Computertechnik 2

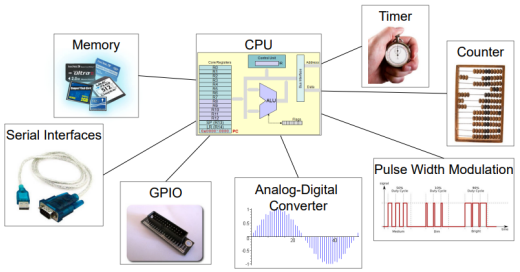
Jil Zerndt
FS 2025

Microcontroller Basics

Microcontroller Architecture

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

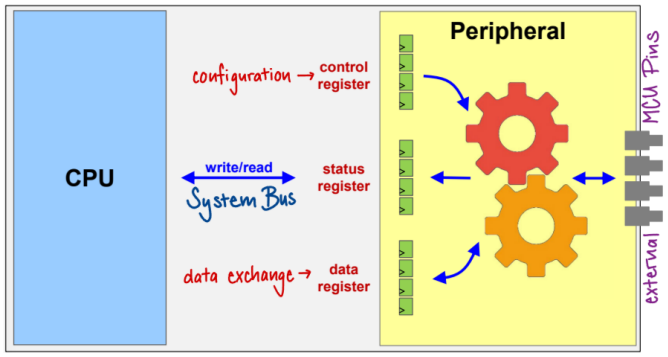
Single Chip Solution
⇒ CPU with integrated memory and peripherals



Peripherals and Registers

Peripherals

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



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

CPU read/write to peripheral registers
How does the CPU write to and read from peripheral registers?
- CPU reads/writes to peripheral registers
- CPU uses memory-mapped I/O to access peripheral registers
- CPU uses load/store instructions to access peripheral registers

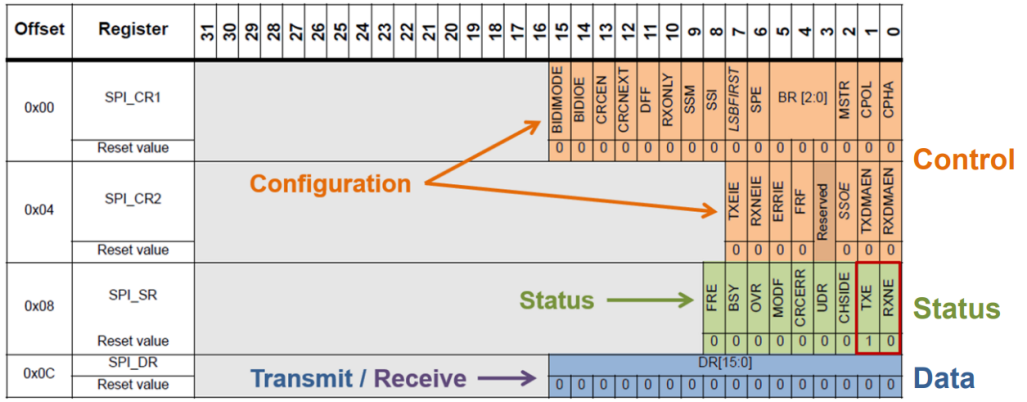
⇒ System Buses

Control and Status Registers
Peripherals interact with the CPU through registers:
- **Control Registers**: CPU configures peripherals
  - CPU writes to register bit
  - Slave hardware uses the output of this bit
  - Usually read/write
- **Status Registers**: CPU monitors peripheral state
  - Slave writes status into register bit
  - CPU reads register bit
  - Usually read-only

Both control & status bits can be in same register.
- **Data Registers**
  enable CPU to exchange data with the peripheral

CPU access to individual Peripheral Registers
- ARM & STM map the peripheral registers into the memory address range
- Reference Manual shows the defined addresses

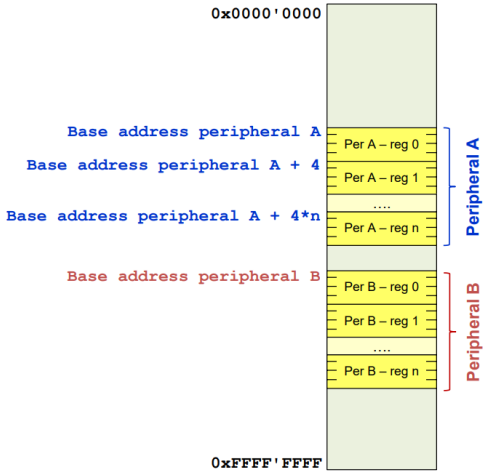
Example SPI (Serial Peripheral Interface)



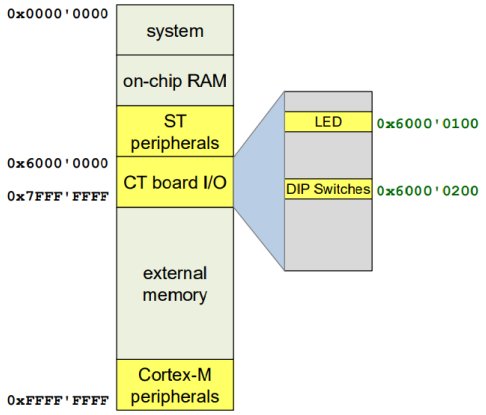
Each column shows a single flip-flop

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



Memory-Mapped Peripheral Registers
- Control register: controls states of LEDs
- Status register: monitors states of DIP switches

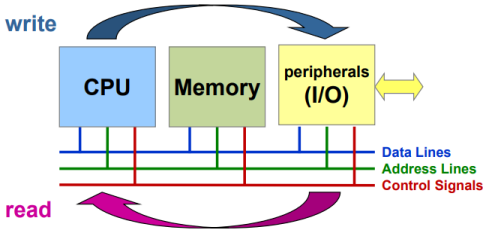


Memory-Mapped Registers
Registers are mapped into the memory address range - each has a specific address:

```
// Define memory-mapped register addresses
#define ADDR_LED_31_0       0x60000100
#define ADDR_DIP_SWITCH_31_0 0x60000200

// Read from DIP switches and write to LEDs
uint32_t value = read_word(ADDR_DIP_SWITCH_31_0);
write_word(ADDR_LED_31_0, value);
```

System Bus

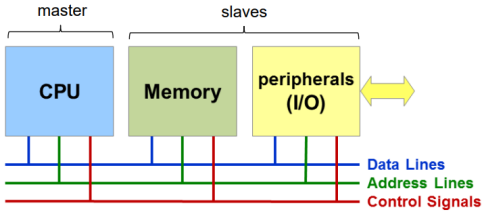
- Interconnects CPU with memory and peripherals, allowing data transfer between components.
- CPU acts as master: initiating and controlling all transfers
- Peripherals and memory act as slaves: responding to requests from the CPU
- System bus is a shared resource



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

Signal Groups

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



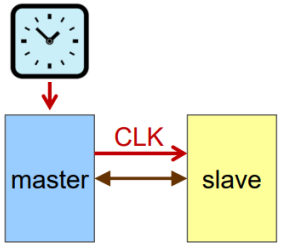
Bus Specification

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

Bus Timing Options

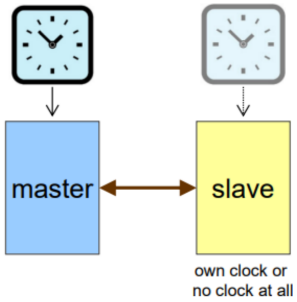
Synchronous

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



Asynchronous

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



But how can a driver be disconnected electrically?

Multiple devices driving the same data line

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

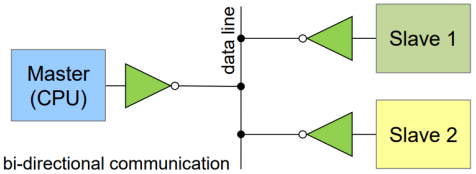
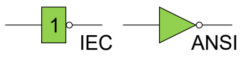
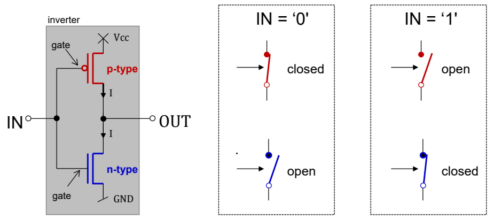


Figure only shows output paths, input paths are not shown.

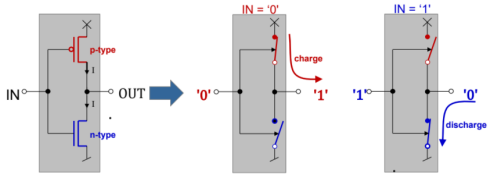
CMOS Inverter



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

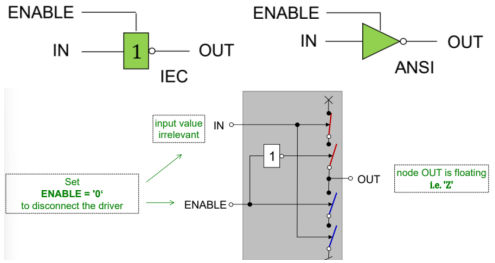


e.g. Vcc = 3V = '1', Gnd = 0V = '0'
Vcc is the supply voltage of the circuit/chip

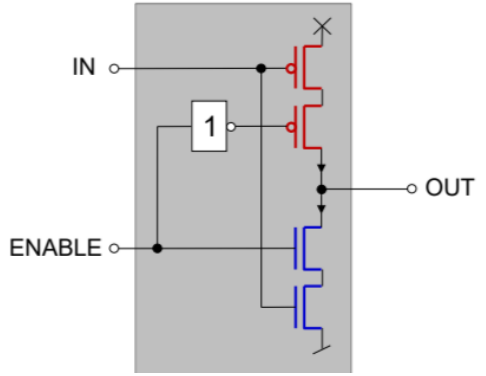


A buffer is built by connecting two inverters in series

CMOS Tri-State Inverter



Implementation:



Tri-State Logic

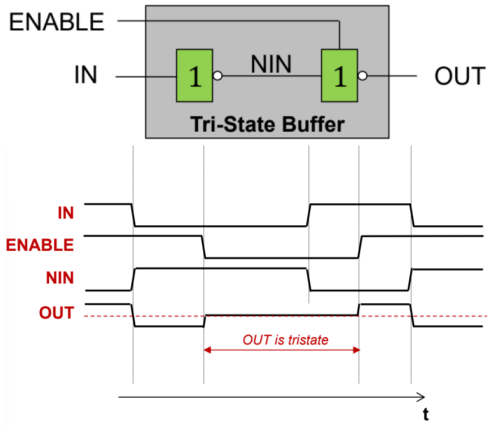
Multiple devices can drive the same data line thanks to tri-state capability:

- 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

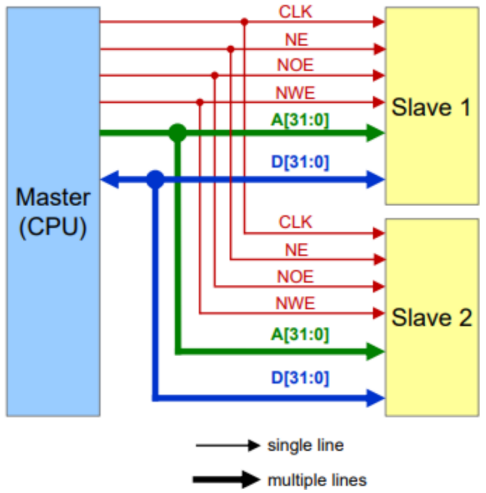
Synchronous Bus Timing

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)

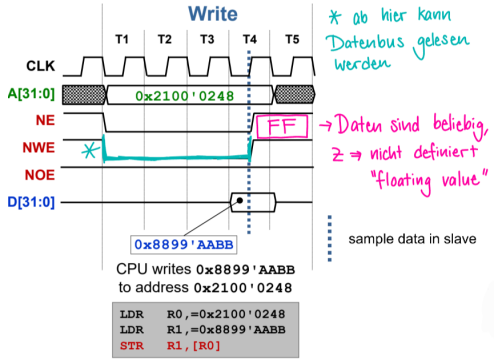
Note that 'N' prefix indicates active-low signals.

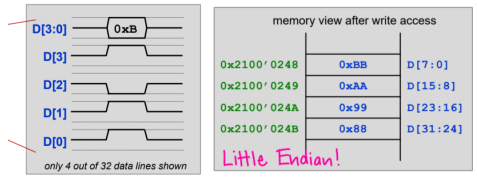
Block Diagram



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)

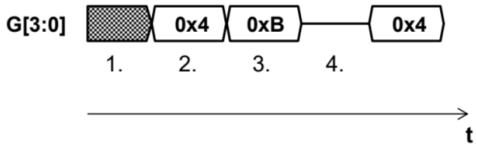




WICHTIG: nicht genau mit Flanke schreiben, Daten brauchen eine gewisse Zeit um stabil zu werden (keine genaue Definition, muss einfach 'genug' sein)
READ dauert länger als WRITE, da die Daten erst stabil werden müssen bevor sie gelesen werden können

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'

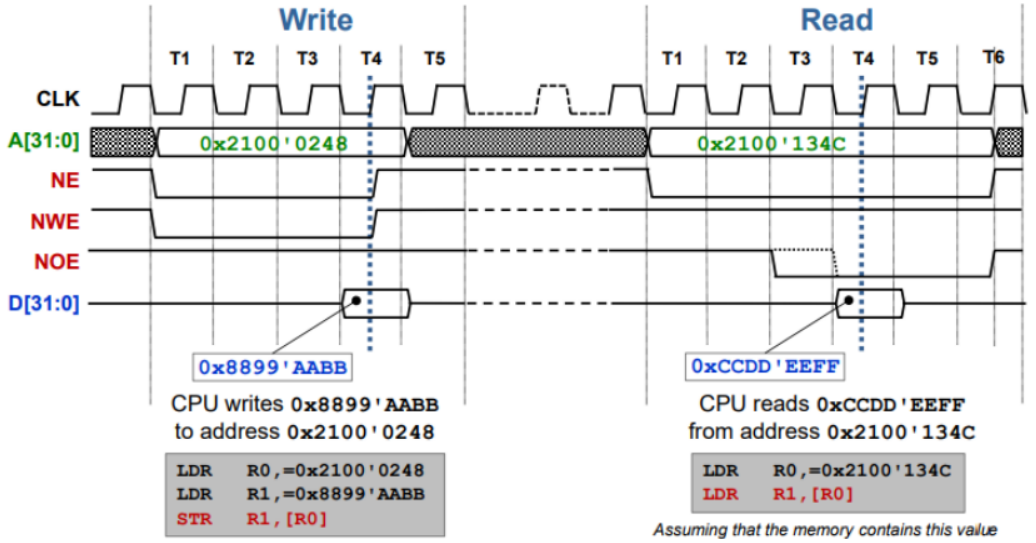
3. The bus holds the value $0 \times B$

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"

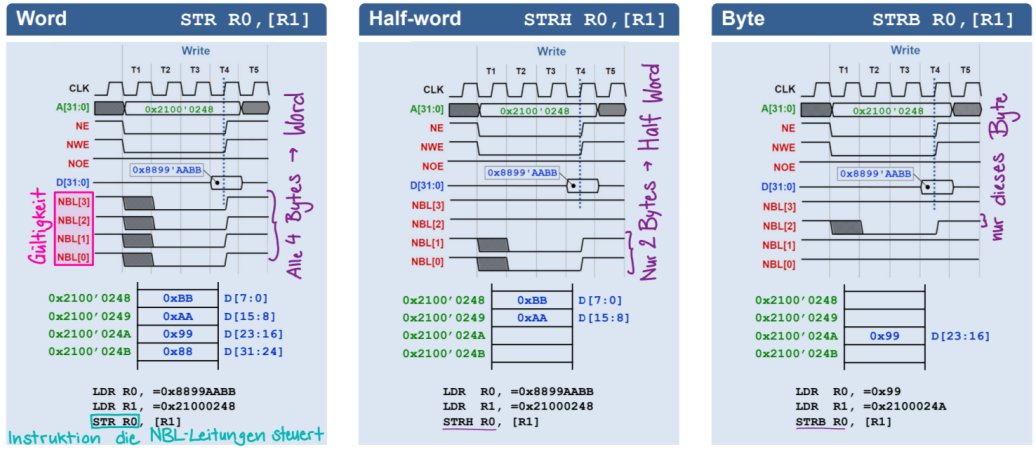
Timing Diagram

- `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

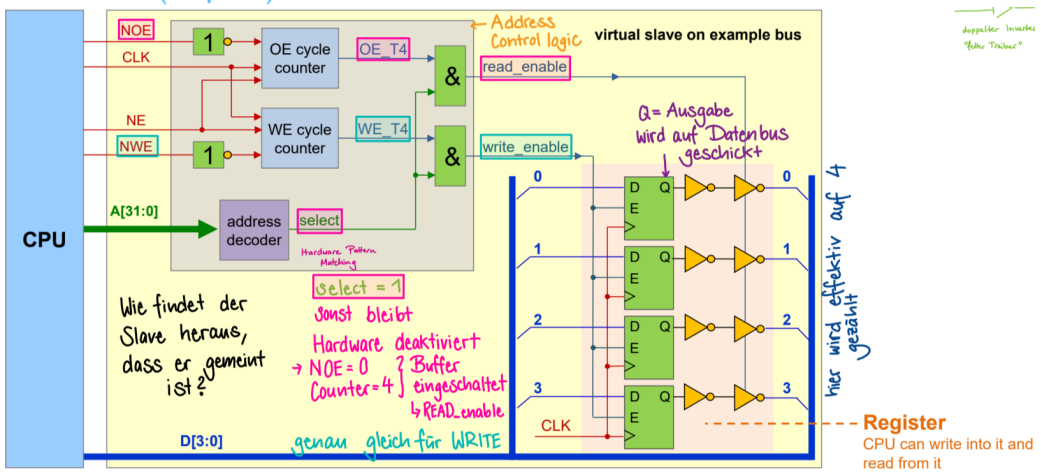


Gültigkeit: damit zeigt die CPU an, welche der 4 Bytes übertragen werden sollen (gültig = 0 (unten))

- 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

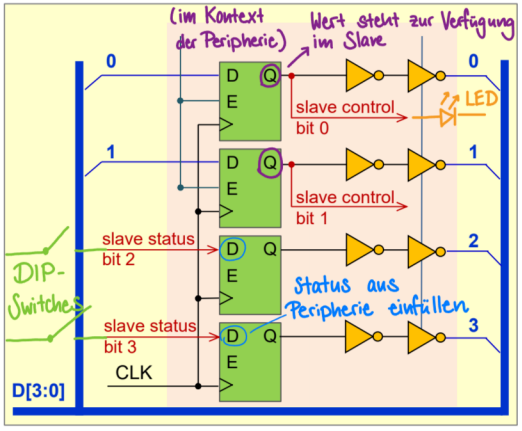
Example STM32 Power Control/Status Register PWR_CSR

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

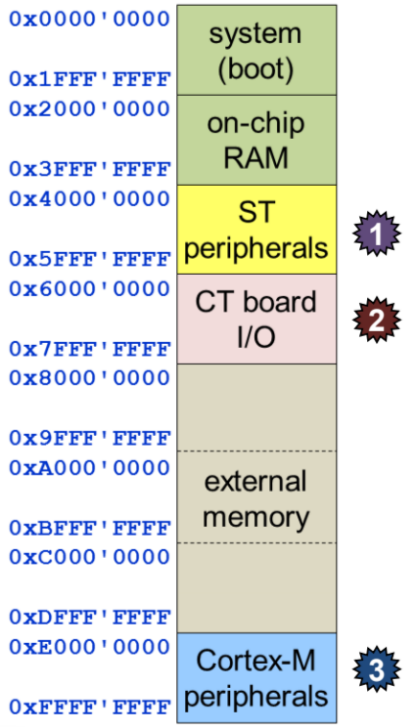
Control/Status register on example bus:



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

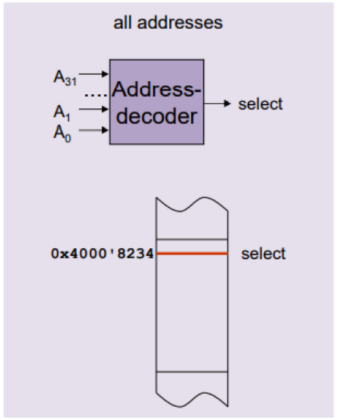
Full Address Decoding

- 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

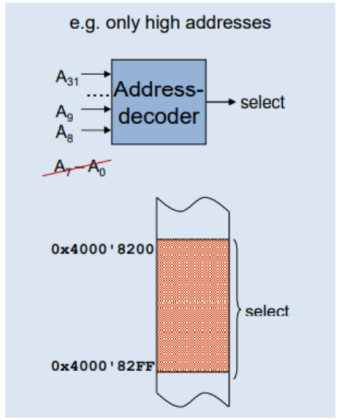
Partial Address Decoding

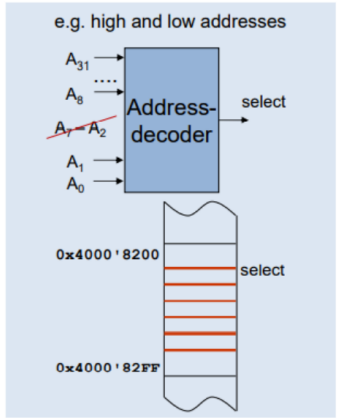
- Only a subset of address lines are decoded
- A control register can be accessed at multiple locations
- 1:n mapping: Multiple addresses map to the same hardware register
- Motivation: Simpler and possible Aliasing (Map a hardware register to several addresses)

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:

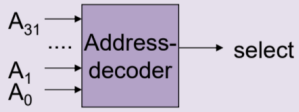
- Each decoded address represents a range of 2^m addresses
- The size of this address range is 2^m bytes
- The start address has all lower bits set to 0
- The end address has all lower bits set to 1

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

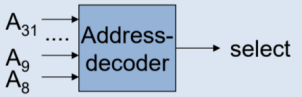
- all addresses from A31 to A0



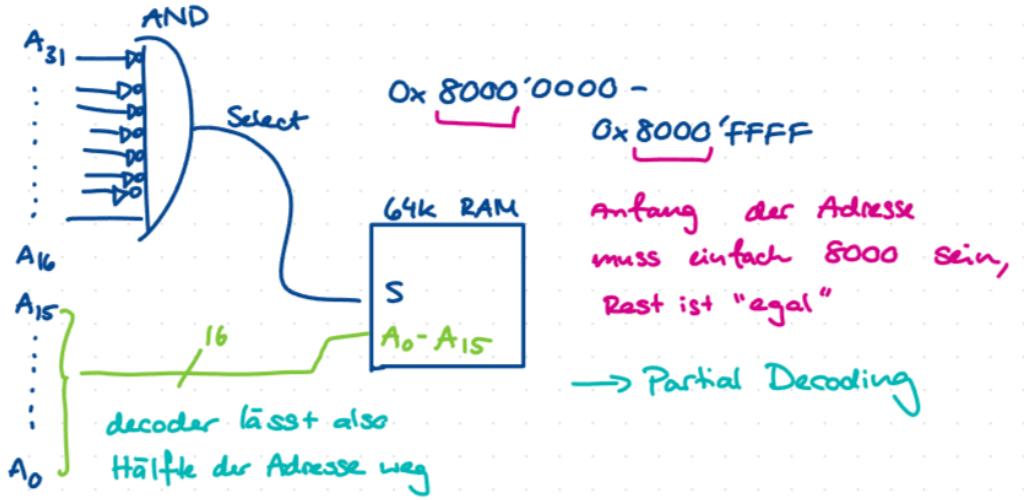
- select is active for exactly one address
- E.g. at `0x4000'8234`

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

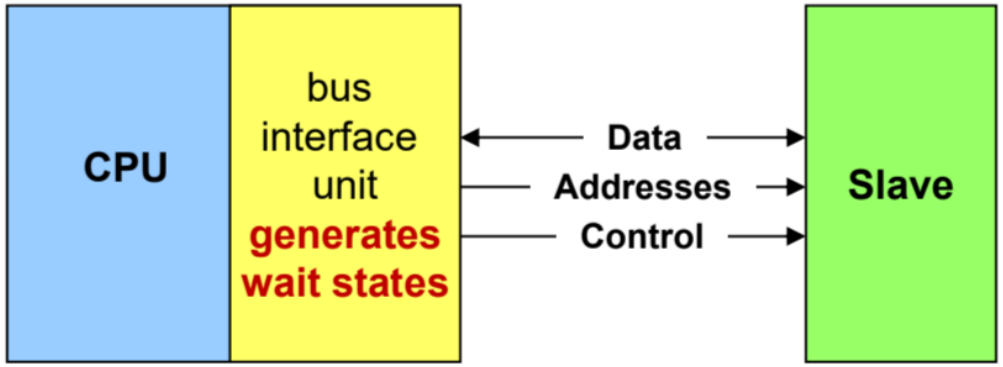
Problem: Individual Slave Access times

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

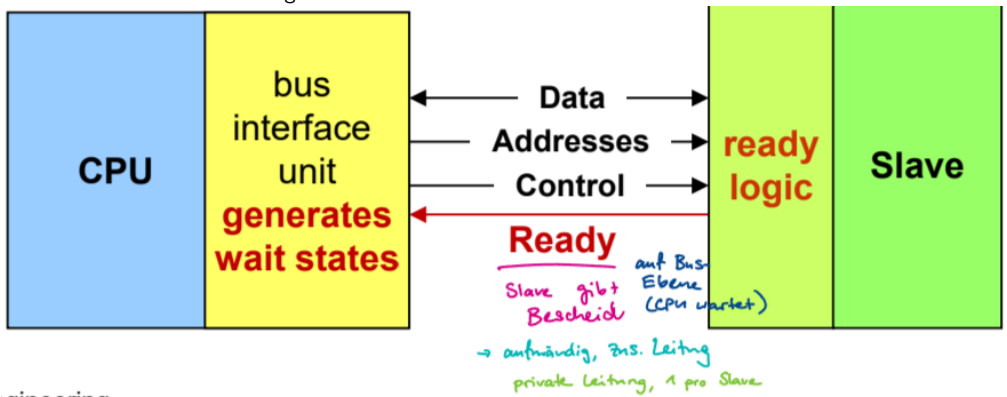
Solutions for slow slaves

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)

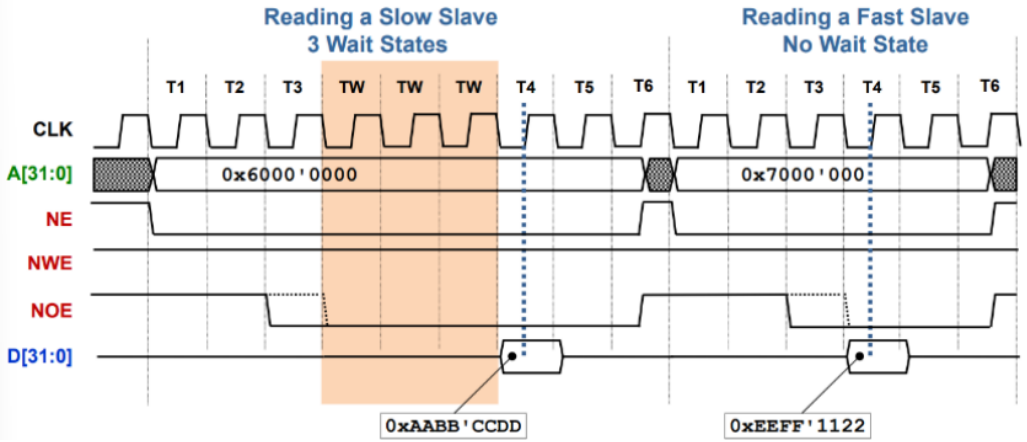


- **Bus Mastering** Slave tells bus interface unit when it is ready
  Allow a peripheral to take control of the bus and perform its own accesses (e.g. DMA)
  Well suited for slaves with long or variable access times



Individual Wait States

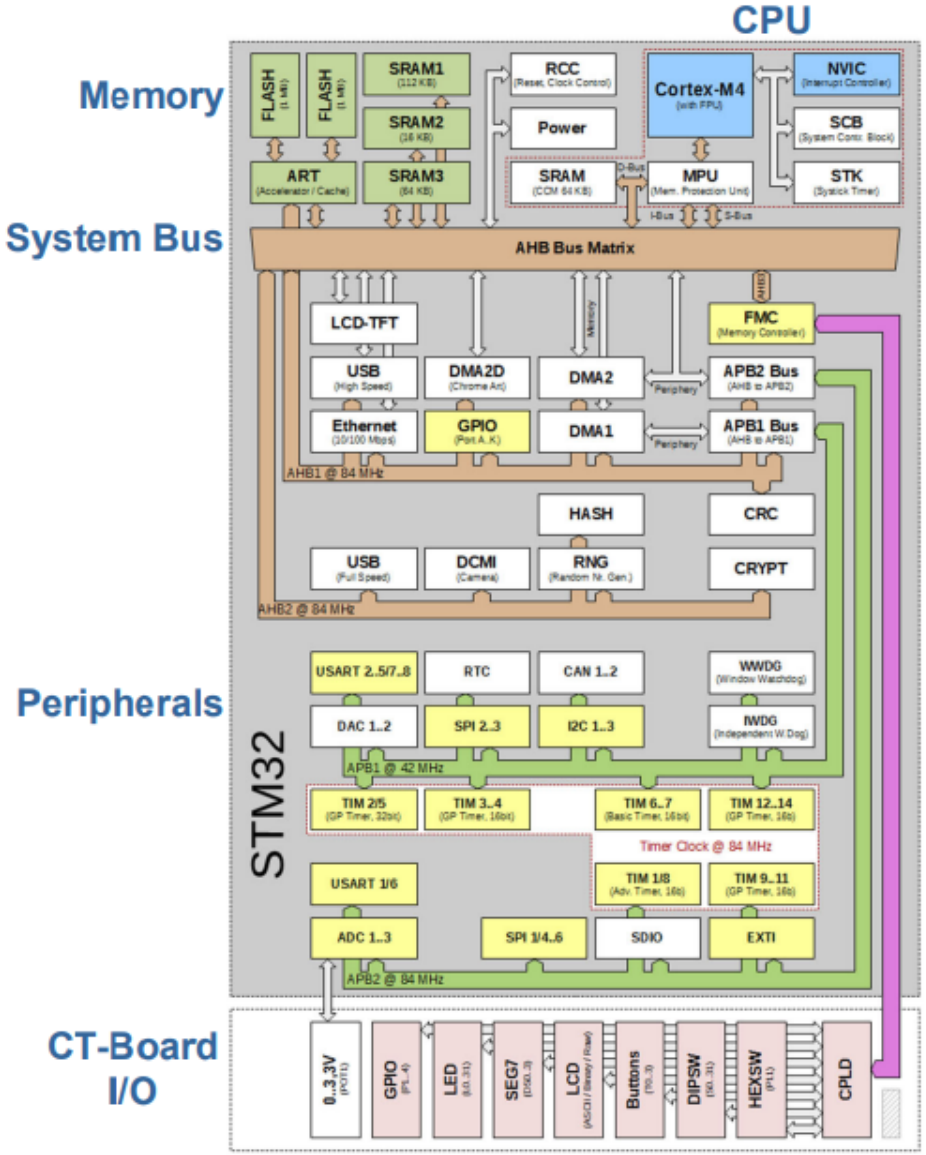
Wait states are inserted to slow down the CPU to match the speed of the slowest peripheral (depending on the address of an access)



STM32 Microcontroller

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

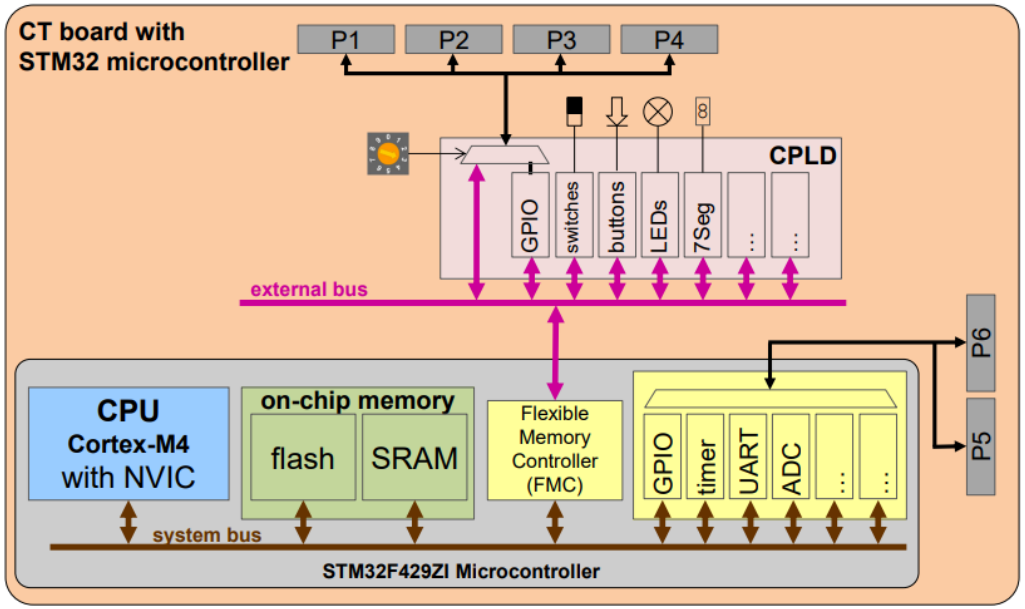
- On-chip system bus: 32 data lines, 32 address lines and control signals
- Off-chip external bus: 16 data lines, 24 address lines and control signals



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

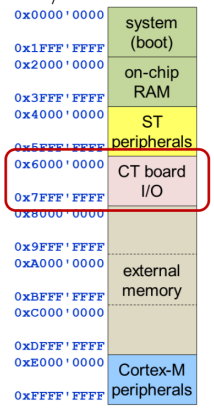
e.g. writing to and reading from CT Board I/O

```
// 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

Microcontroller Basics Exercises

Bus Access Analysis

Analyzing Bus Cycle Diagrams

Read access analysis

- Examine clock (CLK), address lines (A), data lines (D), and control signals (NWE, NOE, NE)
- Identify whether it's a read (NOE active) or write (NWE active) operation
- Note the address value from address lines
- For read operations, note the data value returned on data lines

Write access analysis

- Identify write operations (NWE active)
- Note the address value from address lines
- Note the data value being written from data lines

Memory mapping

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

Bus Cycle Analysis Example

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

Access 1: T1 T2 T3 T4 T5 T6 with A[31:0] = 0x6100ABC8, NOE active, D[31:0] = 0x11223344
Access 2: T1 T2 T3 T4 T5 with A[31:0] = 0x6100ABC8, NWE active, D[31:0] = 0x55667788

Solution:

Access 1: This is a read operation because NOE is active.
- Address: 0x6100ABC8
- Data read: 0x11223344

Access 2: This is a write operation because NWE is active.
- Address: 0x6100ABC8
- Data written: 0x55667788

Memory contents (little-endian):

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

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

Address Decoding

Analyzing Address Decoding Logic

Identifying addressable range

- Determine the total address space from the number of address lines: 2^n bytes
- For partial address decoding, identify which address lines are actually decoded
- Find which address bits are fixed (compared in the logic) and which are "don't care"

Calculating address ranges

- For fully decoded address: Only one specific address activates the select signal
- For partially decoded address:
  - Identify how many address bits are not decoded ("don't care"bits)
  - Calculate the number of different addresses that map to the same location: 2^m where m is the number of "don't care"bits
  - Use the fixed bits to determine the base address pattern
  - Substitute all possible values for the "don't care"bits to find all addresses

Creating decoding logic

- For exact address decoding: AND all address bits (or their complements) according to the required pattern
- For partial decoding: Only include the relevant address bits in the logic

Address Decoding Example

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

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

Solution:

The decoding logic fixes 4 address bits: A[5:2] = 1100
The bits A[1:0] are "don't care"bits (not included in the logic), giving 4 possible addresses:

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

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

Memory-Mapped I/O Access in C

Memory-Mapped Register Access in C

Define register addresses

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

Reading from registers

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

Writing to registers

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

Waiting for status bits

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

Memory-Mapped Register Access Example

Write C code to:

1. Read an 8-bit control register at address 0x61000007
2. Set all bits of a 16-bit control register at address 0x61000008 to '1'
3. Wait until bit 15 in a 32-bit register at address 0x6100000C is set
4. Set bit 16 in a 32-bit register at address 0x61000010 without changing other bits

Solution:

```c
#include <stdint.h>

// Define register addresses with appropriate types
#define MY_BYTE_REG     (*((volatile uint8_t *)(0x61000007)))
#define MY_HALFWORD_REG (*((volatile uint16_t *)(0x61000008)))
#define MY_WORD_REG     (*((volatile uint32_t *)(0x6100000C)))
#define MY_WORD_REG2    (*((volatile uint32_t *)(0x61000010)))

void register_operations(void) {
    // Read 8-bit control register
    uint8_t my_var = MY_BYTE_REG;

    // Set all bits of 16-bit register to '1'
    MY_HALFWORD_REG = 0xFFFF;

    // Wait until bit 15 is set in 32-bit register
    while (!(MY_WORD_REG & 0x00008000)) {
        // Empty loop body
    }

    // Set bit 16 without changing other bits
    MY_WORD_REG2 |= 0x00010000;
}
```

GPIO (General Purpose I/O)

GPIO Fundamentals

GPIO

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

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

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

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

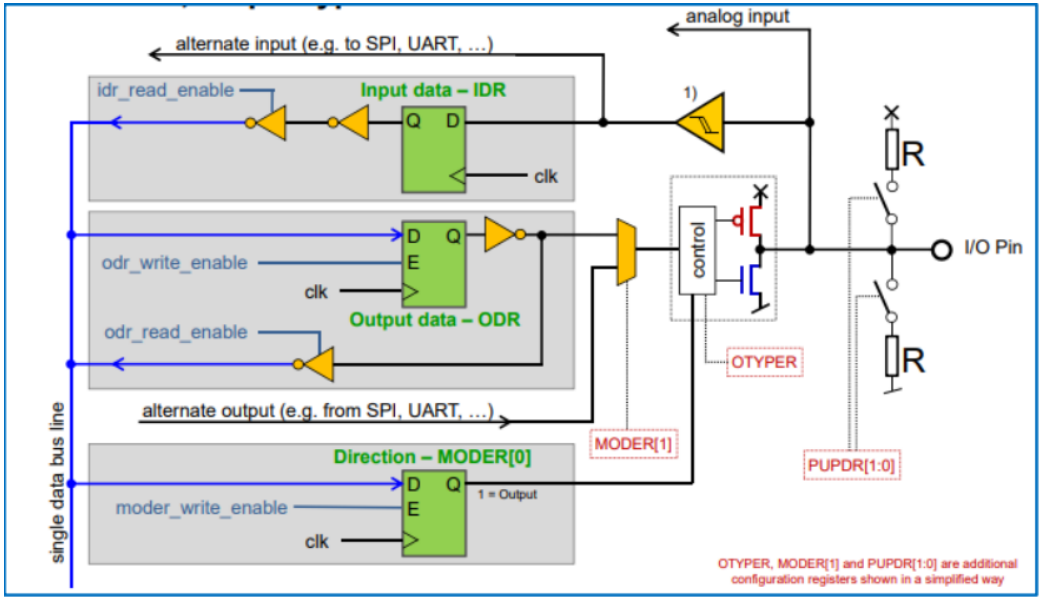
Not all functions can be used simultaneously; configuration registers define pin usage.

GPIO Structure

GPIO Registers

Each GPIO port has several configuration registers:
- **GPIOx_MODER**: Mode register (input, output, alternate function, analog) - configures pin as input or output (direction control)
- **GPIOx_OTYPER**: Output type register (push-pull or open-drain)
- **GPIOx_OSPEEDR**: Output speed register (low, medium, high, very high) - configures speed
- **GPIOx_PUPDR**: Pull-up/pull-down register
- **GPIOx_IDR**: Input data register (read-only) - reads the pin state
- **GPIOx_ODR**: Output data register (read/write) - sets the output state
- **GPIOx_BSRR**: Bit set/reset register (atomic operations)
- **GPIOx_LCKR**: Configuration lock register
- **GPIOx_AFRL/H**: Alternate function registers (AF selection for pins 0-7 and 8-15)

Structure



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

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

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.

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)

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
```

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)
```

Data Registers and Operations

Register Access

Register Address = Base address + Offset

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

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)

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));
```

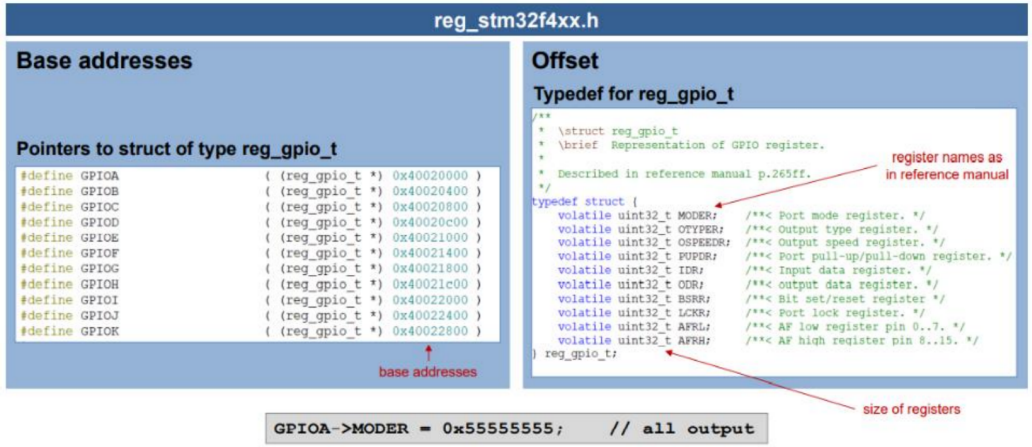
Hardware Abstraction Layer (HAL)

```
#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




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

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);
```

General Purpose I/O (GPIO) Exercises

GPIO Configuration and Programming

Configuring GPIO Pins

Identify GPIO port and pin

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

Calculate register addresses

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

Configure GPIO mode

- Use GPIOx_MODER register to set pin mode:
  - 00: Input mode
  - 01: General purpose output mode
  - 10: Alternate function mode
  - 11: Analog mode
- Each pin requires 2 bits, so pin n uses bits 2n+1:2n

Configure output type (for output mode)

- Use GPIOx_OTYPER register:
  - 0: Push-pull output (can drive high or low)
  - 1: Open-drain output (can drive low only)

Configure pull-up/pull-down resistors

- Use GPIOx_PUPDR register:
  - 00: No pull-up, no pull-down
  - 01: Pull-up
  - 10: Pull-down
  - 11: Reserved
- Each pin requires 2 bits, so pin n uses bits 2n+1:2n

Configure speed (for output mode)

- Use GPIOx_OSPEEDR register:
  - 00: Low speed
  - 01: Medium speed
  - 10: High speed
  - 11: Very high speed
- Each pin requires 2 bits, so pin n uses bits 2n+1:2n

Enable clock for GPIO port

- Use RCC_AHB1ENR register to enable clock for the GPIO port
- Set the corresponding bit for the port

GPIO Configuration Example

Configure pin PA3 as a low-speed, open-drain output with pull-up.

Solution:

First, identify the registers and their addresses:

- GPIOA base address: 0x4002 0000
- GPIOA_MODER: 0x4002 0000 (offset 0x00)
- GPIOA_OTYPER: 0x4002 0004 (offset 0x04)
- GPIOA_OSPEEDR: 0x4002 0008 (offset 0x08)
- GPIOA_PUPDR: 0x4002 000C (offset 0x0C)

Then, calculate the bit fields for pin 3:

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

C code to configure the pin:

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

// Configure PA3 as output (01)
// Clear bits 7:6 then set bit 6
GPIOA->MODER &= ~(0x3 << 6);
GPIOA->MODER |= (0x1 << 6);

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

// Configure as low speed (00)
GPIOA->OSPEEDR &= ~(0x3 << 6);

// Configure with pull-up (01)
GPIOA->PUPDR &= ~(0x3 << 6);
GPIOA->PUPDR |= (0x1 << 6);
```

Using GPIO Data Registers

Reading input pins

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

Writing output pins

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

Atomic bit manipulation

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

GPIO Data Operations Example

Write code to:

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

Solution:

First, define the register addresses and pin mappings:

```c
// Button B1 is on PA0
// Green LED LD3 is on PG13
// Red LED LD4 is on PG14

#include "reg_stm32f4xx.h"

void button_led_control(void) {
    // Read button state (PA0)
    uint32_t button_state = GPIOA->IDR & (1 << 0);

    if (button_state) {
        // Button pressed - turn on green LED (PG13) and turn off red LED (PG14)
        // Set bit 13, clear bit 14 atomically
        GPIOG->BSRR = (1 << 13) | (1 << (16+14));
    } else {
        // Button not pressed - turn on red LED (PG14) and turn off green LED (PG13)
        // Set bit 14, clear bit 13 atomically
        GPIOG->BSRR = (1 << 14) | (1 << (16+13));
    }
}
```

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

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

Serial Data Transfer - 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.

```c
// 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

Serial Data Transfer

Overview

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

SPI - Serial Peripheral Interface

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

UART - Universal Asynchronous Receiver Transmitter (Serial Interface)

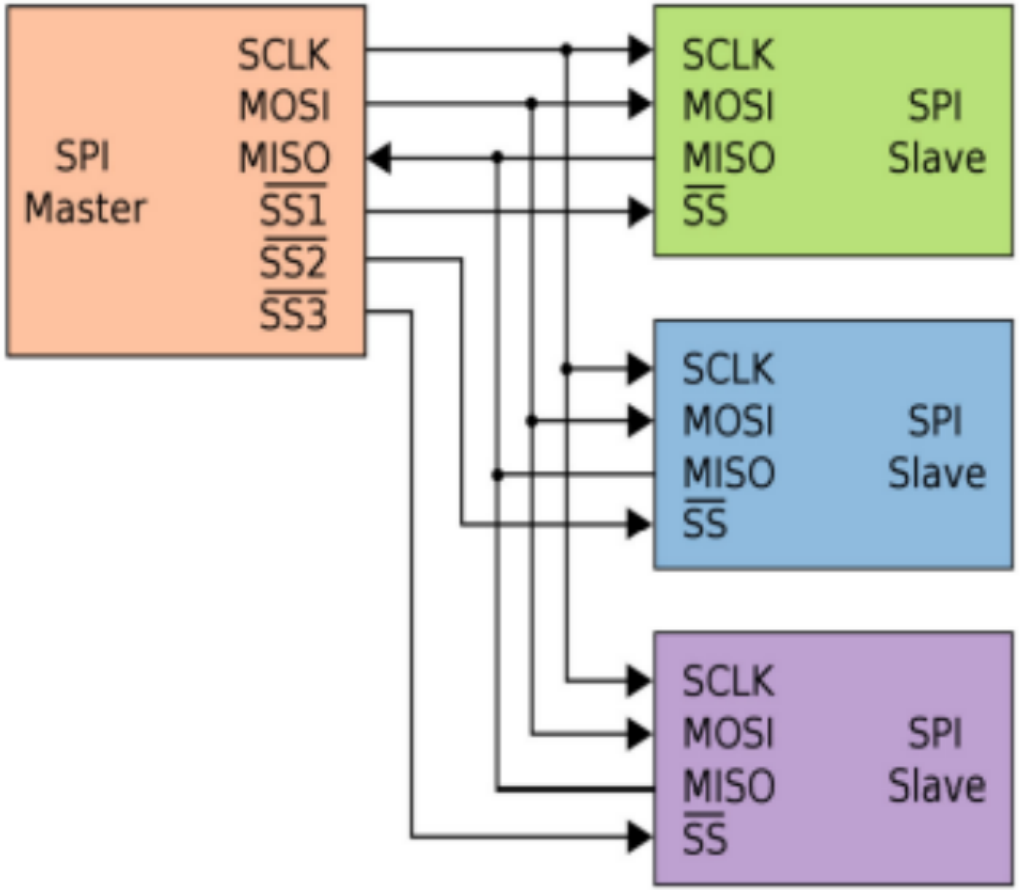
- Transmitter and receiver use diverging clocks
- synchronization using start and stop bits → overhead
- longer connections require line drivers → RS-232/RS-485

I2C - Inter-Integrated Circuit

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

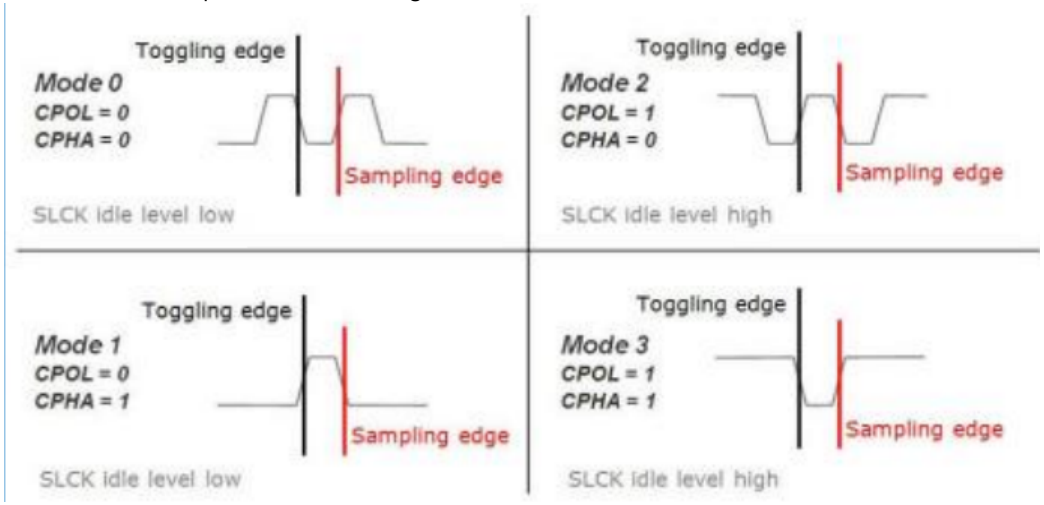
Single Master - Multiple Slaves

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



Clock Polarity (CPOL) Clock Phase (CPHA)

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



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

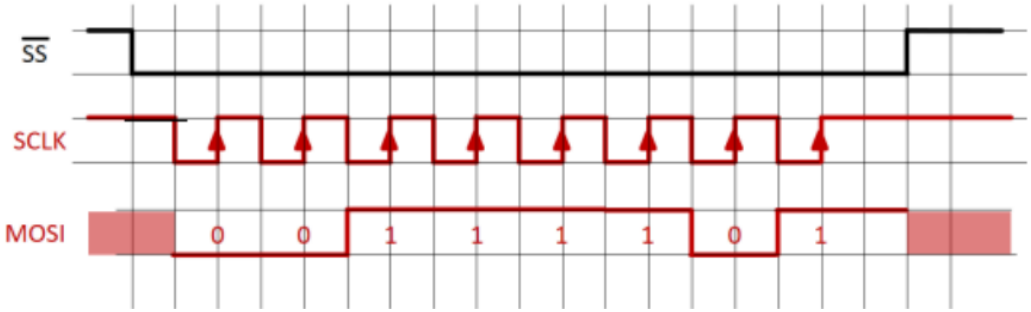
Serial Connection - SPI

Properties of SPI

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

SPI - Serial Peripheral Interface

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



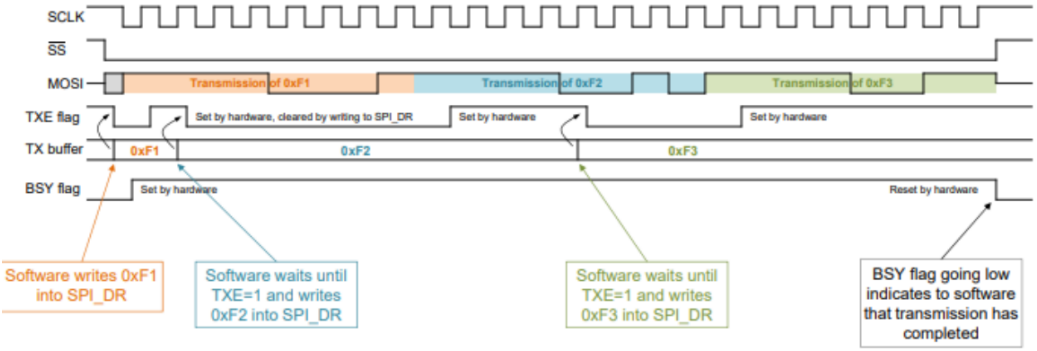
Synchronizing Hardware and Software

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

Transmitting in Software

Example with CPOL=1, CPHA=1, MSB first

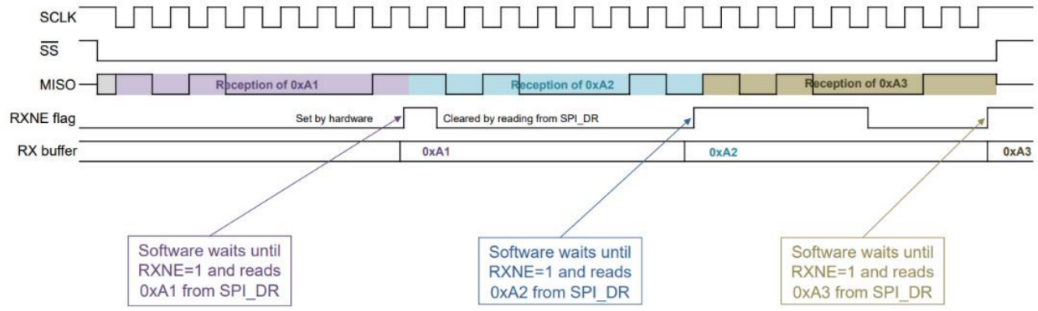
- Example: SW wants to transmit bytes 0xF1, 0xF2, 0xF3



Receiving in Software

Example with CPOL=1, CPHA=1, MSB first

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

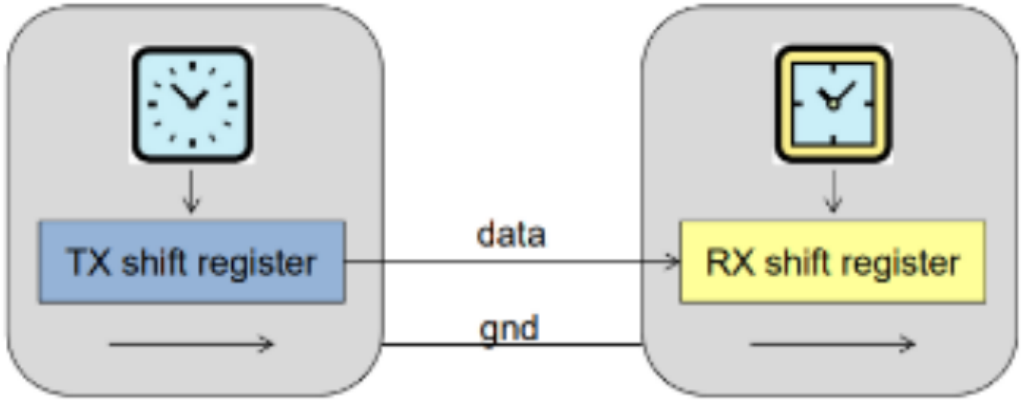


UART/I2C

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

Serial Data Transfer Exercises

UART Communication

Analyzing UART Transmission

Determine bit timing

- Calculate bit period (T) from baud rate: T = 1/baud_rate
- Example: 19200 baud = 1/19200 s = 52.1 µs per bit

Identify frame structure

- Start bit: Always '0'
- Data bits: LSB first, typically 7 or 8 bits
- Parity bit: Optional, can be even, odd, mark ('1'), or space ('0')
- Stop bit(s): Always '1', can be 1, 1.5, or 2 bits
- Idle state: Line remains high ('1')

Calculate total frame time

- Add up all bits: Start bit + Data bits + Parity bit + Stop bits
- Multiply by bit period (T)
- Example: 8N1 (8 data bits, no parity, 1 stop bit) = 10 bits total

Calculate maximum throughput

- Divide baud rate by total number of bits per frame
- Example: 9600 baud, 8N1 format = 9600/10 = 960 bytes/second

Evaluate clock tolerance

- Maximum tolerable deviation is typically $\pm(0.5/n)$ where n is the number of bits from start bit to the last data bit
- Example: For 8 data bits, max deviation = $\pm(0.5/8.5) \approx \pm 5.9\%$

UART Transmission Analysis

For a UART transmission with 19,200 bits/s, 7 data bits, and one stop bit (no parity), draw the timing diagram for transmitting the ASCII characters ÄC".

ASCII values:

- 'A' = 0x41 = 100 0001b
- 'C' = 0x43 = 100 0011b

Solution:

First, calculate the bit period:

- T = 1/19200 s = 52.1 µs

For each character, the frame consists of:

- 1 start bit (S) - always '0'
- 7 data bits (D) - LSB first
- 1 stop bit (E) - always '1'

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

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

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

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

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

The timing diagram:

```
I I S D D D D D D E S D D D D D D D E I I I
1 1 0 1 0 0 0 0 0 1 1 0 1 1 0 0 0 0 1 1 1 1 1
<--------------'A'------------><--------------'C'------------->
```

Each bit has a duration of T = 52.1 µs, so the total transmission time for both characters is:

- 2 characters × 9 bits per character × 52.1 µs = 937.8 µs

The clock tolerance can be calculated as:

- For 7 data bits: max deviation = $\pm(0.5/7.5) \approx \pm 6.67\%$

SPI Communication

Analyzing SPI Communication

Identify SPI mode

- Determine CPOL (Clock Polarity):
  - CPOL=0: Clock idles low
  - CPOL=1: Clock idles high
- Determine CPHA (Clock Phase):
  - CPHA=0: Data sampled on first edge (leading edge)
  - CPHA=1: Data sampled on second edge (trailing edge)
- Identify SPI mode from CPOL and CPHA:
  - Mode 0: CPOL=0, CPHA=0
  - Mode 1: CPOL=0, CPHA=1
  - Mode 2: CPOL=1, CPHA=0
  - Mode 3: CPOL=1, CPHA=1

Draw timing diagram

- First, draw clock (SCLK) based on CPOL
- Draw chip select (SS#) - active low
- Draw MOSI line with data from master to slave
- Draw MISO line with data from slave to master
- Indicate sampling edges based on CPHA:
  - CPHA=0: Sample on rising edge if CPOL=0, falling edge if CPOL=1
  - CPHA=1: Sample on falling edge if CPOL=0, rising edge if CPOL=1

Calculate timing parameters

- Bit cell time = 1/clock_frequency
- Frame time = (number_of_bits)/clock_frequency
- Example: 8 bits at 100 kHz = 80 µs per frame

SPI Timing Diagram Example

Draw the timing diagram for an SPI transfer with:

- Mode 3 (CPOL=1, CPHA=1)
- 8-bit data: MOSI = 0xA7, MISO = 0x37
- MSB first
- Clock frequency = 100 kHz

Mark all sampling edges on the diagram.

Solution:

For Mode 3:

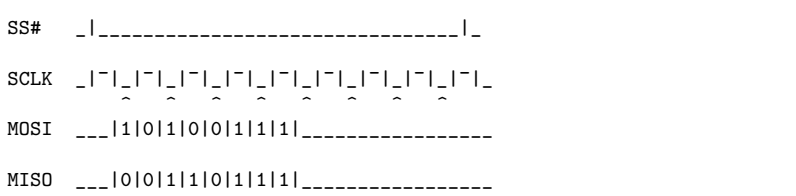
- CPOL=1: Clock idles high
- CPHA=1: Data sampled on second edge (trailing edge)
- Sampling occurs on rising clock edges

Data values in binary:

- MOSI: 0xA7 = 10100111
- MISO: 0x37 = 00110111

Bit cell time = 1/100 kHz = 10 µs

Timing diagram:



The arrows (∧) mark the sampling edges, which are the rising edges in Mode 3.

Total frame time = 8 bits × 10 µs = 80 µs.

I2C Communication

Analyzing I2C Communication

Identify I2C protocol elements

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

Parse I2C frames

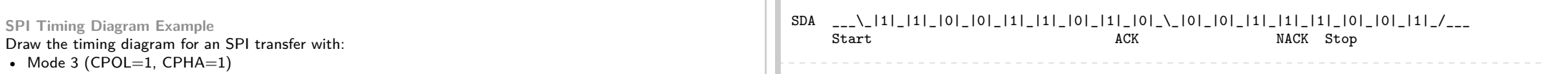
- First byte after start: 7-bit address + R/W bit
  - R/W = 0: Write operation
  - R/W = 1: Read operation
- Check for ACK/NACK after each byte
- Data bytes follow (8 bits each)
- Master or slave can end transmission with STOP condition

Interpret timing diagram

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

I2C Communication Analysis

Analyze the following I2C timing diagram and describe the transaction:



Solution:

The I2C transaction consists of:

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

The complete transaction is:

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

Protocol Comparison and Selection

Selecting the Appropriate Serial Protocol

Understand protocol characteristics

- **UART:**
  - Point-to-point, full-duplex
  - Asynchronous, no shared clock
  - Simple wiring (2-3 wires)
  - Moderate speed (up to 5 Mbps)
- **SPI:**
  - Master-slave, full-duplex
  - Synchronous with shared clock
  - Multiple wires (MOSI, MISO, SCLK, SS)
  - High speed (up to 50+ Mbps)
- **I2C:**
  - Multi-master, multi-slave
  - Synchronous with shared clock
  - Two wires (SCL, SDA)
  - Medium speed (100 kbps, 400 kbps, up to 5 Mbps)

Consider application requirements

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

Evaluate protocol overhead

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

Consider hardware support

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

Serial Protocol Selection

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

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

Solution:

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

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

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

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

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% duty cycle:**
   - 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;
```

Timer/Counter Exercises

Timer Configuration

Configuring Basic Timer Operations

Understand timer components

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

Calculate timer parameters

- Determine desired period (T): The time between events
- Calculate required counts: counts = T × input_frequency
- If counts > 65,536 (16-bit limit), use prescaler:
  - Choose prescaler value (PSC): PSC = ⌊ counts/65,536 ⌋
  - Adjusted counter value: ARR = ⌊ counts/(PSC+1) ⌋ - 1
- For precision, minimize prescaler while keeping ARR within limits

Configure timer registers

- Enable timer clock in RCC registers
- Set prescaler value in TIMx_PSC register
- Set auto-reload value in TIMx_ARR register
- Configure counting direction in TIMx_CR1:
  - DIR=0: Up-counting
  - DIR=1: Down-counting
- Enable interrupt if needed in TIMx_DIER (UIE bit)
- Enable timer by setting CEN bit in TIMx_CR1

Basic Timer Configuration

Configure Timer 2 on an STM32F4 microcontroller to generate an interrupt every 1 second. The timer clock frequency is 84 MHz.

Solution:

First, calculate the required counts:

- counts = 1s × 84MHz = 84,000,000

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

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

C code for configuration:

```c
// Enable TIM2 clock
RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;

// Configure TIM2
TIM2->PSC = 8399;          // Prescaler: 8400-1
TIM2->ARR = 9999;          // Auto-reload: 10000-1
TIM2->CR1 = 0;             // Up-counting mode
TIM2->DIER |= TIM_DIER_UIE; // Enable update interrupt
TIM2->CR1 |= TIM_CR1_CEN;   // Enable timer

// Configure NVIC
NVIC_EnableIRQ(TIM2_IRQn);
```

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

Capture/Compare Operations

Configuring PWM Output

Understand PWM parameters

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

Calculate PWM parameters

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

Configure PWM output

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

PWM Configuration Example

Configure Timer 4 for PWM output with:

- ARR = 0x9C3F (value 39999)
- Up-counting mode, PWM Mode 1
- 25% duty cycle

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

Solution:

For up-counting mode with PWM Mode 1:

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

For down-counting mode with PWM Mode 2:

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

C code for configuration:

```c
// Up-counting mode, PWM Mode 1, 25% duty cycle
TIM4->ARR = 0x9C3F;        // Auto-reload value (39999)
TIM4->CCR1 = 0x2710;       // Compare value for 25% duty cycle (10000)
TIM4->CCMR1 = 0x0060;      // PWM Mode 1 for channel 1
TIM4->CCER |= 0x0001;      // Enable output for channel 1
TIM4->CR1 &= ~TIM_CR1_DIR; // Up-counting mode
TIM4->CR1 |= TIM_CR1_CEN;  // Enable timer

// Down-counting mode, PWM Mode 2, 25% duty cycle
TIM4->ARR = 0x9C3F;        // Auto-reload value (39999)
TIM4->CCR1 = 0x752F;       // Compare value for 25% duty cycle (30000)
TIM4->CCMR1 = 0x0070;      // PWM Mode 2 for channel 1
TIM4->CCER |= 0x0001;      // Enable output for channel 1
TIM4->CR1 |= TIM_CR1_DIR;  // Down-counting mode
TIM4->CR1 |= TIM_CR1_CEN;  // Enable timer
```

Timer Capture/Compare Timing Analysis

Analyzing Timer Signal Generation

Understand the timing diagram

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

Trace the counter value

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

Analyze capture/compare events

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

Calculate timing parameters

- Actual period = (ARR+1) × (PSC+1) / timer_clock
- Duty cycle = time_active / period × 100%
- For PWM: Duty cycle = CCRx / (ARR+1) × 100% (up-counting mode)

Timer Signal Analysis Example

Analyze the following timer configuration:

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

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

Solution:

First, calculate the effective counter frequency:

- Timer input = 0.5 MHz
- Prescaler = 499+1 = 500
- Counter frequency = 0.5 MHz / 500 = 1 kHz

Calculate the period:

- ARR = 30000-1 = 29999
- Period = (ARR+1) / counter frequency = 30000 / 1000 Hz = 30 seconds

Determine the duty cycle:

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

Therefore, the PWM signal has:

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

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;
}
```

ADC/DAC Exercises

ADC Error Analysis

Analyzing ADC Errors

Understand error types

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

Determine ADC resolution

- Calculate LSB size: LSB = V_{REF} / 2^N where N is the number of bits
- Example: 3-bit ADC with V_{REF} = 2V has LSB = 2V/8 = 0.25V

Draw ideal transfer function

- Plot digital output vs. analog input
- First transition at 0.5 LSB
- Each subsequent transition at (i + 0.5) × LSB
- For N-bit ADC: 2^N - 1 total steps

Calculate error values

- **Offset error in LSB:** Measure deviation at zero input
- **Offset error in volts:** Multiply LSB by offset error in LSB
- **Gain error in LSB:** Difference between actual and ideal slope
- **Error as percentage of FSR (%FSR):** (Error in volts / FSR) × 100%
  - FSR (Full Scale Range) = V_{REF} - 1 LSB

ADC Error Analysis

For a 3-bit ADC with V_{REF} = 2V and an offset error of -1.5 LSB:

1. Draw the ideal transfer function
2. Draw the actual transfer function with the offset error
3. Calculate the LSB in volts
4. Calculate the offset error in volts
5. Calculate the offset error as a percentage of FSR

Solution:

1. **Ideal transfer function:**
   - N = 3 bits → 8 possible output codes (000 to 111)
   - LSB = $2V/2^3 = 2V/8 = 0.25V$
   - First transition: 0.5 × LSB = 0.125V
   - Code transitions at: 0.125V, 0.375V, 0.625V, 0.875V, 1.125V, 1.375V, 1.625V
2. **Actual transfer function with offset error:**
   - Offset error = -1.5 LSB
   - All transitions shift by -1.5 × 0.25V = -0.375V
   - New transitions at: -0.25V, 0V, 0.25V, 0.5V, 0.75V, 1V, 1.25V
3. **LSB in volts:**
   - LSB = $V_{REF}/2^N = 2V/8 = 0.25V$
4. **Offset error in volts:**
   - Offset error = -1.5 LSB × 0.25V/LSB = -0.375V
5. **Offset error as percentage of FSR:**
   - FSR = V_{REF} - 1 LSB = 2V - 0.25V = 1.75V
   - Offset error (%FSR) = (-0.375V / 1.75V) × 100% = -21.43%

ADC Programming

Configuring and Using STM32 ADCs

Enable peripheral clocks

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

Configure GPIO pin for analog mode

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

Configure ADC common settings

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

Configure ADC specific settings

- Set resolution in ADC_CR1 (RES bits)
- Configure scan mode if using multiple channels
- Set conversion mode in ADC_CR2 (CONT bit):
  - 0: Single conversion
  - 1: Continuous conversion
- Set data alignment in ADC_CR2 (ALIGN bit)
- Configure ADC trigger source if needed

Configure channel settings

- Set sampling time in ADC_SMPR1/2 registers
- Configure regular sequence in ADC_SQR1/2/3:
  - Set sequence length in L[3:0] bits
  - Set channel order in SQx[4:0] bits

Start conversion and read results

- Enable ADC by setting ADON bit in ADC_CR2
- Start conversion by setting SWSTART bit in ADC_CR2
- Poll EOC flag in ADC_SR or use interrupt
- Read conversion result from ADC_DR

ADC Configuration and Usage

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

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

Use register addresses from the provided documentation.

Solution:

First, let's define the relevant register addresses:

```
// ADC1 registers
#define ADC1_SR     (*((volatile uint32_t*)(0x40012000)))  // Status register
#define ADC1_CR1    (*((volatile uint32_t*)(0x40012004)))  // Control register 1
#define ADC1_CR2    (*((volatile uint32_t*)(0x40012008)))  // Control register 2
#define ADC1_DR     (*((volatile uint32_t*)(0x4001204C)))  // Data register

// ADC2 registers
#define ADC2_SR     (*((volatile uint32_t*)(0x40012100)))  // Status register

// ADC3 registers
#define ADC3_CR1    (*((volatile uint32_t*)(0x40012204)))  // Control register 1

// Bit positions
#define ADC_SR_EOC  (1 << 1)  // End of conversion flag
#define ADC_SR_OVR  (1 << 5)  // Overrun flag
#define ADC_CR1_RES_MASK (3 << 24)  // Resolution mask
#define ADC_CR1_RES_10BIT (1 << 24)  // 10-bit resolution
```

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

1. **Wait until ADC1 conversion has completed**

```
// Wait for end of conversion flag
while (!(ADC1_SR & ADC_SR_EOC)) {
    // Empty loop
}
```

2. **Check for overrun on ADC2 and set var2 accordingly**

```
uint8_t var2;
if (ADC2_SR & ADC_SR_OVR) {
    var2 = 0xFF;  // Overrun occurred
} else {
    var2 = 0x00;  // No overrun
}
```

3. **Set ADC3 resolution to 10-bit**

```
// Clear resolution bits and set to 10-bit
ADC3_CR1 &= ~ADC_CR1_RES_MASK;  // Clear bits
ADC3_CR1 |= ADC_CR1_RES_10BIT;  // Set 10-bit resolution
```

ADC Configuration and Usage (continued)

Complete function:

```
void adc_operations(void) {
    // 1. Wait until ADC1 conversion has completed
    while (!(ADC1_SR & ADC_SR_EOC)) {
        // Empty loop
    }

    // 2. Check for overrun on ADC2
    uint8_t var2;
    if (ADC2_SR & ADC_SR_OVR) {
        var2 = 0xFF;  // Overrun occurred
    } else {
        var2 = 0x00;  // No overrun
    }

    // 3. Set ADC3 resolution to 10-bit
    ADC3_CR1 &= ~ADC_CR1_RES_MASK;  // Clear bits
    ADC3_CR1 |= ADC_CR1_RES_10BIT;  // Set 10-bit resolution
}
```

Memory

Memory Technologies Overview

Semiconductor Memory Classifications

Semiconductor memories can be classified into two main categories:

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

Memory Organization

Memory devices are organized as arrays of bit cells:

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

PROM, EEPROM, and Flash Memory

PROM (Programmable Read-Only Memory)

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

EEPROM (Electrically Erasable PROM)

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

Flash Memory

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

NOR vs. NAND Flash

NOR Flash:

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

NAND Flash:

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

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.

Memory Exercises

Memory Technology Comparison

Comparing Memory Technologies

Identify memory characteristics

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

Compare memory performance metrics

- **Access time:** Time to read/write data
  - SRAM: 2-10ns
  - SDRAM: 60ns+ for first access, then fast for burst
  - NOR Flash: ĩ20ns read
  - NAND Flash: Slow random access (25μs first byte)
- **Density:** Storage capacity per unit area
  - SRAM: Low (large cells, expensive)
  - DRAM: High (small cells, inexpensive)
  - NAND Flash: Very high (highest density, lowest cost per bit)
- **Power consumption:** Energy required for operation
  - SRAM: Low static power (no refresh needed)
  - DRAM: Higher (requires refresh)
  - Flash: Very low when not being written/erased

Analyze application suitability

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

Memory Technology Comparison

Compare SDRAM (Synchronous Dynamic RAM) and SRAM (Static RAM) in terms of:

1. Storage mechanism and structure
2. Access patterns and timing characteristics
3. Density and cost
4. Refresh requirements
5. Typical applications

Solution:

1. **Storage mechanism and structure:**
   - **SRAM:** Uses flip-flop circuits with 6 transistors per bit. Stores data as long as power is applied, without requiring refresh.
   - **SDRAM:** Uses a single capacitor and transistor per bit. Stores data as charge in a capacitor that leaks over time.
2. **Access patterns and timing:**
   - **SRAM:** Provides fast random access with consistent timing for all accesses (5ns). All operations take roughly the same time.
   - **SDRAM:** Synchronous interface with clock. Has higher latency for first access in a row (60ns), but very fast subsequent access within same row. Optimized for burst transfers.
3. **Density and cost:**
   - **SRAM:** Low density due to large cell size (6 transistors). Higher cost per bit. Typically limited to 64Mb per device.
   - **SDRAM:** High density due to small cell size (1 transistor + 1 capacitor). Lower cost per bit. Can reach 4Gb or more per device.
4. **Refresh requirements:**
   - **SRAM:** No refresh required (static). Maintains data as long as power is applied.
   - **SDRAM:** Requires periodic refresh (dynamic). Capacitors discharge within milliseconds and must be refreshed regularly.
5. **Typical applications:**
   - **SRAM:** CPU cache memory, small buffer memory, applications requiring fast random access.
   - **SDRAM:** Main system memory, large buffers, applications requiring high capacity with reasonable access speed.

External Memory Configuration

Connecting External Memory to STM32

Identify external memory type and requirements

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

Configure FMC (Flexible Memory Controller)

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

Configure GPIO pins for FMC

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

Understand memory mapping

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

External Memory Connection Analysis

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

Answer:

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

Solution:

1. **Number of address pins:**
   - Memory size = 64K = 2^{16} bytes
   - Need 16 address lines: ADDR[15:0]
2. **FMC signals to connect:**
   - Address lines: A[15:0]
   - Data lines: D[7:0] (8-bit data bus)
   - Chip select: NE[3] (for Bank 3 based on address 0x6800'0000)
   - Output Enable: NOE
   - Write Enable: NWE
3. **Highest memory address:**
   - Lowest address: 0x6800'0000
   - Memory size: 64K = 0x10000 bytes
   - Highest address: 0x6800'0000 + 0xFFFF = 0x6800'FFFF
4. **Multiple address access:**
   - This occurs due to partial address decoding
   - Only address lines A[15:0] are connected to the memory chip
   - Higher address bits A[25:16] are not decoded/connected
   - Any address where A[27:26] = 01 (Bank 2) and A[15:0] match will access the same memory location
   - Example: Addresses 0x6800'0000, 0x6900'0000, 0x6A00'0000, etc. all access the same byte in memory
   - The number of 64KB address blocks that map to the same memory is $2^{10} = 1024$ (from the 10 undecoded bits A[25:16])

Memory Access Decomposition

Analyzing Memory Access Patterns

Determine memory access type

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

Memory address decomposition

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

Analyze byte enables for sub-word accesses

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

Determine access pattern

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

Memory Access Analysis

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

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

Solution:

1. **Memory device and bank:**
   - Address 0x6402'8F21 begins with 0x64...
   - From address bits [27:26] = 01, this is in Bank 2
   - This activates NE2 (chip select 2)
2. **Memory location in device:**
   - Only address bits [25:1] are passed to the SRAM device
   - Bit [0] is used to select the high/low byte within a 16-bit word
   - Memory location = 0x0014'7490 (shifted right by 1 bit and ignoring higher bits that aren't connected)
3. **Byte line activation:**
   - Address bit [0] = 1, so we're accessing the high byte in the 16-bit word
   - For 16-bit memory: NBL[1:0] = 10 (high byte active)
   - NBL[0] = 1 (not active), NBL[1] = 0 (active)
4. **Complete address decomposition:**
   - 0x6402'8F21 decomposed:
     - Bits [31:28]: 0x6 (not used for decoding)
     - Bits [27:26]: 01 (Bank 2, activates NE2)
     - Bits [25:1]: Memory address within SRAM (0x0014'7490)
     - Bit [0]: 1 (select high byte)
   - FMC signals during access:
     - NE2 = 0 (active)
     - A[24:0] = 0x0014'7490 (memory address)
     - NBL[1:0] = 10 (access high byte)
     - NWE = 0 (active, writing)
     - Data appears on D[15:8] (high byte)

Cache

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

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

Principle of Locality

Memory Hierarchy and Locality

Programs usually access small regions of memory in a given interval of time.
The memory hierarchy in computer systems is designed to exploit the principle of locality:
- **Spatial Locality**: If a memory location is accessed, nearby locations are likely to be accessed soon
- → Current data location is likely being close to next accessed location
  - Example: Sequential access to array elements
- **Temporal Locality**: If a memory location is accessed, it's likely to be accessed again soon
- → Current data location is likely being accessed again in near future
  - Example: Loop variables, frequently called functions

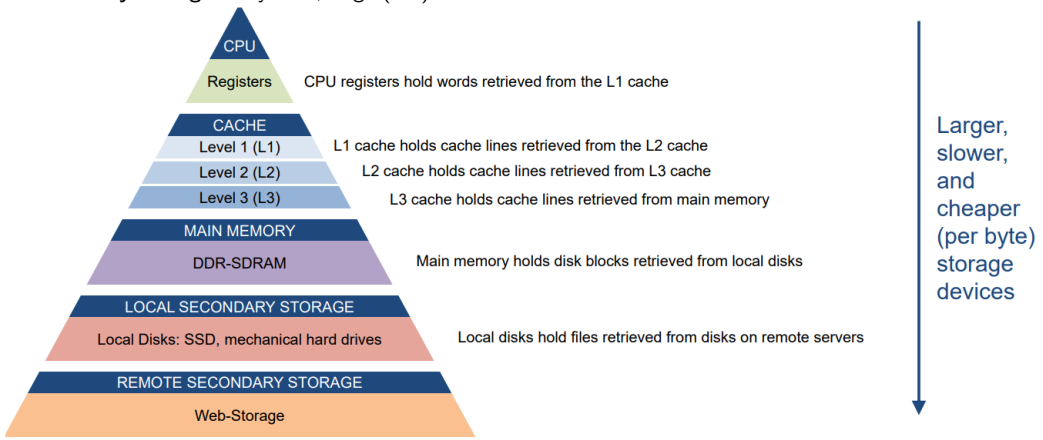
The memory hierarchy takes advantage of these patterns:
- Faster, smaller memories (cache) store recently/frequently accessed data
- Larger, slower memories (main memory, disk) store less frequently accessed data

Principle of Locality

```
for (int i = 0; i < N; i++) {    // incremental access
    a[i] = b[i];                 // spatial locality
}

if (a[1234] == a[4321]) {        // temporal locality
    a[1234] = 0;                 // access to same location again
}
```

Memory Hierarchy Levels

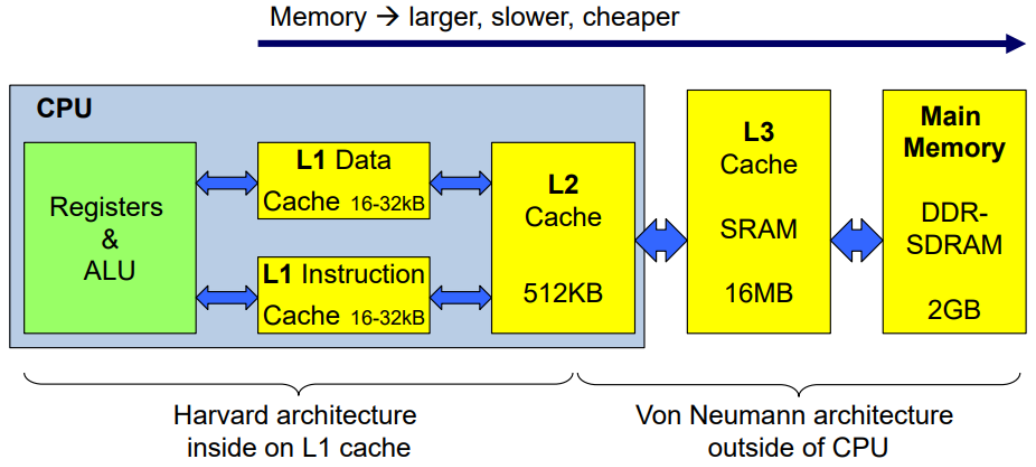
Typical memory hierarchy in a modern system:
- **CPU Registers**: Fastest, smallest (bytes)
- **L1 Cache**: Very fast, small (KB)
  - Often split into instruction and data caches (Harvard architecture)
- **L2 Cache**: Fast, medium size (hundreds of KB)
- **L3 Cache**: Moderately fast, larger (MB)
- **Main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



Cache Mechanics

Cache Levels

Typical Cache Architecture



Cache Operation

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

Cache Terminology

Key terms in cache design:

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

Cache Organization

Memory Addressing for Cache

A memory address is typically divided into:

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

Tag	Index	Offset

The size of each field depends on the cache organization.

Cache Organization Types

Three main cache organization strategies:

- **Fully Associative**:
  - Any memory block can be placed in any cache line
  - Address format: [Tag | Offset]
  - Highest flexibility, best hit rate
  - Complex hardware: requires comparing tag with all cache lines
- **Direct Mapped**:
  - Each memory block maps to exactly one cache line
  - Address format: [Tag | Index | Offset]
  - Simple hardware: only one comparison needed
  - Lower hit rate due to conflicts
- **N-Way Set Associative**:
  - Cache divided into sets, each with N lines
  - Memory block maps to specific set, can be placed in any line within that set
  - Address format: [Tag | Set Index | Offset]
  - Compromise between flexibility and complexity

Fully Associative Cache

In a fully associative cache:

- Any memory block can be stored in any cache line
- Address is divided into tag and offset only
- Each cache line stores:
  - Valid bit (is data valid?)
  - Tag (which memory block)
  - Data (contents of memory block)
- On memory access, the tag is compared with all cache lines in parallel
- Requires many comparators (one per cache line)
- Provides highest flexibility but most complex hardware

Direct Mapped Cache

In a direct mapped cache:

- Each memory block maps to exactly one cache line
- Mapping function: Line = Block $\bmod m$ (where m is number of cache lines)
- Address is divided into tag, index, and offset
- The index directly selects which cache line to check
- Only one tag comparison needed
- Simple hardware but can suffer from conflict misses
  - Different memory blocks mapping to the same cache line

N-Way Set Associative Cache

In an N-way set associative cache:

- Cache is divided into sets, each containing N lines
- Memory block maps to a specific set, can go in any line within that set
- Mapping function: Set = Block $\bmod s$ (where s is number of sets)
- Address is divided into tag, set index, and offset
- Set index selects which set to check
- N tag comparisons needed (one per line in the set)
- Compromise between fully associative and direct mapped
  - More flexible than direct mapped
  - Less hardware than fully associative

Cache Organization Calculation

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

- Direct mapped
- 4-way set associative

Given:

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

1. Direct Mapped Cache:

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

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

2. 4-Way Set Associative:

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

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

Cache Performance

Cache Miss Types

Three fundamental types of cache misses:

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

Cache Performance

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

$$AMAT = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty} \tag{9}$$

Example:

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

$$AMAT = 1 + 0.03 \times 100 = 1 + 3 = 4 \text{ cycles} \tag{10}$$

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

$$AMAT = 1 + 0.01 \times 100 = 1 + 1 = 2 \text{ cycles} \tag{11}$$

This is a 50% performance improvement!

Cache Size vs. Hit Rate

Relationship between cache parameters and hit rate:

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

Replacement and Write Strategies

Replacement Strategies

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

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

Write Strategies

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

- **Write-Through**:
  - Write data to both cache and main memory
  - Memory always consistent with cache
  - Slower for writes (must wait for memory)
  - Simpler coherence in multiprocessor systems
- **Write-Back**:
  - Write data only to cache
  - Mark block as "dirty"
  - Write to memory only when block is evicted
  - Faster for repeated writes to same block
  - More complex coherence handling

For write misses, two approaches:

- **Write-Allocate**:
  - Fetch block into cache, then update
  - Works well with write-back
- **No-Write-Allocate**:
  - Write directly to memory, don't fetch block
  - Works well with write-through

Programmer's Perspective

Cache-Friendly Programming

Maximize spatial locality

Access memory in sequential patterns whenever possible.

Maximize temporal locality

Reuse recently accessed data before it gets evicted from cache.

Be aware of cache line size

Structure data to minimize cache line crossings.

Consider memory layout

Organize multidimensional arrays to match access patterns.

```
// Cache-unfriendly code (column-major traversal of row-major array)
for (j = 0; j < N; j++) {
    for (i = 0; i < N; i++) {
        sum += array[i][j];  // Poor spatial locality
    }
}

// Cache-friendly code (row-major traversal of row-major array)
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        sum += array[i][j];  // Good spatial locality
    }
}
```

Cache Performance Optimization

General guidelines for cache-friendly programming:

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

Optimizing Matrix Multiplication for Cache

Optimize a simple matrix multiplication algorithm to be cache-friendly.

Standard matrix multiplication:

```
// Standard implementation - cache unfriendly
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        sum = 0;
        for (k = 0; k < N; k++) {
            sum += A[i][k] * B[k][j];  // Poor locality for B
        }
        C[i][j] = sum;
    }
}
```

Cache-optimized version using blocking:

```
// Blocked implementation - cache friendly
for (i = 0; i < N; i += BLOCK_SIZE) {
    for (j = 0; j < N; j += BLOCK_SIZE) {
        for (k = 0; k < N; k += BLOCK_SIZE) {
            // Block multiplication
            for (ii = i; ii < min(i+BLOCK_SIZE, N); ii++) {
                for (jj = j; jj < min(j+BLOCK_SIZE, N); jj++) {
                    sum = C[ii][jj];
                    for (kk = k; kk < min(k+BLOCK_SIZE, N); kk++) {
                        sum += A[ii][kk] * B[kk][jj];
                    }
                    C[ii][jj] = sum;
                }
            }
        }
    }
}
```

The blocked implementation works on small submatrices that fit in cache, improving temporal locality.

Cache Exercises

Cache Organization and Addressing

Analyzing Cache Organization

Identify cache parameters

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

Address decomposition

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

Determine cache organization

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

Map memory addresses to cache

- Calculate set number: (address / block size) mod (number of sets)
- Calculate tag: address / (block size $\times$ number of sets)
- Calculate offset: address mod block size

Cache Organization Analysis

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

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

Solution:

1. **Number of cache lines:**
   - Cache size = 4 KiB = 4096 bytes
   - Line size = 16 bytes
   - Number of lines = 4096 / 16 = 256 lines
2. **Address bit fields:**
   - Offset bits = $\log_2(16) = 4$ bits (addresses byte within the line)
   - Index bits = $\log_2(256) = 8$ bits (selects the cache line)
   - Tag bits = 32 - (4 + 8) = 20 bits (identifies which memory block is cached)
3. **Address decomposition for 0x1234ABCD:**
   - Convert to binary: 0001 0010 0011 0100 1010 1011 1100 1101
   - Offset: last 4 bits = 1101 = 0xD
   - Index: next 8 bits = 1011 1100 = 0xBC
   - Tag: remaining 20 bits = 0001 0010 0011 0100 1010 = 0x1234A
4. **Other addresses mapping to the same cache line:**
   - Addresses with the same index (0xBC) but different tags would map to the same line
   - General form: 0xXXXXXBCY where:
     - XXXXX can be any value except 0x1234A (different tag)
     - BC is fixed (same index)
     - Y can be any value from 0 to F (different offset within the line)
   - Examples: 0x0034ABCD, 0x5678ABCF, 0x9ABCDBCE, etc.
   - These addresses would cause cache conflicts if accessed in sequence

Cache Performance Analysis

Analyzing Cache Hit Rates and Performance

Calculate basic cache metrics

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

Identify types of cache misses

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

Analyze memory access patterns

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

Evaluate impact of cache parameters

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

Cache Performance Calculation

Consider a system with the following cache parameters:

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

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

Solution:

1. **Calculate AMAT for both configurations:**
   - AMAT = Hit time + (Miss rate × Miss penalty)
   - For configuration A:
     - $\text{AMAT}_A = 1 \text{ cycle} + (0.03 \times 100 \text{ cycles}) = 1 + 3 = 4 \text{ cycles}$
   - For configuration B:
     - $\text{AMAT}_B = 1 \text{ cycle} + (0.01 \times 100 \text{ cycles}) = 1 + 1 = 2 \text{ cycles}$
2. **Effect of increased associativity:**
   - Higher associativity reduces conflict misses
   - In configuration A (lower associativity), more memory blocks map to the same cache set
   - With configuration B (doubled associativity), each set can hold twice as many blocks
   - This reduces the chance of having to evict a useful block due to set conflicts
   - The improved miss rate (from 3% to 1%) is primarily due to the reduction in conflict misses
   - Note that capacity misses and compulsory misses would remain the same
3. **Cycles saved:**
   - Total accesses = 1,000,000
   - Cycles for configuration A = 1,000,000 × 4 = 4,000,000 cycles
   - Cycles for configuration B = 1,000,000 × 2 = 2,000,000 cycles
   - Cycles saved = 4,000,000 - 2,000,000 = 2,000,000 cycles

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

Cache Programming Considerations

Optimizing Code for Cache Performance

Analyze array access patterns

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

Improve spatial locality

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

Improve temporal locality

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

Avoid cache conflicts

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

Cache-Friendly Programming Example

Consider a matrix multiplication function for 1000×1000 matrices. Two implementations are shown:

```
// Version A - Row by Column
for(i = 0; i < 1000; i++) {
    for(j = 0; j < 1000; j++) {
        for(k = 0; k < 1000; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}

// Version B - Blocked/Tiled
for(i = 0; i < 1000; i += 64) {
    for(j = 0; j < 1000; j += 64) {
        for(k = 0; k < 1000; k += 64) {
            for(ii = i; ii < min(i+64, 1000); ii++) {
                for(jj = j; jj < min(j+64, 1000); jj++) {
                    for(kk = k; kk < min(k+64, 1000); kk++) {
                        C[ii][jj] += A[ii][kk] * B[kk][jj];
                    }
                }
            }
        }
    }
}
```

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

Solution:

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

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

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

Software State Machines

Introduction - FSM in Hardware vs Software

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

Hardware vs Software FSM Implementation

Hardware FSM

- Intrinsically parallel - multiple FSMs process simultaneously
- Clock-driven - inputs evaluated at clock edges
- Unchanged input signals create no overhead
- Uses flip-flops to store internal state
- State can only change on a clock edge
- Typically implemented as either Moore or Mealy machines

Software FSM

- Intrinsically sequential - CPU processes one FSM after another
- Event-driven - only evaluate FSM when inputs change
- Evaluating all inputs on each function call would waste CPU time
- Synchronization issues in sequential systems if using ßynchronous clock approach"
- More flexibility in implementation patterns
- Better suited for complex state transitions and conditions

Moore vs Mealy Hardware FSM

// Moore FSM - Output depends only on state Moore FSM:

- State A (output: 0) → State B (output: 0) // event: I=0
- State B (output: 0) → State C (output: 1) // event: I=0
- State C (output: 1) → State D (output: 0) // event: I=1
- State D (output: 0) → State A (output: 0) // event: I=1

// Mealy FSM - Output depends on state and input Mealy FSM:

- State A (output: 0) → State B (output: 0) // event: I=1
- State B (output: 1) → State A (output: 1) // event: I=0
- State B (output: 0) → State C (output: 0) // event: I=1
- State C (output: 1) → State A (output: 1) // event: I=0

Moore vs Mealy FSMs

Moore FSM

In a Moore FSM, the outputs are determined solely by the current state, not directly by the inputs.

$$\text{State} \rightarrow \text{Output}$$

Mealy FSM

In a Mealy FSM, the outputs are determined by both the current state and the current inputs.

$$(\text{State}, \text{Input}) \rightarrow \text{Output}$$

Reactive Systems

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

Components of a Reactive System

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

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

Common applications of state-event models include:

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

Modeling State Machines in UML

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

UML State Diagram Elements

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

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

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

Simple light switch state diagram in UML:

state_off → state_on (event: on / action: lamp_on)
state_on → state_off (event: off / action: lamp_off)

With an initial default transition to state_off.
Note that the transition öffïn state_off is not shown because it would have no effect and trigger no action.

Traffic Light Controller // UML state diagram for a simple traffic light controller
Initial state → Red
Red → Red_Yellow (event: timer_expired / action: set_red_yellow) Red_Yellow → Green (event: timer_expired / action: set_green) Green → Yellow (event: timer_expired / action: set_yellow) Yellow → Red (event: timer_expired / action: set_red)
// Each state has an associated timer that generates // a timer_expired event after a state-specific duration

UML State Diagram Design Rules

Basic Rules

- Every state diagram must have an initial state
- Each state has to be reachable through a transition
- The state diagram has to be deterministic - for each event it has to be defined which transition is triggered
- Avoid inconsistent or contradictory transitions that could lead to ambiguity

Semantics of UML FSM

- FSM is passive - only reacts to events from outside
- Always has a defined state
- Reaction depends on current state
- Event is discarded if no transition exists for it in current state
- Run-to-completion - once started, a transition cannot be interrupted
- Strives to avoid querying additional input - transitions should depend only on the event, not additional inputs

Motor Control FSM Example Consider a motor control system with the following requirements:

- Operator can turn motor on/off using start/stop buttons
- System shuts down motor if temperature limit is exceeded
- Restarting after temperature shutdown requires acknowledging the fault
- Normal operation and alarm conditions have different indicator lamps

UML state diagram would include:

- States: idle, operate, fault, wait_ack
- Events: start, stop, ack, t_limit_exceeded, t_normalized
- Actions: motor_on, motor_off, op_lamp_on, op_lamp_off, al_lamp_on, al_lamp_off

Key transitions:

- idle → operate (event: start / action: motor_on, op_lamp_on)
- operate → idle (event: stop / action: motor_off, op_lamp_off)
- operate → fault (event: t_limit_exceeded / action: motor_off, op_lamp_off, al_lamp_on)
- fault → wait_ack (event: t_normalized / action: al_lamp_off)
- wait_ack → idle (event: ack)

Implementation in C

Basic Implementation Structure Software implementation of a state machine typically uses two main functions:

- **fsm_init():** Initializes the state machine to its default state
- **fsm_handle_event(event):** Processes an incoming event based on current state

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

FSM Implementation Pattern

```c
// State and event enumerations
typedef enum {
    STATE_A,
    STATE_B,
    STATE_C,
    // ...other states
} state_t;

typedef enum {
    NO_EVENT,
    EVENT_X,
    EVENT_Y,
    // ...other events
} event_t;

// Current state variable
static state_t state;

// Initialization function
void fsm_init(void) {
    state = STATE_A;  // Set initial state
    // Additional initialization if needed
}

// Event handler function
void fsm_handle_event(event_t event) {
    switch (state) {
        case STATE_A:
            switch (event) {
                case EVENT_X:
                    action1();
                    state = STATE_B;
                    break;
                case EVENT_Y:
                    action2();
                    // No state change
                    break;
                default:
                    // Event ignored in this state
                    break;
            }
            break;
        case STATE_B:
            // Handle events for state B
            break;
        // Other states
    }
}

// Main loop
int main(void) {
    event_t event;

    fsm_init();

    while (1) {
        event = get_event();
        if (event != NO_EVENT) {
            fsm_handle_event(event);
        }
    }
}
```

LED Control FSM Implementation

```c
// State and event definitions
typedef enum {
    STATE_OFF,
    STATE_ON
} state_t;

typedef enum {
    NO_SWITCH,
    S0,
    S1
} event_t;

// Current state
static state_t state = STATE_OFF;

// Initialize the FSM
void fsm_init(void) {
    state = STATE_OFF;
    // Initialize hardware, ensure LED is off
    led_off();
}

// Handle incoming events
void fsm_handle_event(event_t event) {
    switch (state) {
        case STATE_OFF:
            if (event == S0) {
                led_on();  // Action
                state = STATE_ON;  // State transition
            }
            break;
        case STATE_ON:
            if (event == S0) {
                led_off();  // Action
                state = STATE_OFF;  // State transition
            }
            break;
    }
}

// Event detection function
event_t get_event(void) {
    // Read hardware inputs
    uint32_t switches = read_switches();

    // Check switch positions
    if (switch_changed(switches, 0)) {
        return S0;
    } else if (switch_changed(switches, 1)) {
        return S1;
    }

    return NO_SWITCH;  // No event
}

// Main program
int main(void) {
    event_t event;

    // System initialization
    peripherals_init();
    fsm_init();

    // Main loop
    while (1) {
        event = get_event();
        if (event != NO_SWITCH) {
            fsm_handle_event(event);
        }
    }
}
```

Interaction of FSMs

FSM Interaction Components

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

Event Queue for FSM Interaction

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

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

A lamp FSM reacts to events created by a button FSM:
Button FSM states: released, pressed
Button port: out (pressed, released)
Lamp FSM states: off, on
Lamp port: in (pressed, released)
Button transitions:
- released → pressed / out.pressed
- pressed → released / out.released

Lamp transitions:
- off → on (on event in.pressed)
- on → off (on event in.released)

The communication between these two FSMs happens through messages passed via their ports.

Building Event-Based FSM Systems

System Structure

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

Implementation

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

Benefits

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

Application Example: Car Wash Control System

Car Wash Control System Based on the exercise, a car wash control system can be modeled as a state machine with the following specifications:

States

- rest (idle state, waiting for start button)
- wash (washing with water and shampoo)
- rinse (rinsing with water only)
- dry (drying with air)

Events

- start (start button pressed)
- stop (stop button pressed)
- time_out (timer has expired)

Actions

- water_on (turn on water)
- water_off (turn off water)
- shampoo_on (mix in shampoo)
- shampoo_off (stop mixing shampoo)
- air_on (turn on air for drying)
- air_off (turn off air)
- timer_start (start the timer)

Key Transitions

- rest → wash (event: start / actions: water_on, shampoo_on, timer_start)
- wash → rinse (event: time_out / actions: shampoo_off, timer_start)
- rinse → dry (event: time_out / actions: water_off, air_on, timer_start)
- dry → rest (event: time_out / actions: air_off)
- wash/rinse/dry → rest (event: stop / appropriate actions to turn off actuators)

This example demonstrates a typical process control application with sequential states, timer-based transitions, and the ability to abort the process at any point.

Car Wash FSM Implementation

```c
// States
typedef enum {
    STATE_REST,
    STATE_WASH,
    STATE_RINSE,
    STATE_DRY
} state_t;

// Events
typedef enum {
    NO_EVENT,
    EVENT_START,
    EVENT_STOP,
    EVENT_TIMEOUT
} event_t;

// Current state
static state_t state = STATE_REST;

// Initialize the FSM
void fsm_init(void) {
    state = STATE_REST;
    // Make sure all actuators are off
    water_off();
    shampoo_off();
    air_off();
}

// Handle events
void fsm_handle_event(event_t event) {
    switch (state) {
        case STATE_REST:
            if (event == EVENT_START) {
                water_on();
                shampoo_on();
                timer_start();
                state = STATE_WASH;
            }
            break;

        case STATE_WASH:
            if (event == EVENT_TIMEOUT) {
                shampoo_off();
                timer_start();
                state = STATE_RINSE;
            } else if (event == EVENT_STOP) {
                water_off();
                shampoo_off();
                state = STATE_REST;
            }
            break;

        case STATE_RINSE:
            if (event == EVENT_TIMEOUT) {
                water_off();
                air_on();
                timer_start();
                state = STATE_DRY;
            } else if (event == EVENT_STOP) {
                water_off();
                state = STATE_REST;
            }
            break;

        case STATE_DRY:
            if (event == EVENT_TIMEOUT || event == EVENT_STOP) {
                air_off();
                state = STATE_REST;
            }
            break;
```

Conclusion and Best Practices

Key Differences: Software vs Hardware FSMs

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

FSM Design Best Practices

Design Phase

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

Implementation Phase

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

Testing and Debugging

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

Interrupt Performance

Event Detection Methods

Polling vs. Interrupt-Driven I/O

Two primary methods for detecting events in embedded systems:

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

Polling Implementation

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

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

Advantages:

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

Disadvantages:

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

Interrupt-Driven I/O

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

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

Advantages:

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

Disadvantages:

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

Implementing Polling in C

Step 1: Identify status registers

Determine which peripheral registers contain status information.

Step 2: Create a polling loop

Implement a loop that regularly checks the status flags.

Step 3: Check for events

Test specific bits in the status registers to detect events.

Step 4: Handle detected events

Process events when their status flags are set.

Step 5: Clear status flags

Reset status flags to prepare for next event detection.

```
// Main polling loop
while (1) {
    // Check SPI transmit buffer empty flag
    if (SPI1->SR & SPI_SR_TXE) {
        // Handle SPI transmission
        if (spi_tx_count < spi_tx_length) {
            SPI1->DR = spi_tx_buffer[spi_tx_count++];
        }
    }

    // Check UART receive data register not empty flag
    if (USART2->SR & USART_SR_RXNE) {
        // Handle UART reception
        uint8_t data = USART2->DR;
        process_uart_data(data);
    }

    // Check ADC end of conversion flag
    if (ADC1->SR & ADC_SR_EOC) {
        // Handle ADC conversion complete
        uint16_t adc_value = ADC1->DR;  // Reading DR clears EOC flag
        process_adc_data(adc_value);
    }

    // Other system tasks
    process_system_tasks();
}
```

Configuring Interrupt-Driven I/O

Step 1: Configure interrupt sources

Enable specific interrupt sources in peripherals.

Step 2: Configure NVIC

Set up the Nested Vectored Interrupt Controller for the interrupts.

Step 3: Implement ISRs

Create Interrupt Service Routines to handle specific events.

Step 4: Enable global interrupts

Enable the global interrupt flag.

```
// Step 1: Configure SPI interrupt
void configure_spi_interrupt(void) {
    // Enable SPI peripheral clock
    RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;

    // Configure SPI parameters
    SPI1->CR1 = SPI_CR1_MSTR | SPI_CR1_SSI | SPI_CR1_SSM;

    // Enable SPI TX buffer empty interrupt
    SPI1->CR2 |= SPI_CR2_TXEIE;

    // Step 2: Configure NVIC for SPI1
    NVIC_SetPriority(SPI1_IRQn, 2);  // Set priority level
    NVIC_EnableIRQ(SPI1_IRQn);       // Enable interrupt in NVIC
}

// Step 3: Implement SPI1 ISR
void SPI1_IRQHandler(void) {
    // Check if TX buffer empty interrupt
    if (SPI1->SR & SPI_SR_TXE) {
        if (spi_tx_count < spi_tx_length) {
            // Send next byte
            SPI1->DR = spi_tx_buffer[spi_tx_count++];
        } else {
            // Transfer complete, disable interrupt
            SPI1->CR2 &= ~SPI_CR2_TXEIE;
        }
    }
}

// Main function
int main(void) {
    // Initialize system
    system_init();

    // Configure SPI interrupt
    configure_spi_interrupt();

    // Step 4: Enable global interrupts
    __enable_irq();

    // Main loop - can perform other tasks
    while (1) {
        process_system_tasks();
    }
}
```

Interrupt Performance Analysis

Key Performance Metrics

Several metrics characterize interrupt performance:

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

System Impact Calculation

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

$$\text{Impact (\%)} = f_{INT} \times t_{ISR} \times 100\% \tag{12}$$

Examples:

- Keyboard interrupt: $f_{INT} = 20$ Hz, $t_{ISR} = 6$ µs

$$\text{Impact} = 20 \text{ Hz} \times 6 \text{ µs} \times 100\% = 0.012\% \tag{13}$$

- High-speed serial interface: $f_{INT} = 28,800$ Hz, $t_{ISR} = 6$ µs

$$\text{Impact} = 28,800 \text{ Hz} \times 6 \text{ µs} \times 100\% = 17.3\% \tag{14}$$

Interrupt Overload Conditions

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

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

Factors that may lead to overload:

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

Optimizing ISR Performance

Keep ISRs short

Perform only time-critical operations in the ISR.

Defer processing to main loop

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

Use hardware features

Take advantage of DMA and peripheral buffers.

Apply appropriate priorities

Assign higher priorities to more time-critical interrupts.

Disable interrupts judiciously

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

```
// Non-optimized ISR
void USART2_IRQHandler(void) {
    if (USART2->SR & USART_SR_RXNE) {
        // Directly process data in ISR (slow)
        char c = USART2->DR;
        process_character(c);  // Time-consuming
        update_display();      // Even more time-consuming
    }
}

// Optimized ISR
volatile uint8_t rx_buffer[256];
volatile uint8_t rx_write_idx = 0;
volatile uint8_t rx_read_idx = 0;
volatile bool new_data_available = false;

void USART2_IRQHandler(void) {
    if (USART2->SR & USART_SR_RXNE) {
        // Only store data in buffer (fast)
        rx_buffer[rx_write_idx++] = USART2->DR;
        new_data_available = true;
    }
}

// Process data in main loop
void main(void) {
    // Initialize
    system_init();

    while (1) {
        if (new_data_available) {
            // Process all received data
            while (rx_read_idx != rx_write_idx) {
                process_character(rx_buffer[rx_read_idx++]);
            }

            // Update display once after processing all characters
            update_display();

            // Reset flag
            new_data_available = false;
        }

        // Other system tasks
        process_system_tasks();
    }
}
```

Interrupt Latency

Interrupt Latency

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

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

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

Sources of Interrupt Latency

Several factors contribute to interrupt latency:

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

Managing Interrupt Latency

Use interrupt priorities

Assign appropriate priorities based on timing requirements.

Limit interrupt disable periods

Minimize sections where interrupts are disabled.

Optimize context switching

Use processor features that minimize context save/restore.

Preempt lower-priority tasks

Allow high-priority interrupts to preempt less critical ones.

Move waiting loops to main program

Don't block inside ISRs waiting for slow peripherals.

```
// Poor: Blocking in ISR
void SPI1_IRQHandler(void) {
    // Read data from input
    input_data = read_input_source();

    // Process data
    processed_data = process_data(input_data);

    // Wait for output device to be ready (blocking)
    while (!(SPI1->SR & SPI_SR_TXE)) { }

    // Write data to output
    SPI1->DR = processed_data;
}

// Better: Queue-based approach
void SPI1_IRQHandler(void) {
    // Read data from input
    input_data = read_input_source();

    // Process data
    processed_data = process_data(input_data);

    // Add to output queue and return
    queue_add(output_queue, processed_data);
}

// In main loop
void main(void) {
    // Initialize
    system_init();

    while (1) {
        // Check if there's data to send and SPI is ready
        if (!queue_empty(output_queue) && (SPI1->SR & SPI_SR_TXE)) {
            // Send next item
            SPI1->DR = queue_get(output_queue);
        }

        // Other tasks
        process_system_tasks();
    }
}
```

Pre-emption and Priority

Nested Vector Interrupt Controller (NVIC)

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

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

Interrupt Preemption

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

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

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

Priority-Based Interrupt Handling

Key aspects of priority-based interrupt handling:

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

Configuring Interrupt Priorities

Step 1: Analyze timing requirements

Determine which interrupts are most time-critical.

Step 2: Group interrupts by importance

Create logical groups based on system criticality.

Step 3: Assign priority levels

Configure NVIC priority registers for each interrupt.

Step 4: Configure priority grouping

Set priority group using NVIC_SetPriorityGrouping().

```
// Configure interrupt priorities
void configure_interrupt_priorities(void) {
    // Set priority grouping (4 bits for pre-emption priority, 0 bits for
        sub-priority)
    NVIC_SetPriorityGrouping(3);

    // Critical interrupts (highest priority)
    NVIC_SetPriority(EXTI0_IRQn, 0);      // Emergency stop button
    NVIC_SetPriority(TIM1_UP_IRQn, 1);    // Critical timing control

    // Medium priority interrupts
    NVIC_SetPriority(SPI1_IRQn, 4);       // Sensor data acquisition
    NVIC_SetPriority(USART2_IRQn, 5);     // Communication

    // Lower priority interrupts
    NVIC_SetPriority(ADC_IRQn, 10);       // Regular ADC sampling
    NVIC_SetPriority(TIM3_IRQn, 11);      // Status LED updates

    // Enable the interrupts
    NVIC_EnableIRQ(EXTI0_IRQn);
    NVIC_EnableIRQ(TIM1_UP_IRQn);
    NVIC_EnableIRQ(SPI1_IRQn);
    NVIC_EnableIRQ(USART2_IRQn);
    NVIC_EnableIRQ(ADC_IRQn);
    NVIC_EnableIRQ(TIM3_IRQn);
}
```

Interrupt-Driven State Machines

Integrating Interrupts and State Machines

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

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

Implementing Interrupt-Driven FSM

Step 1: Create event queue

Implement a buffer to store events from ISRs.

Step 2: Implement ISRs

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

Step 3: Implement state machine

Create state machine logic to process events from queue.

Step 4: Connect with main loop

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

```
// Define event types and queue
typedef enum {
    EVENT_BUTTON_PRESS,
    EVENT_TIMEOUT,
    EVENT_SENSOR_TRIGGER,
    EVENT_NO_EVENT
} event_t;

#define EVENT_QUEUE_SIZE 16
event_t event_queue[EVENT_QUEUE_SIZE];
int queue_head = 0;
int queue_tail = 0;
int queue_count = 0;

// Add event to queue (called from ISRs)
void queue_add_event(event_t event) {
    if (queue_count < EVENT_QUEUE_SIZE) {
        event_queue[queue_tail] = event;
        queue_tail = (queue_tail + 1) % EVENT_QUEUE_SIZE;
        queue_count++;
    }
}

// Get event from queue (called from main loop)
event_t queue_get_event(void) {
    event_t event = EVENT_NO_EVENT;

    if (queue_count > 0) {
        event = event_queue[queue_head];
        queue_head = (queue_head + 1) % EVENT_QUEUE_SIZE;
        queue_count--;
    }

    return event;
}

// Button interrupt handler
void EXTI0_IRQHandler(void) {
    // Check if EXTI0 interrupt occurred
    if (EXTI->PR & EXTI_PR_PR0) {
        // Add button press event to queue
        queue_add_event(EVENT_BUTTON_PRESS);

        // Clear pending bit
        EXTI->PR = EXTI_PR_PR0;
    }
}

// Timer interrupt handler
void TIM2_IRQHandler(void) {
    if (TIM2->SR & TIM_SR_UIF) {
        // Add timeout event to queue
        queue_add_event(EVENT_TIMEOUT);

        // Clear update interrupt flag
        TIM2->SR &= ~TIM_SR_UIF;
    }
}
```

Complete Interrupt-Driven FSM Example
Implement a button-controlled LED system with debouncing using timer interrupts.

The system has three states: - OFF: LED is off - ON: LED is on at full brightness - BLINK: LED is blinking Events: - SHORT_PRESS: Button pressed briefly - LONG_PRESS: Button held for >1 second - BLINK_TIMER: Timer for blinking

```
// Event definitions
typedef enum {
    EVENT_NONE,
    EVENT_SHORT_PRESS,
    EVENT_LONG_PRESS,
    EVENT_BLINK_TIMER
} event_t;

// State definitions
typedef enum {
    STATE_OFF,
    STATE_ON,
    STATE_BLINK
} state_t;

// Global variables
volatile event_t event_queue[10];
volatile uint8_t queue_head = 0;
volatile uint8_t queue_tail = 0;
volatile uint8_t queue_count = 0;
volatile state_t current_state = STATE_OFF;
volatile uint32_t button_press_time = 0;
volatile uint8_t button_released = 1;

// Button interrupt handler (EXTI0)
void EXTI0_IRQHandler(void) {
    uint32_t current_time = HAL_GetTick();

    if (EXTI->PR & EXTI_PR_PR0) {
        if (GPIOA->IDR & GPIO_PIN_0) {
            // Button released
            if (!button_released) {
                uint32_t press_duration = current_time - button_press_time;
                if (press_duration > 1000) {
                    // Long press (>1s)
                    if (queue_count < 10) {
                        event_queue[queue_tail] = EVENT_LONG_PRESS;
                        queue_tail = (queue_tail + 1) % 10;
                        queue_count++;
                    }
                } else if (press_duration > 50) {
                    // Short press (debounced)
                    if (queue_count < 10) {
                        event_queue[queue_tail] = EVENT_SHORT_PRESS;
                        queue_tail = (queue_tail + 1) % 10;
                        queue_count++;
                    }
                }
                button_released = 1;
            }
        } else {
            // Button pressed
            button_press_time = current_time;
            button_released = 0;
        }

        // Clear pending bit
        EXTI->PR = EXTI_PR_PR0;
    }
}

// Timer interrupt for blinking (TIM3)
void TIM3_IRQHandler(void) {
    if (TIM3->SR & TIM_SR_UIF) {
```

Interrupt Performance Exercises

Interrupt Performance Analysis

Analyzing Interrupt System Performance

Calculate interrupt frequency

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

Determine interrupt service time

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

Calculate system impact

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

Optimize interrupt handling

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

Interrupt System Analysis

A processor system running at 1 MHz receives data through a peripheral interface at a rate of 16 kbit/s. The peripheral can buffer 32 bits of data and signals the processor via an interrupt line when the buffer is ready to be read. If the data is not read before the next interrupt, it is lost.
The Interrupt Service Routine (ISR) requires 100 clock cycles on average, including call and return overhead. The system uses no other interrupts.

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

Solution:

1. **Calculate interrupt impact:**
- Interrupt frequency: f_{INT} = 16 kbit/s ÷ 32 bits = 500 Hz
- Interrupt service time: t_{ISR} = 100 cycles ÷ 1 MHz = 100 mus
- Impact = $f_{INT} \times t_{ISR} \times 100\%$ = 500 Hz × 100 mus × 100% = 5%

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

2. **Maximum data rate calculation:**
- For 100% CPU usage: 1 = $f_{INT} \times t_{ISR}$
- f_{INT} = 1 ÷ t_{ISR} = 1 ÷ 100 mus = 10,000 Hz
- Data rate = f_{INT} × 32 bits = 10,000 × 32 = 320 kbit/s

3. **Cause of occasional data loss:**
- At 90% CPU utilization, there should theoretically be enough time to process all interrupts
- However, the service time (t_{ISR}) is not constant but an average
- Some interrupt instances may take longer than the average 100 cycles
- When t_{ISR} varies, occasional peaks can exceed the time between interrupts
- The variation may come from:
  - Different code paths within the ISR depending on the data
  - Memory access times that vary due to cache effects
  - Context switching overhead that varies based on CPU state
  - Interrupt latency variations based on what instruction was executing
- When a single interrupt takes too long, the next interrupt arrives before processing is complete
- With high CPU utilization (90%), there's little margin for these variations
- Result: occasional buffer overflows and data loss

Interrupt Latency Optimization

Managing Interrupt Latency

Understand latency components

- **Hardware latency:**
  - Time for interrupt controller to recognize interrupt
  - Time to finish current instruction (multi-cycle instructions)
  - Time to push registers onto stack
- **Software latency:**
  - Interrupt disabled periods
  - Higher priority interrupt processing
  - Cache misses during ISR execution

Measure interrupt latency

- Use an oscilloscope to measure time between:
  - Interrupt trigger signal
  - GPIO pin toggle in ISR
- Alternatively, use a timer to capture timestamps:
  - Configure timer to capture on interrupt signal
  - Read timer value at ISR entry
  - Calculate difference
- Measure best-case, worst-case, and average latency

Optimize for consistent latency

- Minimize interrupt disable periods
  - Keep critical sections short
  - Use selective interrupt masking instead of global disable
- Optimize ISR code
  - Minimize stack usage to reduce push/pop operations
  - Keep ISRs short and efficient
  - Ensure frequently used data and code are in cache
- Use appropriate priorities
  - Assign higher priorities to time-critical interrupts
  - Group interrupts with similar timing requirements

Implement deterministic response

- For time-critical operations, consider:
  - Polling instead of interrupts if determinism is crucial
  - Hardware solutions (DMA, dedicated controllers)
  - Real-time operating system with predictable scheduling
- Move processing from ISRs to main loop
  - ISR only captures essential data and signals main loop
  - Use flags, semaphores, or message queues for communication

Interrupt Latency Optimization

A microcontroller-based system needs to sample an analog signal at precise 100 µs intervals. The current implementation uses Timer1 to generate interrupts every 100 µs, and the ISR reads the ADC. Measurements show that the interrupt latency varies between 5-20 µs, causing jitter in the sampling.
Design an improved solution to minimize sampling jitter, explaining your approach and implementation.

Solution:

The goal is to reduce sampling jitter by minimizing variations in the time between the timer interrupt and ADC sampling. Here's an improved solution:
Hardware-triggered ADC approach:
1. **Use timer to trigger ADC directly:**
- Configure Timer1 to generate a trigger signal every 100 µs
- Connect this trigger to the ADC hardware trigger input
- Configure the ADC to start conversion automatically on timer trigger
- Use DMA to transfer ADC results to memory without CPU intervention

2. **Implementation details:**

```
// Configure Timer1
void configure_timer(void) {
    // Enable clock for Timer1
    RCC->APB2ENR |= RCC_APB2ENR_TIM1EN;

    // Set prescaler and period for 100 microsecond interval
    TIM1->PSC = (SystemCoreClock / 1000000) - 1;  // 1 microsecond timer ticks
    TIM1->ARR = 100 - 1;  // 100 microsecond period

    // Configure timer to generate trigger output
    TIM1->CR2 &= ~TIM_CR2_MMS;
    TIM1->CR2 |= TIM_CR2_MMS_1;  // Update event as trigger output

    // Enable timer
    TIM1->CR1 |= TIM_CR1_CEN;
}

// Configure ADC with timer trigger
void configure_adc(void) {
    // Enable clocks for ADC and GPIO
    RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;

    // Configure GPIO pin as analog input
    GPIOA->MODER |= GPIO_MODER_MODER0;  // PA0 as analog

    // Configure ADC for timer trigger
    ADC1->CR2 |= ADC_CR2_EXTEN_0;        // Enable external trigger on rising edge
    ADC1->CR2 |= ADC_CR2_EXTSEL_3;       // Select Timer1 TRGO as trigger

    // Configure single conversion on channel 0
    ADC1->SQR3 = 0;  // Channel 0 as first conversion
    ADC1->SQR1 = 0;  // 1 conversion

    // Enable DMA for ADC
    ADC1->CR2 |= ADC_CR2_DMA;

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

// Configure DMA for ADC
void configure_dma(uint16_t* buffer, uint32_t buffer_size) {
    // Enable DMA clock
    RCC->AHB1ENR |= RCC_AHB1ENR_DMA2EN;

    // Configure DMA stream
    DMA2_Stream0->CR &= ~DMA_SxCR_EN;  // Disable DMA during config

    DMA2_Stream0->PAR = (uint32_t)&ADC1->DR;       // Source: ADC data register
    DMA2_Stream0->M0AR = (uint32_t)buffer;         // Destination: buffer
    DMA2_Stream0->NDTR = buffer_size;              // Number of data items
```

Designing Efficient Interrupt-Driven Systems

Choose between polling and interrupts

- **Use interrupts when:**
  - Events occur infrequently or at unpredictable times
  - System needs to respond to external events quickly
  - Power efficiency is important (can sleep between events)
  - Multiple event sources need to be monitored simultaneously
- **Use polling when:**
  - Events occur at very high frequency
  - Absolute determinism is required
  - Interrupt overhead would be excessive
  - Extremely low latency is required

Design efficient ISRs

- Keep ISRs as short as possible
  - Only handle time-critical operations in the ISR
  - Defer processing to main loop using flags or queues
  - Avoid complex calculations or I/O operations
- Manage shared resources
  - Use volatile for variables shared between ISR and main code
  - Consider atomic operations for simple updates
  - Implement proper synchronization for complex data structures
- Avoid nesting or recursion in ISRs

Implement event queues

- Use queue to transfer events from ISRs to main loop
  - ISR detects event and enqueues it
  - Main loop dequeues and processes events
- Design queue to be interrupt-safe
  - Use atomic operations if available
  - Consider implementing lock-free algorithms
  - Size queue appropriately to avoid overflow
- Process events in order of importance or arrival

Implement event-driven architecture

- Structure code as state machines responding to events
- Separate event detection (ISRs) from processing (handlers)
- Design for different operating modes
  - Normal operation
  - Power-saving modes
  - Error handling and recovery
- Consider using an RTOS for complex systems

Interrupt-Driven UART Communication System

Design an interrupt-driven communication system with the following requirements:

- UART interface operating at 115200 baud, 8N1 format
- Must handle both transmit and receive operations
- System should never lose incoming data
- Main application should not be blocked by I/O operations
- CPU runs at 72 MHz and has other tasks to perform

Solution:

The system will use an interrupt-driven approach with queues to handle UART communication efficiently:

1. **System architecture:**

- Receive interrupt (RXNE) to handle incoming data
- Transmit interrupt (TXE) to send outgoing data
- Circular buffers for both TX and RX data
- Non-blocking API for the main application

2. **Implementation:**

```c
#include <stdint.h>
#include <stdbool.h>

// Buffer size must be power of 2 for efficient wrap-around
#define UART_BUFFER_SIZE 256
#define UART_BUFFER_MASK (UART_BUFFER_SIZE - 1)

// Circular buffer structure
typedef struct {
    volatile uint8_t data[UART_BUFFER_SIZE];
    volatile uint16_t head;
    volatile uint16_t tail;
} circular_buffer_t;

// Global buffers
static circular_buffer_t rx_buffer = {0};
static circular_buffer_t tx_buffer = {0};
static volatile bool tx_busy = false;

// Function prototypes
void uart_init(uint32_t baudrate);
bool uart_send_byte(uint8_t data);
bool uart_send_data(const uint8_t* data, uint16_t length);
bool uart_read_byte(uint8_t* data);
uint16_t uart_available(void);
void uart_flush(void);

// External hardware functions
extern void uart_hw_init(uint32_t baudrate);
extern void uart_hw_send_byte(uint8_t data);
extern uint8_t uart_hw_read_byte(void);
extern bool uart_hw_tx_empty(void);
extern bool uart_hw_rx_ready(void);
extern void uart_hw_enable_tx_interrupt(bool enable);
extern void uart_hw_enable_rx_interrupt(bool enable);

// Buffer operations
static bool buffer_is_full(const circular_buffer_t* buffer) {
    return ((buffer->head + 1) & UART_BUFFER_MASK) == buffer->tail;
}

static bool buffer_is_empty(const circular_buffer_t* buffer) {
    return buffer->head == buffer->tail;
}

static bool buffer_push(circular_buffer_t* buffer, uint8_t data) {
    uint16_t next_head = (buffer->head + 1) & UART_BUFFER_MASK;

    if (next_head == buffer->tail) {
        return false; // Buffer full
    }

```