPIOx_IDR**: Input data register (read-only)
- **GPIOx_ODR**: Output data register (read/write)
- **GPIOx_BSRR**: Bit set/reset register (atomic operations)
- **GPIOx_LCKR**: Configuration lock register
- **GPIOx_AFRL/H**: Alternate function registers

GPIO Configuration

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

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

Push-Pull vs. Open-Drain

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

Input Data Register (IDR)

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

Output Data Register (ODR)

The GPIOx_ODR can be read and written to set the output state of GPIO pins.

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.

Configuring a GPIO Pin

Step 1: Enable GPIO clock

Enable the clock to the GPIO port using RCC register.

Step 2: Configure pin direction

Set the mode register (MODER) to configure as input, output, etc.

Step 3: Configure additional properties

Configure output type, speed, and pull-up/down as needed.

Step 4: Set initial state (for outputs)

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

```
// Step 1: Enable GPIOA clock
RCC->AHB1ENR |= (1 << 0);  // Set bit 0 for GPIOA

// Step 2: Configure PA5 as output (bits 10-11 = 01)
GPIOA->MODER &= ~(3 << 10);  // Clear bits 10-11
GPIOA->MODER |= (1 << 10);   // Set as output

// Step 3: Configure as push-pull, high speed, no pull-up/down
GPIOA->OTYPER &= ~(1 << 5);      // Push-pull (clear bit 5)
GPIOA->OSPEEDR |= (3 << 10);     // Very high speed (set bits 10-11)
GPIOA->PUPDR &= ~(3 << 10);      // No pull-up/down (clear bits 10-11)

// Step 4: Set initial state (turn on LED)
GPIOA->ODR |= (1 << 5);          // Set PA5 high
// OR: Atomic set
GPIOA->BSRR = (1 << 5);          // Set PA5 high
```

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.

```
// Reading input from GPIOA pin 0
uint32_t buttonState = (GPIOA->IDR & (1 << 0)) != 0;

// Writing output using ODR (not atomic)
// Set pin high
GPIOA->ODR |= (1 << 5);
// Set pin low
GPIOA->ODR &= ~(1 << 5);

// Writing output using BSRR (atomic)
// Set pin high (bits 0-15)
GPIOA->BSRR = (1 << 5);
// Set pin low (bits 16-31)
GPIOA->BSRR = (1 << (5 + 16));
```

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.

```
// Enable GPIOA clock
RCC->AHB1ENR |= (1 << 0);  // Bit 0 for GPIOA

// Configure PA3 as output (bits 6-7 = 01)
GPIOA->MODER \&= ~(3 << 6);  // Clear bits 6-7
GPIOA->MODER |= (1 << 6);   // Set bit 6 (output mode)

// Configure as open-drain (bit 3 = 1)
GPIOA->OTYPER |= (1 << 3);

// Configure as low speed (bits 6-7 = 00)
GPIOA->OSPEEDR \&= ~(3 << 6);  // Clear bits 6-7 (low speed)

// Configure with pull-up (bits 6-7 = 01)
GPIOA->PUPDR \&= ~(3 << 6);    // Clear bits 6-7
GPIOA->PUPDR |= (1 << 6);     // Set bit 6 (pull-up)

// Set pin to low state using BSRR
GPIOA->BSRR = (1 << (3 + 16));  // Set bit 19 (reset PA3)
```

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.

Using HAL for GPIO

```
// Using HAL style access
#include "reg_stm32f4xx.h"  // Contains structure definitions

// Configure PA3 as output
GPIOA->MODER &= ~(3 << 6);  // Clear bits 6-7
GPIOA->MODER |= (1 << 6);   // Set bit 6 (output mode)

// Instead of direct register access:
// volatile uint32_t *reg = (volatile uint32_t *)(0x40020000);
// *reg &= ~(3 << 6);
// *reg |= (1 << 6);
```

Serial Data Transfer - SPI

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

SPI Overview

SPI - Serial Peripheral Interface

SPI is a synchronous serial bus primarily for on-board connections:

- Introduced by Motorola (now NXP) around 1979
- Also called 4-wire bus
- De facto standard (no legally binding specification)
- Used for short-distance communication
- Connects microcontroller to external devices (sensors, displays, flash memory, etc.)
- Full-duplex communication
- Single master, multiple slaves
- Synchronous (master provides clock)

SPI Signals

- **SCLK** (Serial Clock): Generated by master
- **MOSI** (Master Out Slave In): Data from master to slave
- **MISO** (Master In Slave Out): Data from slave to master
- **SS/CS** (Slave Select/Chip Select): Enables specific slave(s)

SPI Master-Slave Architecture

- Master generates common clock signal (SCLK) for all slaves
- Master selects slave by asserting the appropriate SS line (active low)
- 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 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

SPI Modes

Clock Polarity and Phase

SPI has four different modes based on two parameters:

- **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 1**: CPOL = 0, CPHA = 1
  - Clock idles low
  - Data sampled on falling edge, changed on rising 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

SPI Characteristics

SPI Properties

- No defined addressing scheme (uses SS lines instead)
- No built-in acknowledgment or error detection
- Originally for single-byte transfers (now also used for streaming)
- Flexible data rate (clock signal is transmitted with data)
- No flow control (master controls timing, slave cannot influence)
- Susceptible to clock line noise

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

STM32 SPI Register Configuration

```
// SPI configuration example
// Configure SPI1 in master mode, CPOL=1, CPHA=1, 8-bit data

// Enable SPI1 clock
RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;

// Configure SPI1
SPI1->CR1 = (1 << 0)   // CPHA=1
          | (1 << 1)   // CPOL=1
          | (1 << 2)   // Master mode
          | (3 << 3)   // BR[2:0]=011: fPCLK/16 (prescaler)
          | (0 << 7)   // MSB first (LSBFIRST=0)
          | (1 << 8)   // SSI=1 (needed for software SS)
          | (1 << 9);  // SSM=1 (software slave management)

// Enable SPI
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.

```
// Transmit a byte over SPI
uint8_t transmit_byte(uint8_t data) {
    // Step 1: SPI should already be configured

    // Step 2: Wait until TXE=1 (transmit buffer empty)
    while (!(SPI1->SR & (1 << 1))) { }

    // Step 3: Write data to transmit
    SPI1->DR = data;

    // Step 4: Wait for reception (needed to get received data)
    while (!(SPI1->SR & (1 << 0))) { }

    // Return received data (read DR clears RXNE flag)
    return SPI1->DR;
}
```

Handling Full-Duplex SPI Communication

Step 1: Enable SPI with SPE bit

Ensure SPI is properly configured and enabled.

Step 2: Write first byte to transmit

Write the first byte to DR to start transmission/reception.

Step 3: Process data in a loop

Check TXE and RXNE flags to handle both transmission and reception.

Step 4: Wait for completion

Wait for BSY=0 to ensure all transfers are complete.

```
// Full-duplex SPI communication
void spi_transfer(uint8_t *tx_data, uint8_t *rx_data, uint16_t length) {
    // Enable SPI
    SPI1->CR1 |= (1 << 6);  // SPE=1

    // Write first TX byte to start transmission
    if (length > 0) {
        SPI1->DR = tx_data[0];
    }

    uint16_t tx_count = 1;
    uint16_t rx_count = 0;

    // Process data
    while (rx_count < length) {
        // Check if we can transmit more data
        if ((tx_count < length) && (SPI1->SR & (1 << 1))) {  // TXE=1
            SPI1->DR = tx_data[tx_count++];
        }

        // Check if we received data
        if (SPI1->SR & (1 << 0)) {  // RXNE=1
            rx_data[rx_count++] = SPI1->DR;
        }
    }

    // Wait until BSY=0 (SPI not busy)
    while (SPI1->SR & (1 << 7)) { }
}
```

SPI Mode Identification from Timing Diagram

Given the following timing diagram, identify the SPI mode (CPOL/CPHA values): **add actual timing diagram here**

Data: 1 0 1 0 1 1 0 1

Looking at the diagram: - Clock starts low and returns to low between transfers (CPOL = 0) - Data appears to change on rising edges - Data is sampled on falling edges
This corresponds to Mode 1: CPOL = 0, CPHA = 1

Serial Data Transfer - UART & I2C

UART Fundamentals

UART - Universal Asynchronous Receiver Transmitter

UART is an asynchronous serial communication interface:

- Asynchronous - no shared clock between transmitter and receiver
- 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

UART Signals

Basic UART connections require:

- **TX** (Transmit): Data output from transmitter to receiver
- **RX** (Receive): Data input from transmitter to receiver
- **GND** (Ground): Common reference level

Extended UART (with hardware flow control) may include:

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

UART Data Format and Timing

UART Frame Structure

A UART frame consists of:

- **Idle state**: Line is high ('1') when no transmission occurs
- **Start bit**: Always '0', indicates 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'

UART Synchronization

Without a common clock, UART devices synchronize as follows:

- 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 Diagram Analysis

Given a UART signal with 8 data bits, no parity, 1 stop bit, analyze this pattern:

I I S D D D D D D D E I I

Where I=Idle, S=Start, D=Data, E=Stop

Breaking down this signal: - Idle (high) → Start bit (low) → 8 data bits → Stop bit (high) → Idle (high) - Reading data bits (LSB first): 01010011 - Converting to binary: 0b01010011 = 0x53 = ASCII 'S' This UART transmission contains the character 'S'.

UART Calculations

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

Frame Time (seconds): $T_{frame} = T_{bit} \times (1 + \text{Data Bits} + \text{Parity Bits} + \text{Stop Bits})$

Maximum Data Rate (bytes/second): $\text{Data Rate} = \frac{\text{Baud Rate}}{\text{Total Bits per Byte}}$

where Total Bits per Byte = 1 (start) + Data Bits + Parity Bits + Stop Bits

Clock Tolerance in UART

Maximum clock deviation

The receiver must sample correctly until the last bit:

Formula

Maximum allowed clock deviation (as percentage): $\text{Deviation}_{\max} = \frac{0.5}{N} \times 100\%$

where N is the number of bit times between synchronization points.

For a standard UART frame (1 start, 8 data, 1 stop): Synchronization occurs at start bit, Last bit (stop bit) is 10 bits later

Maximum clock deviation = 0.5/10 = 5%

Sample calculation:

If sender clock is 9600 Hz: Receiver can be between 9120 Hz and 10080 Hz

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. Enable UART2 and GPIO clock
RCC->APB1ENR |= RCC_APB1ENR_USART2EN;  // Enable UART2 clock
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;   // Enable GPIOA clock
// 2. Configure GPIO pins for UART (PA2 = TX, PA3 = RX)
// Set alternate function mode (0x10)
GPIOA->MODER &= ~(0x3 << (2*2) | 0x3 << (2*3));
GPIOA->MODER |= (0x2 << (2*2) | 0x2 << (2*3));
// Set to AF7 (UART2)
GPIOA->AFR[0] &= ~(0xF << (4*2) | 0xF << (4*3));
GPIOA->AFR[0] |= (0x7 << (4*2) | 0x7 << (4*3));
// 3. Configure UART
// Reset UART configuration
USART2->CR1 = 0;
USART2->CR2 = 0;
USART2->CR3 = 0;
// Set baud rate (assuming 84MHz APB1 clock)
// BRR = f_CK / baud rate = 84,000,000 / 115,200 = 729.16
USART2->BRR = 729;
// Enable UART, TX, and RX
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.

```
// Send a character
void UART_SendChar(USART_TypeDef *uart, char c) {
    // Wait until TXE flag is set (transmit buffer empty)
    while (!(uart->SR & USART_SR_TXE)) { }
    // Write the character to the data register
    uart->DR = c;
}
// Send a string
void UART_SendString(USART_TypeDef *uart, const char *str) {
    while (*str) {
        UART_SendChar(uart, *str++);
    }
}
// Receive a character (blocking)
char UART_ReceiveChar(USART_TypeDef *uart) {
    // Wait until RXNE flag is set (data received)
    while (!(uart->SR & USART_SR_RXNE)) { }
    // Return received data
    return uart->DR;
}
```

I2C Fundamentals

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 Protocol

I2C Bus Conditions

Special conditions on the I2C bus:

- **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 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.

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 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 on STM32F4

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

```
// Configure I2C1 for 100kHz standard mode

// 1. Enable I2C1 and GPIO clock
RCC->APB1ENR |= RCC_APB1ENR_I2C1EN;   // Enable I2C1 clock
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;  // Enable GPIOB clock

// 2. Configure GPIO pins for I2C (PB6 = SCL, PB7 = SDA)
// Set alternate function mode (0x10)
GPIOB->MODER &= ~(0x3 << (2*6) | 0x3 << (2*7));
GPIOB->MODER |= (0x2 << (2*6) | 0x2 << (2*7));

// Set open-drain output type
GPIOB->OTYPER |= (1 << 6) | (1 << 7);

// Set to AF4 (I2C1)
GPIOB->AFR[0] &= ~(0xF << (4*6) | 0xF << (4*7));
GPIOB->AFR[0] |= (0x4 << (4*6) | 0x4 << (4*7));

// 3. Reset I2C
I2C1->CR1 = I2C_CR1_SWRST;  // Software reset
I2C1->CR1 = 0;              // Clear reset

// 4. Configure I2C
// Set peripheral clock frequency (42MHz)
I2C1->CR2 = 42;  // FREQ = 42MHz

// Set CCR for 100kHz standard mode
// CCR = PCLK1 / (2 * I2C_FREQ) = 42MHz / (2 * 100kHz) = 210
I2C1->CCR = 210;

// Set TRISE
// TRISE = (max rise time * PCLK1) + 1 = (1000ns * 42MHz) + 1 = 43
I2C1->TRISE = 43;

// 5. Enable I2C
I2C1->CR1 |= I2C_CR1_PE;  // Peripheral enable
```

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.

```
// I2C Master Transmit
void I2C_MasterTransmit(I2C_TypeDef *i2c, uint8_t address, uint8_t *data, uint16_t
    size) {
    // 1. Wait until I2C bus is free
    while (i2c->SR2 & I2C_SR2_BUSY) { }

    // 2. Generate START condition
    i2c->CR1 |= I2C_CR1_START;

    // 3. Wait for EV5 (START sent)
    while (!(i2c->SR1 & I2C_SR1_SB)) { }

    // 4. Send slave address (write mode)
    i2c->DR = address << 1;  // Address + Write bit (0)

    // 5. Wait for EV6 (address sent)
    while (!(i2c->SR1 & I2C_SR1_ADDR)) { }

    // 6. Clear ADDR by reading SR2
    (void)i2c->SR2;

    // 7. Send data bytes
    for (uint16_t i = 0; i < size; i++) {
        // Wait until TXE is set
        while (!(i2c->SR1 & I2C_SR1_TXE)) { }

        // Send data byte
        i2c->DR = data[i];
    }

    // 8. Wait for transfer to complete
    while (!(i2c->SR1 & I2C_SR1_TXE) || !(i2c->SR1 & I2C_SR1_BTF)) { }

    // 9. Generate STOP condition
    i2c->CR1 |= I2C_CR1_STOP;
}
```

Comparison of Serial Interfaces

UART vs. SPI vs. I2C Comparison

Feature	UART	SPI	I2C
Wires	2 (TX, RX)	4 (MOSI, MISO, SCLK, SS)	2 (SDA, SCL)
Clock	Asynchronous	Synchronous	Synchronous
Connection Type	Point-to-point	Point-to-multipoint	Multi-point
Duplex	Full-duplex	Full-duplex	Half-duplex
Addressing	Higher layer only	Device selection via SS	7/10-bit addressing
Error Detection	Optional parity bit	None	ACK/NACK
Speed	Up to 5 Mbps	Up to 50 Mbps	100 kbps to 5 Mbps
Master/Slave	Peer-to-peer	Single master, multiple slaves	Multiple masters, multiple slaves
Typical Use	Terminal communication, debug ports	On-board high-speed communication	Board-to-board, chip-to-chip

Timer/Counter

Timer/Counter Basics

Timer/Counter Fundamentals

A timer/counter is a digital circuit that counts events:

- **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

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

STM32F4 Timers

STM32F4 Timer Types

The STM32F4 includes several types of timers:

- **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

STM32F4 timers can use different 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

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

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 the 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.

```
// Configure TIM2 for a 1Hz interrupt (1s period)
// Assuming system clock = 84MHz

// Step 1: Enable TIM2 clock
RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;

// Step 2: Configure prescaler
// PSC = (SystemCoreClock / 2) / 10000 - 1
TIM2->PSC = 8399;  // Produces 10kHz timer clock (84MHz/8400)

// Step 3: Configure auto-reload value
// ARR = (timer clock / desired frequency) - 1
TIM2->ARR = 9999;  // 10kHz/10000 = 1Hz

// Step 4: Configure counting mode (up-counting, default)
TIM2->CR1 &= ~TIM_CR1_DIR;  // Up-counting mode

// Step 5: Enable update interrupt
TIM2->DIER |= TIM_DIER_UIE;

// Step 6: Enable timer
TIM2->CR1 |= TIM_CR1_CEN;

// Configure NVIC to handle TIM2 interrupt
NVIC_EnableIRQ(TIM2_IRQn);
NVIC_SetPriority(TIM2_IRQn, 1);
```

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

For a desired output frequency: $ARR = \frac{f_{timer}}{f_{desired}} - 1$

For a desired time period: $ARR = f_{timer} \times T_{desired} - 1$

Timer Period Calculation

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 = 84MHz / 1MHz - 1 = 84 - 1 = 83
Step 2: Calculate the auto-reload value. ARR = f_timer × T_desired - 1 ARR = 1MHz × 0.05s - 1 ARR = 50,000 - 1 = 49,999
However, this exceeds the 16-bit range (maximum 65,535). Let's adjust to use a 10kHz timer clock: PSC = 84MHz / 10kHz - 1 = 8,400 - 1 = 8,399 ARR = 10kHz × 0.05s - 1 = 500 - 1 = 499
Therefore: PSC = 8,399 ARR = 499

Input Capture

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

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.

```
// Configure TIM3 Channel 1 for input capture on rising edge

// Step 1: Configure GPIO pin (PA6 for TIM3_CH1)
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
GPIOA->MODER &= ~GPIO_MODER_MODER6;
GPIOA->MODER |= GPIO_MODER_MODER6_1;  // Alternate function
GPIOA->AFR[0] &= ~GPIO_AFRL_AFRL6;
GPIOA->AFR[0] |= 2 << GPIO_AFRL_AFRL6_Pos;  // AF2 for TIM3

// Step 2: Configure timer base
RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;
TIM3->PSC = 83;      // 84MHz/84 = 1MHz timer clock
TIM3->ARR = 65535;   // Maximum period

// Step 3: Configure capture channel
// CC1S[1:0] = 01 (input, IC1 mapped to TI1)
TIM3->CCMR1 &= ~TIM_CCMR1_CC1S;
TIM3->CCMR1 |= TIM_CCMR1_CC1S_0;

// Configure input filter and prescaler if needed
TIM3->CCMR1 &= ~(TIM_CCMR1_IC1F | TIM_CCMR1_IC1PSC);

// Configure edge sensitivity (rising edge)
TIM3->CCER &= ~(TIM_CCER_CC1P | TIM_CCER_CC1NP);

// Enable capture
TIM3->CCER |= TIM_CCER_CC1E;

// Step 4: Enable capture interrupt
TIM3->DIER |= TIM_DIER_CC1IE;
NVIC_EnableIRQ(TIM3_IRQn);

// Step 5: Enable timer
TIM3->CR1 |= TIM_CR1_CEN;
```

Pulse Width Modulation (PWM)

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 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**: Smallest 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 Generation Using Output Compare

PWM is implemented using the output compare function:

- Counter continuously runs through a 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 is controlled by changing the CCR value
  - Duty cycle = CCR / ARR (in up-counting mode)

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.

```
// Configure TIM3 Channel 1 for PWM output at 1kHz with 50% duty cycle

// Step 1: Configure GPIO pin (PA6 for TIM3_CH1)
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
GPIOA->MODER &= ~GPIO_MODER_MODER6;
GPIOA->MODER |= GPIO_MODER_MODER6_1;  // Alternate function
GPIOA->AFR[0] &= ~GPIO_AFRL_AFRL6;
GPIOA->AFR[0] |= 2 << GPIO_AFRL_AFRL6_Pos;  // AF2 for TIM3

// Step 2: Configure timer base
RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;
TIM3->PSC = 83;     // 84MHz/84 = 1MHz timer clock
TIM3->ARR = 999;    // 1MHz/1000 = 1kHz PWM frequency

// Step 3: Configure PWM mode
// CC1S = 00: Channel configured as output
TIM3->CCMR1 &= ~TIM_CCMR1_CC1S;

// OC1M = 110: PWM mode 1 (active when counter < CCR1)
TIM3->CCMR1 &= ~TIM_CCMR1_OC1M;
TIM3->CCMR1 |= TIM_CCMR1_OC1M_1 | TIM_CCMR1_OC1M_2;

// Enable output compare preload
TIM3->CCMR1 |= TIM_CCMR1_OC1PE;

// Set active high polarity
TIM3->CCER &= ~TIM_CCER_CC1P;

// Step 4: Set initial duty cycle (50%)
TIM3->CCR1 = 500;  // 50% of 1000

// Step 5: Enable output and timer
TIM3->CCER |= TIM_CCER_CC1E;  // Enable output
TIM3->CR1 |= TIM_CR1_CEN;     // Enable counter
```

PWM Duty Cycle Calculation

Up-counting mode:

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

Down-counting mode:

$$\text{Duty Cycle} = \left(1 - \frac{CCR}{ARR+1}\right) \times 100\% \tag{2}$$

To achieve a specific duty cycle:

$$\text{CCR (up-counting)} = (ARR+1) \times \frac{\text{Duty Cycle}}{100\%} \tag{3}$$

$$\text{CCR (down-counting)} = (ARR+1) \times \left(1 - \frac{\text{Duty Cycle}}{100\%}\right) \tag{4}$$

PWM Configuration Exercise

Configure Timer 4 to generate a 20kHz PWM signal with 75

Assuming 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- Up-counting mode, PWM mode 1 - CCR = (ARR+1) * 0.75 = 1050 * 0.75 = 787
3. Configuration code:

```
// Configure GPIO pin (PB7 for TIM4\_CH2)
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;
GPIOB->MODER &= ~GPIO_MODER_MODER7;
GPIOB->MODER |= GPIO_MODER_MODER7_1;  // Alternate function
GPIOB->AFR[0] &= ~GPIO_AFRL_AFRL7;
GPIOB->AFR[0] |= 2 << GPIO_AFRL_AFRL7_Pos;  // AF2 for TIM4

// Configure timer base
RCC->APB1ENR |= RCC_APB1ENR_TIM4EN;
TIM4->PSC = 3;      // 84MHz/4 = 21MHz
TIM4->ARR = 1049;  // 21MHz/1050 = 20kHz

// Configure PWM mode
TIM4->CCMR1 &= ~TIM_CCMR1_CC2S;  // Channel as output
TIM4->CCMR1 &= ~TIM_CCMR1_OC2M;
TIM4->CCMR1 |= TIM_CCMR1_OC2M_1 | TIM_CCMR1_OC2M_2;  // PWM mode 1
TIM4->CCMR1 |= TIM_CCMR1_OC2PE;  // Enable preload
TIM4->CCER &= ~TIM_CCER_CC2P;    // Active high

// Set duty cycle (75%)
TIM4->CCR2 = 787;

// Enable output and timer
TIM4->CCER |= TIM_CCER_CC2E;
TIM4->CR1 |= TIM_CR1_CEN;
```

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:

$$V_{LSB} = \frac{V_{REF}}{2^N} \tag{5}$$

Digital Output Value:

$$D_{out} = \frac{V_{in} \times 2^N}{V_{REF}} \tag{6}$$

Analog Input from Digital Value:

$$V_{in} = \frac{D_{out} \times V_{REF}}{2^N} \tag{7}$$

Percent Full Scale Range:

$$\%FSR = \frac{V_{in}}{V_{REF}} \times 100\% \tag{8}$$

STM32F4 ADC Features

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_{REFINT}, V_{BAT})
- Multiple operating modes:
  - Single conversion vs. continuous conversion
  - Single channel vs. scan mode (multiple channels)
- Maximum sampling rate up to 2.4 MSPS (million samples per second)
- DMA capability
- Configurable sampling time
- Analog watchdog for threshold monitoring

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)

STM32F4 ADC Configuration

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)

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.

```
// Configure ADC1 Channel 0 (PA0) for single conversion

// Step 1: Enable GPIO and ADC clocks
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;  // Enable GPIOA clock
RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;   // Enable ADC1 clock

// Step 2: Configure PA0 as analog input
GPIOA->MODER |= GPIO_MODER_MODER0;  // Analog mode (0b11)

// Step 3: Configure ADC parameters
// ADC Common Control Register
ADC->CCR &= ~ADC_CCR_ADCPRE;  // ADCPRE = 0 (APB2/2, typically 42MHz/2 = 21MHz)

// ADC1 Control Register 1
ADC1->CR1 &= ~ADC_CR1_RES;    // 12-bit resolution (default)
ADC1->CR1 &= ~ADC_CR1_SCAN;   // Disable scan mode (single channel)

// ADC1 Control Register 2
ADC1->CR2 &= ~ADC_CR2_CONT;   // Single conversion mode
ADC1->CR2 &= ~ADC_CR2_ALIGN;  // Right alignment
ADC1->CR2 &= ~ADC_CR2_EXTEN;  // Software trigger

// Step 4: Configure channel and sampling time
// Configure for channel 0
ADC1->SQR1 &= ~ADC_SQR1_L;    // 1 conversion in regular sequence
ADC1->SQR3 &= ~ADC_SQR3_SQ1;  // Clear channel selection
ADC1->SQR3 |= (0 << ADC_SQR3_SQ1_Pos);  // Channel 0 as first conversion

// Set sampling time for channel 0 (e.g., 84 cycles)
ADC1->SMPR2 &= ~ADC_SMPR2_SMP0;  // Clear bits
ADC1->SMPR2 |= (4 << ADC_SMPR2_SMP0_Pos);  // 84 cycles

// Step 5: Enable ADC
ADC1->CR2 |= ADC_CR2_ADON;  // Turn on ADC
```

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.

```
// Function to perform single ADC conversion
uint16_t ADC_ReadChannel(void) {
    // Start conversion
    ADC1->CR2 |= ADC_CR2_SWSTART;  // Software trigger to start conversion

    // Wait for conversion to complete
    while (!(ADC1->SR & ADC_SR_EOC)) { }

    // Read and return result
    return ADC1->DR;
}

// Function to convert ADC value to voltage
float ADC_ConvertToVoltage(uint16_t adcValue) {
    // Assuming VREF = 3.3V and 12-bit resolution (4096 levels)
    float voltage = (float)adcValue * 3.3f / 4095.0f;
    return voltage;
}
```

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.

```
// Configure ADC for multi-channel scanning (channels 0, 1, and 4)

// Enable scan mode
ADC1->CR1 |= ADC_CR1_SCAN;

// Set number of conversions in sequence (3)
ADC1->SQR1 &= ~ADC_SQR1_L;
ADC1->SQR1 |= (2 << ADC_SQR1_L_Pos);  // L = 2 means 3 conversions

// Set channel sequence
ADC1->SQR3 = 0;  // Clear all
ADC1->SQR3 |= (0 << ADC_SQR3_SQ1_Pos);  // CH0 as 1st conversion
ADC1->SQR3 |= (1 << ADC_SQR3_SQ2_Pos);  // CH1 as 2nd conversion
ADC1->SQR3 |= (4 << ADC_SQR3_SQ3_Pos);  // CH4 as 3rd conversion

// Set sampling times for each channel
ADC1->SMPR2 &= ~(ADC_SMPR2_SMP0 | ADC_SMPR2_SMP1 | ADC_SMPR2_SMP4);
ADC1->SMPR2 |= (4 << ADC_SMPR2_SMP0_Pos);  // 84 cycles for CH0
ADC1->SMPR2 |= (4 << ADC_SMPR2_SMP1_Pos);  // 84 cycles for CH1
ADC1->SMPR2 |= (4 << ADC_SMPR2_SMP4_Pos);  // 84 cycles for CH4
```

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.

```
// Configure ADC with DMA for continuous multi-channel sampling

// Configure DMA
RCC->AHB1ENR |= RCC_AHB1ENR_DMA2EN;  // Enable DMA2 clock

// Configure DMA2 Stream0 for ADC1
DMA2_Stream0->CR &= ~DMA_SxCR_EN;  // Disable DMA stream
while (DMA2_Stream0->CR & DMA_SxCR_EN) { }  // Wait until disabled

DMA2_Stream0->CR = 0;
DMA2_Stream0->CR |= (0 << DMA_SxCR_CHSEL_Pos);  // Channel 0
DMA2_Stream0->CR |= DMA_SxCR_PL_1;  // Priority high
DMA2_Stream0->CR |= DMA_SxCR_MSIZE_0;  // Memory data size: 16-bit
DMA2_Stream0->CR |= DMA_SxCR_PSIZE_0;  // Peripheral data size: 16-bit
DMA2_Stream0->CR |= DMA_SxCR_MINC;  // Memory increment mode
DMA2_Stream0->CR &= ~DMA_SxCR_PINC;  // Peripheral fixed
DMA2_Stream0->CR |= DMA_SxCR_CIRC;  // Circular mode

// Set addresses
DMA2_Stream0->PAR = (uint32_t)&ADC1->DR;  // Source: ADC1 data register
DMA2_Stream0->M0AR = (uint32_t)adc_values;  // Destination: buffer
DMA2_Stream0->NDTR = 3;  // Number of data items (3 channels)

// Enable DMA stream
DMA2_Stream0->CR |= DMA_SxCR_EN;

// Configure ADC for DMA
ADC1->CR2 |= ADC_CR2_DMA;  // Enable DMA mode
ADC1->CR2 |= ADC_CR2_CONT;  // Continuous conversion mode

// Start ADC conversion
ADC1->CR2 |= ADC_CR2_SWSTART;
```

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

```
// Configure PA5 as analog input
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;  // Enable GPIOA clock
RCC->APB2ENR |= RCC_PB2ENR_ADC1EN;   // Enable ADC1 clock
GPIOA->MODER |= GPIO_MODER_MODER5;    // Set PA5 to analog mode (0b11)

// Configure ADC1
ADC1->CR1 &= ~ADC_CR1_RES;     // 12-bit resolution (default)
ADC1->CR2 &= ~ADC_CR2_CONT;    // Single conversion mode
ADC1->CR2 &= ~ADC_CR2_ALIGN;   // Right alignment

// Configure for channel 5 (PA5)
ADC1->SQR1 &= ~ADC_SQR1_L;      // 1 conversion in sequence
ADC1->SQR3 &= ~ADC_SQR3_SQ1;    // Clear channel selection
ADC1->SQR3 |= (5 << ADC_SQR3_SQ1_Pos);  // Channel 5 as first conversion

// Set sampling time (84 cycles)
ADC1->SMPR2 &= ~ADC_SMPR2_SMP5;
ADC1->SMPR2 |= (4 << ADC_SMPR2_SMP5_Pos);

// Enable ADC
ADC1->CR2 |= ADC_CR2_ADON;

// Function to read ADC and convert to voltage
float ReadVoltage(void) {
    // Start conversion
    ADC1->CR2 |= ADC_CR2_SWSTART;

    // Wait for conversion to complete
    while (!(ADC1->SR & ADC_SR_EOC)) { }

    // Read ADC value
    uint16_t adcValue = ADC1->DR;

    // Convert to voltage (0-3.3V)
    float voltage = (float)adcValue * 3.3f / 4095.0f;

    return voltage;
}
```

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)

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

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 16µs 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

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.

```
// Configure external SRAM (16-bit) on FMC bank 1

// Step 1: Configure GPIO pins for FMC
// Enable GPIO clocks
RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN | RCC_AHB1ENR_GPIOEEN |
                RCC_AHB1ENR_GPIOFEN | RCC_AHB1ENR_GPIOGEN;

// Configure GPIO pins (example for some pins)
// Set alternate function mode (0x2)
GPIOD->MODER |= 0x55555555;  // All pins to alternate function
GPIOE->MODER |= 0x55555555;
// Set to AF12 (FMC)
GPIOD->AFR[0] = 0xCCCCCCCC;
GPIOD->AFR[1] = 0xCCCCCCCC;
GPIOE->AFR[0] = 0xCCCCCCCC;
GPIOE->AFR[1] = 0xCCCCCCCC;

// Step 2: Enable FMC clock
RCC->AHB3ENR |= RCC_AHB3ENR_FMCEN;

// Step 3: Configure FMC bank 1 for SRAM
// Set timing for SRAM (example values)
FMC_Bank1->BTCR[0] =
    FMC_BCR1_MBKEN |    // Memory bank enable
    FMC_BCR1_MTYP_0 |   // Memory type SRAM
    FMC_BCR1_MWID_0 |   // 16-bit data bus
    FMC_BCR1_WREN;      // Write enable

// Set timing (ADDSET=1, DATAST=2)
FMC_Bank1->BTCR[1] =
    (1 << FMC_BTR1_ADDSET_Pos) |
    (2 << FMC_BTR1_DATAST_Pos);
```

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.