Firmware Coding

By Shimi Cohen

Chapter 1: Getting Started

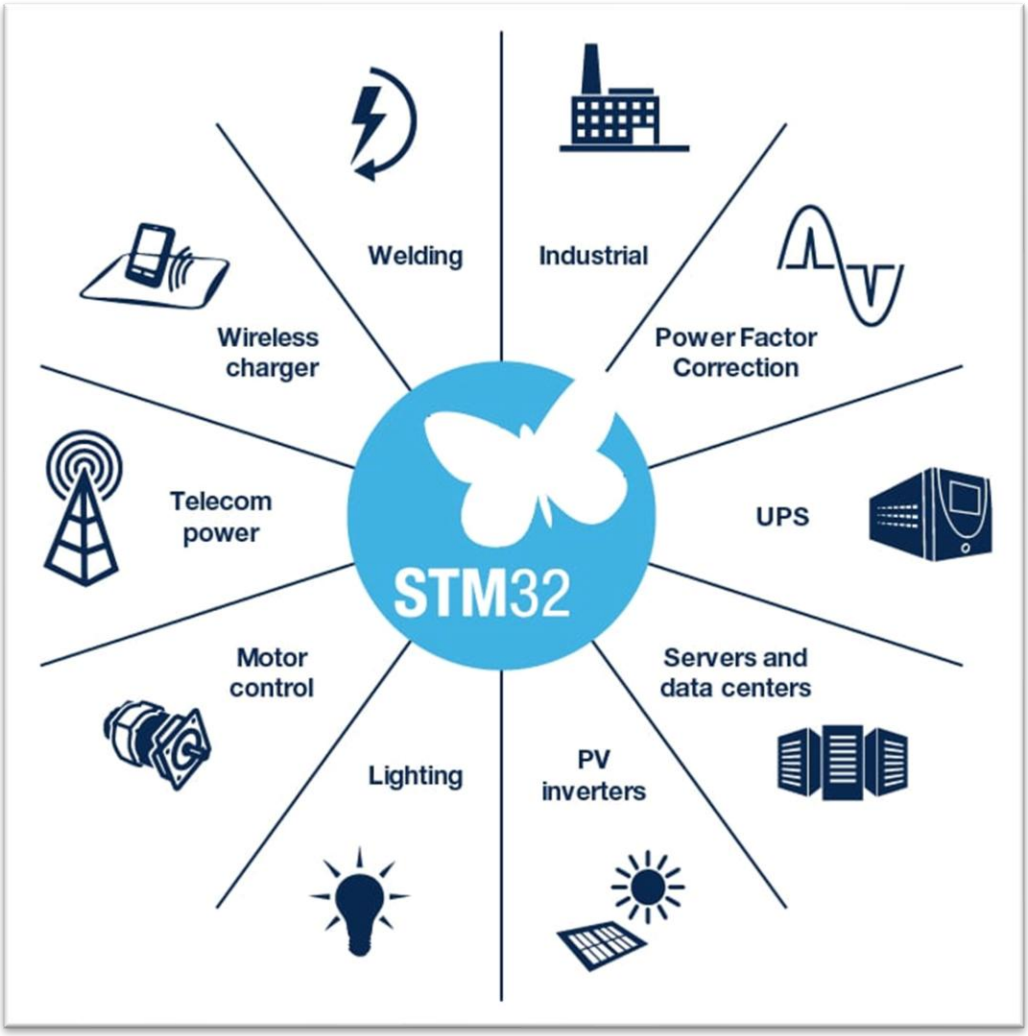
PROLOGUE: STM32 FAMILY

STM32 MCU FAMILY OVERVIEW

STM32 microcontrollers represent one of the most comprehensive and widely adopted MCU families in embedded systems development. Built around ARM Cortex-M cores, STM32 devices offer scalable performance, extensive peripheral integration, and robust development ecosystem support.

STM32 FAMILY CHARACTERISTICS:

- ARM Cortex-M core architecture (M0, M0+, M3, M4, M7)
- Comprehensive peripheral set integration
- Multiple memory configurations
- Advanced power management features
- Extensive development tool support
- Wide range of package options



STM32 PRODUCT LINES

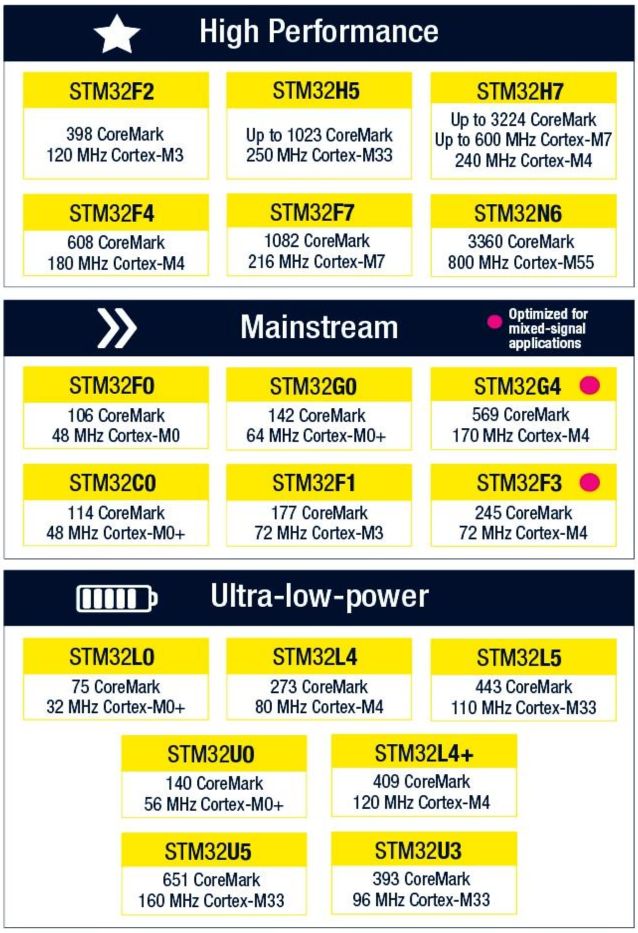
The STM32 family encompasses multiple product lines, each optimized for specific application requirements and performance levels.

MAIN STM32 SERIES:

- STM32F0/G0: Entry-level Cortex-M0/M0+ based
- STM32F1/F3: Mainstream Cortex-M3/M4 solutions
- STM32F4/F7: High-performance Cortex-M4/M7 series
- STM32L: Ultra-low-power optimized devices
- STM32H7: High-performance dual-core solutions
- STM32WB/WL: Wireless connectivity integrated

PERFORMANCE SCALING:

Series	Core	Max Freq	Key Features
STM32F0	Cortex-M0	48MHz	Cost-effective, basic peripherals
STM32F4	Cortex-M4	180MHz	DSP, FPU, advanced timers
STM32H7	Cortex-M7	480MHz	Dual-core, high-speed connectivity



STM32**CUBEMX** TOOL

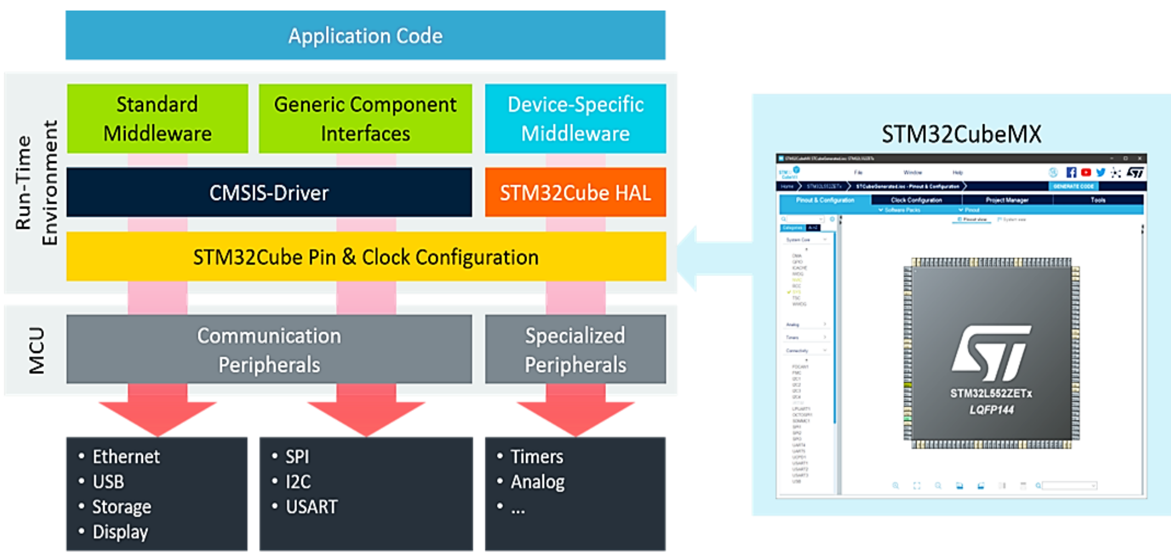
STM32CubeMX serves as the primary configuration and initialization code generation tool for STM32 microcontrollers. It provides graphical peripheral configuration and automatic code generation capabilities.

CUBEMX CORE FEATURES:

- Graphical Pin Configuration: Interactive pin assignment
- Peripheral Configuration: Parameter setting with validation
- Clock Tree Configuration: System and peripheral clock setup
- Code Generation: Automatic initialization code creation
- Middleware Integration: USB, TCP/IP, file system support
- Power Consumption Analysis: Current consumption estimation

CUBEMX WORKFLOW:

1. Device Selection: Choose target STM32 device
2. Pin Configuration: Assign peripheral functions to pins
3. Clock Configuration: Set up system and peripheral clocks
4. Peripheral Configuration: Configure peripheral parameters
5. Middleware Configuration: Enable and configure middleware
6. Code Generation: Generate initialization code
7. Project Export: Export to supported IDEs



STM32CUBEIDE DEV ENVIRONMENT

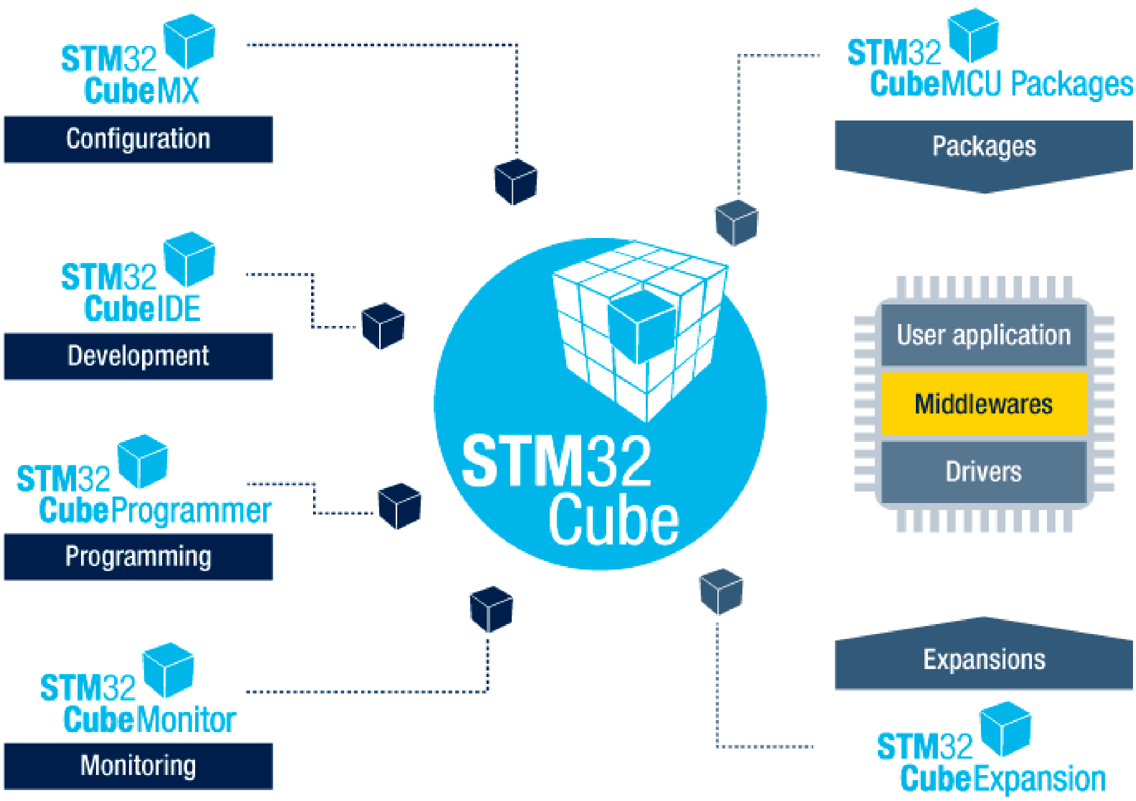
STM32CubeIDE provides a comprehensive integrated development environment specifically designed for STM32 microcontroller development. Built on Eclipse framework, it integrates all necessary tools for firmware development.

CUBEIDE KEY COMPONENTS:

- Code Editor: Syntax highlighting, auto-completion, refactoring
- Compiler Toolchain: GCC-based ARM compiler integration
- Debugger: GDB-based debugging with breakpoints and watch windows
- Project Manager: Template-based project creation
- Version Control: Git integration for source management
- Performance Analysis: CPU usage and memory profiling

DEVELOPMENT TOOL INTEGRATION:

- STM32CubeMX Integration: Direct project import and configuration
- STM32CubeProgrammer: Flash programming and device management
- STM32CubeMonitor: Real-time variable monitoring
- Static Code Analysis: Code quality and security checking



STM32 HAL ECOSYSTEM

The STM32 Hardware Abstraction Layer (HAL) library provides standardized APIs across the entire STM32 family.

HAL LIBRARY STRUCTURE:

- Core HAL: Basic system functions and common definitions
- Peripheral HAL: Standardized peripheral driver APIs
- Low-Level (LL) APIs: Register-level access for optimization
- Board Support Packages (BSP): Board-specific implementations
- Middleware: USB, TCP/IP, file system, graphics libraries

HAL BENEFITS FOR FIRMWARE DEVELOPMENT:

- Consistent API: Uniform interface across STM32 devices
- Code Portability: Easy migration between STM32 families
- Reduced Development Time: Pre-tested, validated drivers
- Comprehensive Documentation: Detailed API reference
- Community Support: Extensive examples and forums

GETTING STARTED WORKFLOW

This guide focuses on practical firmware development using STM32CubeMX and STM32CubeIDE tools.

DEVELOPMENT PROCESS:

1. Project Setup: Create new STM32CubeMX project
2. Hardware Configuration: Configure pins, clocks, and peripherals
3. Code Generation: Generate initialization code with CubeMX
4. IDE Import: Import generated project into STM32CubeIDE
5. Application Development: Implement firmware logic
6. Build and Debug: Compile, program, and debug firmware

ESSENTIAL PREREQUISITES:

- Hardware: STM32 development board (Nucleo, Discovery, custom)
- Software: STM32CubeMX, STM32CubeIDE, device drivers
- Debug Interface: ST-Link programmer/debugger
- Documentation: Reference manual, datasheet, HAL documentation

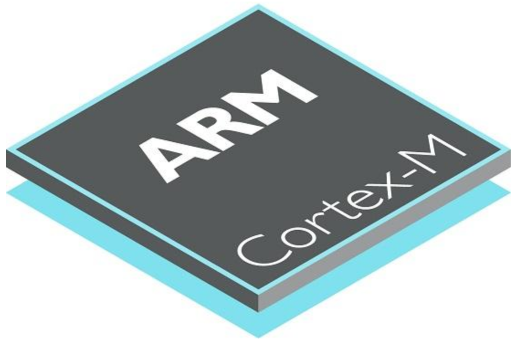
1: INTRODUCTION

1.1 FIRMWARE FUNDAMENTALS

FW represents the lowest-level software layer that directly interfaces with hardware components. Unlike application software that runs on top of operating systems, FW operates without abstraction layers.

KEY CHARACTERISTICS:

- Non-volatile storage in flash memory
- Hardware-specific implementation
- Real-time execution requirements
- Direct register manipulation

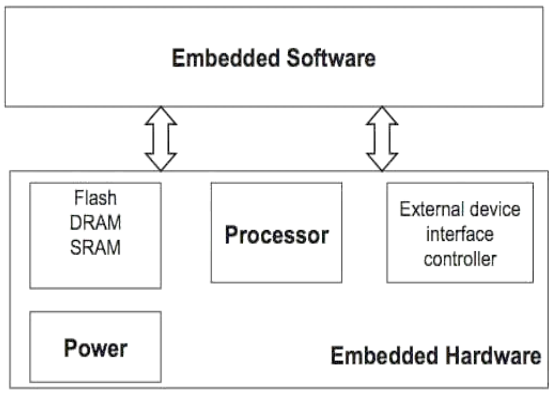


1.2 EMBEDDED SYSTEMS

FW serves as the critical bridge between hardware capabilities and system functionality. It manages HW and handles low-level communication protocols.

PRIMARY FUNCTIONS:

- HW initialization and configuration
- Interrupt service routine (ISR)
- Peripheral device control
- System timing and scheduling
- Power management

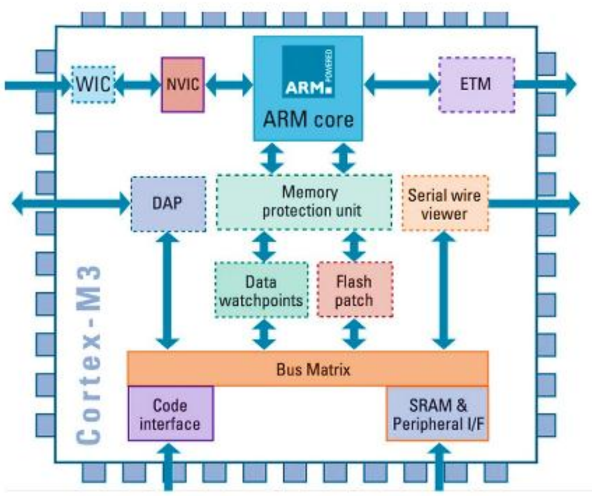


1.3 MICRO-CONTROLLER-UNIT

MCU integrates a processor core, memory, and peripherals on a single chip.

ESSENTIAL MCU COMPONENTS:

- Central Processing Unit (CPU)
- Program memory (Flash)
- Data memory (RAM/SRAM)
- Peripheral interfaces
- Clock generation and management
- Power management unit



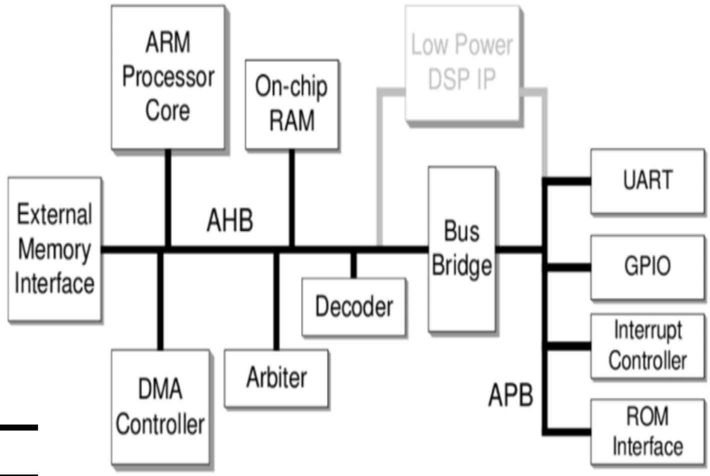
2: MCU ARCHITECTURE

2.1 FW INTERACTION

FW communicates with MCU HW through **memory-mapped registers** and DMA.

INTERACTION METHODS:

- Memory-mapped I/O operations
- Direct register manipulation
- DMA controller programming
- Interrupt vector handling



Element	Access	FW Impact
GPIO Ports	Memory Reg.	Direct bit
Timers	Control Reg.	Interrupt-driven
COMM	Buffer Reg.	Protocol
ADC/DAC	Data Reg.	signal processing

2.2 CORE COMPONENTS

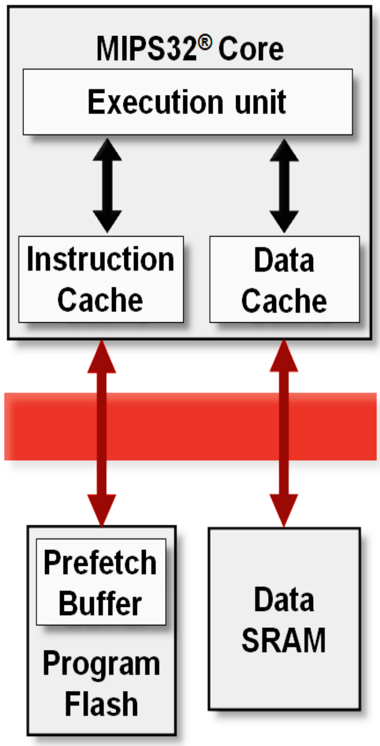
The MCU architecture directly impacts firmware design.

CPU ARCHITECTURE CONSIDERATIONS:

- Instruction pipeline depth
- Register file organization
- Execution unit capabilities
- Cache memory availability

MEMORY HIERARCHY:

- Flash memory for program storage
- SRAM for runtime data
- Register files for immediate access
- Peripheral registers for hardware control



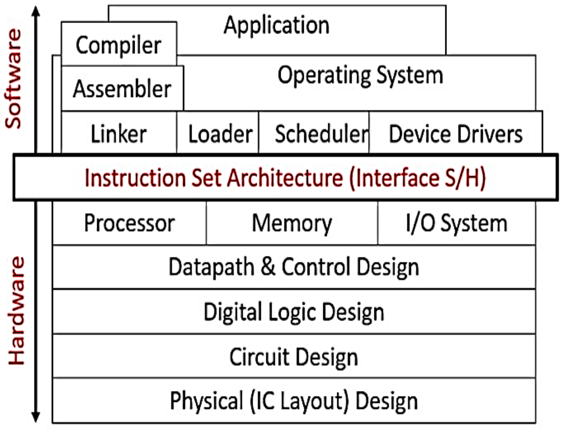
3: INSTRUCTION SET ARCHITECTURE

3.1 FUNDAMENTALS

The Instruction Set Architecture defines the interface between software and hardware. ISA determines available instructions, addressing modes, and data types that firmware can utilize.

ISA COMPONENTS:

- Instruction formats and encoding
- Addressing modes and operand types
- Register organization and usage
- Exception and interrupt handling

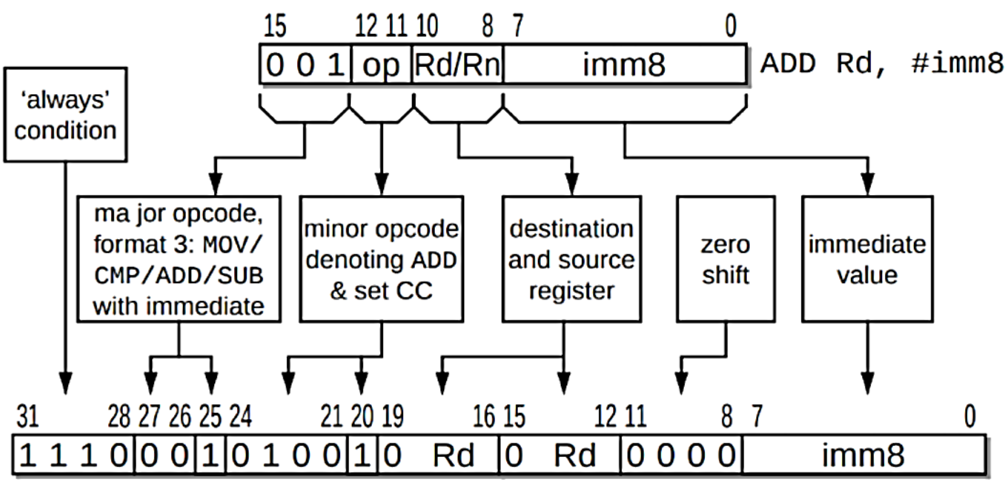


3.2 ISA EFFECT ON FIRMWARE

ISA characteristics directly influence firmware efficiency, code density, and execution performance. Understanding ISA capabilities enables optimal firmware implementation.

ARM CORTEX-M ISA FEATURES:

- Thumb-2 instruction set
- 16-bit and 32-bit instruction mixing
- Conditional execution capabilities
- Hardware multiply/divide support



4: MEMORY

4.1 MEMORY MAP

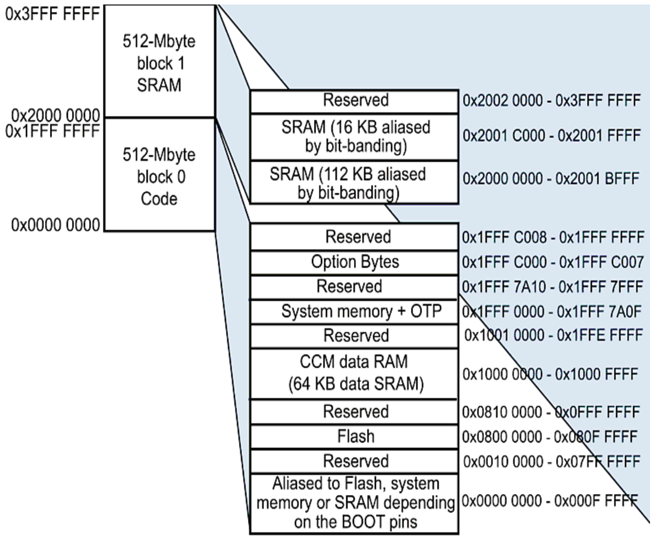
Memory organization determines how FW utilizes available storage.

FLASH MEMORY CHARACTERISTICS:

- Non-volatile program storage
- Sector-based erase operations
- Write Endurance Limitations
- Boot vector storage

RAM MEMORY USAGE:

- Runtime variable storage
- Stack and heap management
- Interrupt Service Routine Context
- DMA buffer allocation



4.2 FW VIEW OF MEMORY LAYOUT

Firmware must manage memory allocation, protection, and optimization. Memory layout decisions impact system performance and maintainability.

TYPICAL MEMORY SECTIONS:

- Vector table (interrupt handlers)
- Program code (.text section)
- Initialized data (.data section)
- Uninitialized data (.bss section)
- Stack and heap regions

Memory Region	Purpose	Size Considerations
Flash	Program code, constants	Code complexity, feature set
RAM	Variables, buffers	Real-time requirements
Stack	Function calls, interrupts	Nesting depth, recursion
Heap	Dynamic allocation	Memory fragmentation

5: BOOT PROCESS

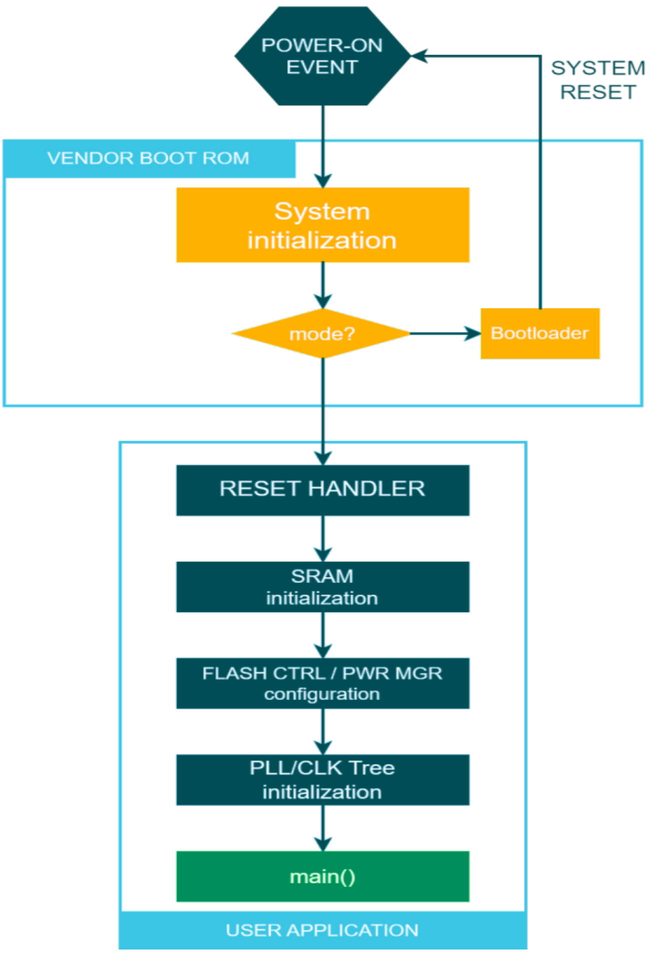
5.1 POWER-UP SEQUENCE

Boot process establishes the foundation for all subsequent firmware operations.

BOOT SEQUENCE STEPS:

1. Power-on reset generation
2. Boot vector fetch from flash
3. Stack pointer initialization
4. System clock configuration
5. Peripheral initialization
6. Application entry point

Boot mode selection pins		Boot mode	Aliasing
BOOT1	BOOT0		
x	0	Main Flash memory	Main Flash memory is selected as boot space
0	1	System memory	System memory is selected as boot space
1	1	Embedded SRAM	Embedded SRAM is selected as boot space



5.2 FW DURING STARTUP

Firmware must configure all system components before entering normal operation. Initialization order affects system reliability and performance.

CRITICAL INITIALIZATION TASKS:

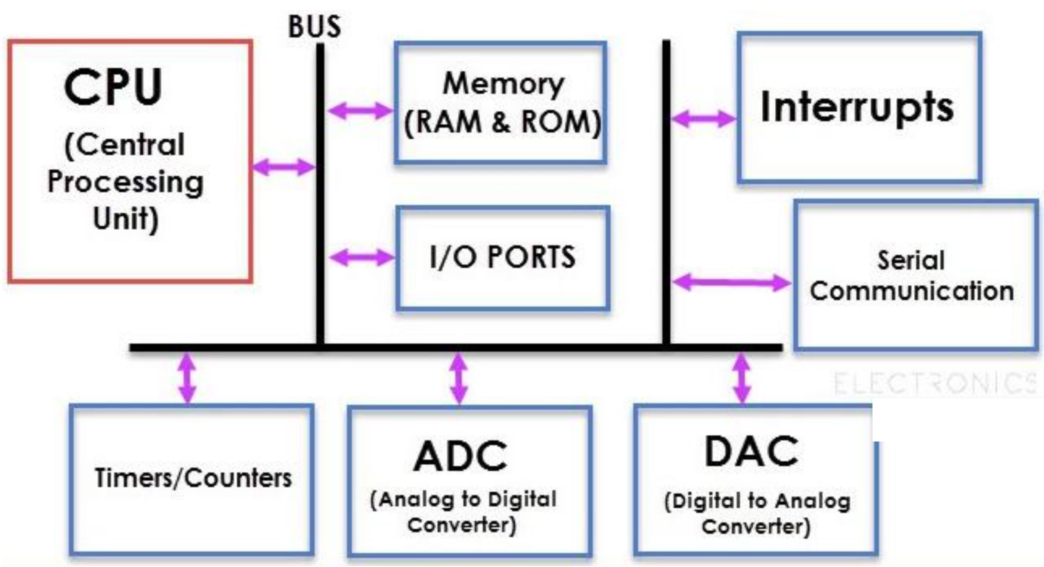
- Clock source selection
- PLL configuration
- Memory controller setup
- Peripheral clock enabling
- GPIO pin configuration
- Interrupt controller setup

STM32F4 BOOT PROCESS:

```
// Reset handler - first function called after boot
void Reset_Handler(void)
 {
    // Configure system clock to 168MHz
    SystemInit();

    // Copy initialized data from flash to RAM
    __libc_init_array();

    // Jump to main application
    main();
}
```



6: GENERAL PURPOSE IN/OUT

6.1 GPIO FUNDAMENTALS

General Purpose Input/Output provides firmware with direct digital signal control. GPIO pins serve as the primary interface between MCU and external components.

GPIO CONFIGURATION OPTIONS:

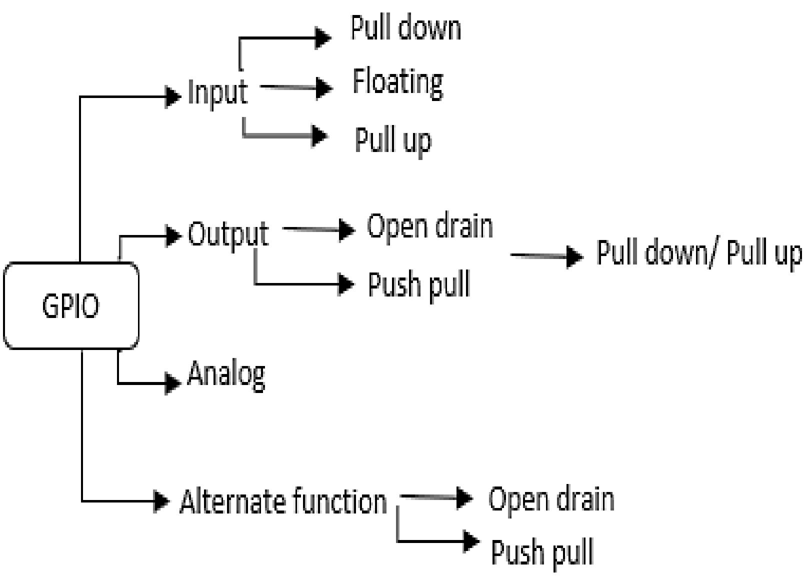
- Input/output direction
- Pull-up/pull-down resistors
- Output drive strength
- Slew rate control
- Alternate function selection

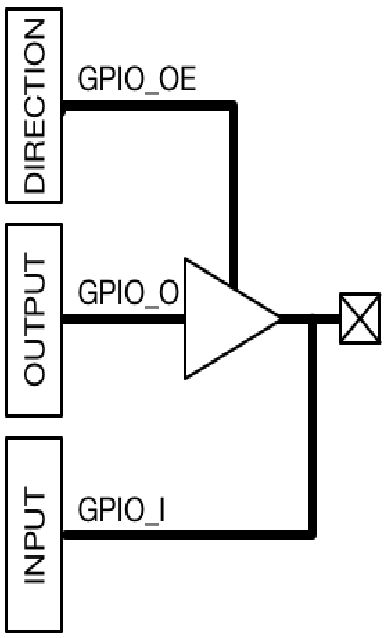
6.2 TOGGLE/READING PINS

GPIO operations form the foundation of embedded system interaction.

BASIC GPIO OPERATIONS:

- Pin state reading
- Output level setting
- Interrupt generation
- Debouncing implementation





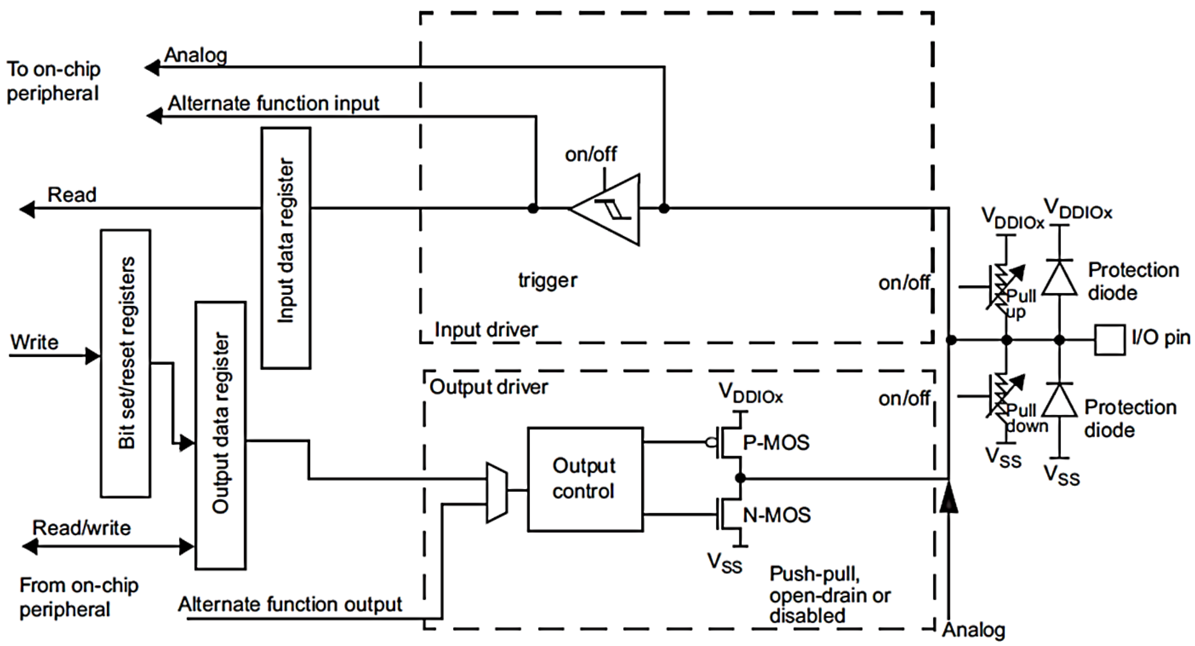
EXAMPLE: LED CONTROL WITH BUTTON INPUT

```
// Configure GPIO for LED output and button input
void GPIO_Init(void)
{
    // Enable GPIO clock
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;

    // Configure PA5 as output (LED)
    GPIOA->MODER |= (1 << 10);

    // Configure PA0 as input (Button)
    GPIOA->MODER &= ~(3 << 0);
    GPIOA->PUPDR |= (1 << 0); // Pull-up
}
// Toggle LED based on button state
void Process_Button(void)
{
    static uint32_t last_press = 0;
    uint32_t current_time = HAL_GetTick();

    if (!(GPIOA->IDR & 1) && (current_time - last_press > 50)) {
        GPIOA->ODR ^= (1 << 5); // Toggle LED
        last_press = current_time;
    }
}
```



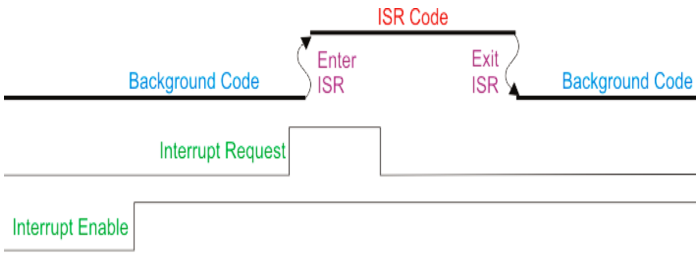
7: INTERRUPTS

7.1 INTERRUPTS FUNDAMENTALS

Interrupts provide asynchronous event notification to firmware. They enable efficient system resource utilization and real-time response capabilities.

INTERRUPT CHARACTERISTICS:

- Asynchronous execution
- Priority-based handling
- Context switching overhead
- Nested interrupt support

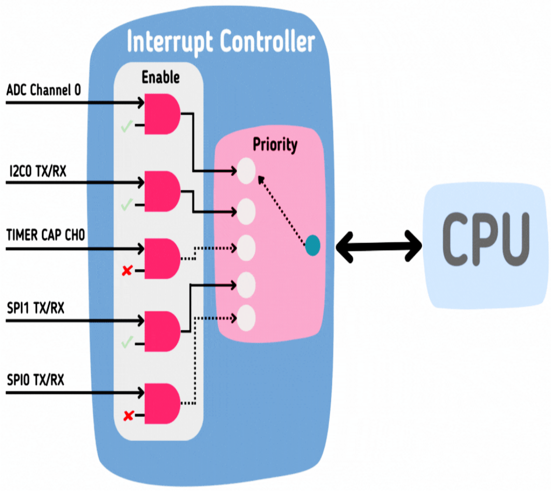


7.2 FW EVENT RESPONSE

Firmware must implement interrupt service routines that handle events efficiently while maintaining system stability.

INTERRUPT HANDLING PRINCIPLES:

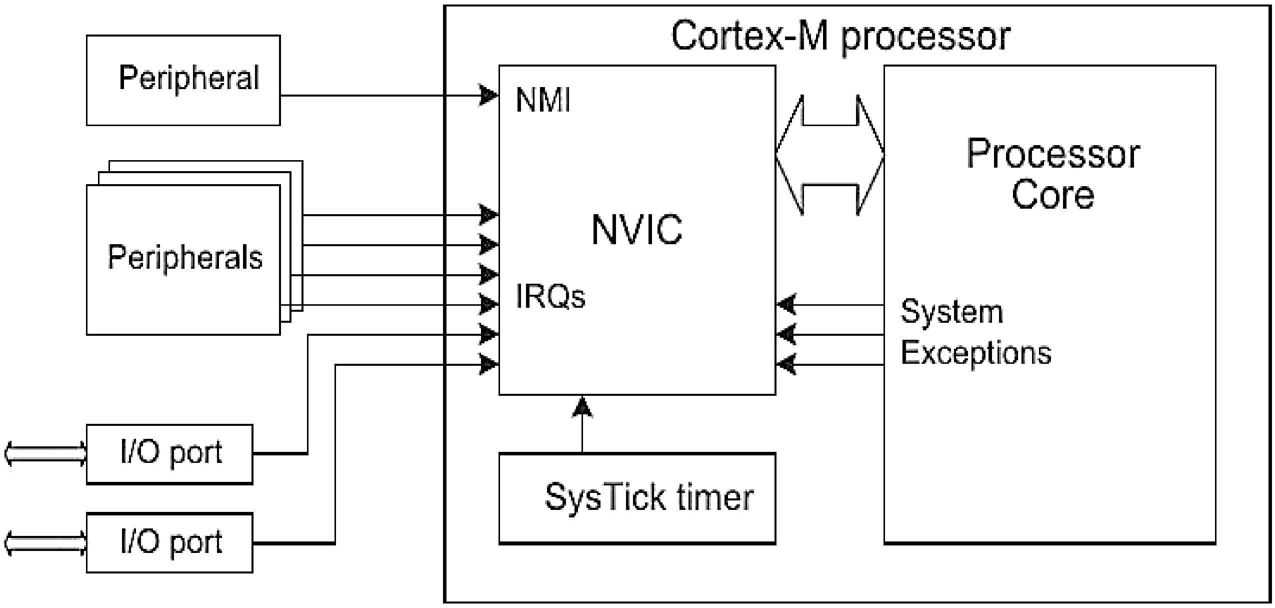
- Minimal ISR execution time
- Atomic operations for shared data
- Priority assignment strategy
- Interrupt latency optimization



INTERRUPT TYPES AND SOURCES:

- External (GPIO, external devices)
- Timer (periodic, overflow)
- Communication (UART, SPI, I2C)
- System (watchdog, power management)

Interrupt Source	Typical Use Case	Firmware Response
External GPIO	Button press, sensor trigger	Event flag setting
Timer Overflow	Periodic tasks	State machine update
UART Reception	Data communication	Buffer management
ADC Conversion	Analog measurement	Data processing



EXAMPLE: TIMER-BASED TASK SCHEDULER

```
// Timer interrupt for 1ms system tick
void TIM2_IRQHandler(void) {
    if (TIM2->SR & TIM_SR_UIF) {
        TIM2->SR &= ~TIM_SR_UIF; // Clear interrupt flag
        system_tick_counter++;

        // Schedule periodic tasks
        if (system_tick_counter % 10 == 0) {
            task_10ms_flag = 1;
        }
                if (system_tick_counter % 100 == 0) {
            task_100ms_flag = 1;
        }
    }
}
```

8: TIMERS

8.1 TYPES OF TIMERS

Timer peripherals provide accurate timing references for FW operations.

TIMER CLASSIFICATIONS:

- Basic timers (simple counting)
- General-purpose timers (PWM, capture/compare)
- Advanced timers (motor control, dead-time)
- Watchdog timers (system monitoring)

Timer type	Timer	Counter resolution	Counter type	Prescaler factor	DMA request generation	Capture/ compare channels	Complementary outputs
Advanced control	TIM1, TIM8	16-bit	Up, down, Up/down	Any integer between 1 and 65536	Yes	4	3
General-purpose	TIM2, TIM5	32-bit	Up, down, Up/down	Any integer between 1 and 65536	Yes	4	No
General-purpose	TIM3, TIM4	16-bit	Up, down, Up/down	Any integer between 1 and 65536	Yes	4	No
General-purpose	TIM15	16-bit	Up	Any integer between 1 and 65536	Yes	2	1
General-purpose	TIM16, TIM17	16-bit	Up	Any integer between 1 and 65536	Yes	1	1
Basic	TIM6, TIM7	16-bit	Up	Any integer between 1 and 65536	Yes	0	No

Timer Type	Key Features	Firmware Usage
Basic Timer	Simple counting	System tick, delays
General Timer	PWM, capture/compare	Motor control, measurement
Advanced Timer	Dead-time, complementary outputs	Power electronics
Watchdog Timer	Independent clock	System monitoring

8.2 TIMER CONFIGURATION

FW relies on timers for scheduling, timeout, and real-time system operation.

TIMER CONFIGURATION PARAMETERS:

- Prescaler value (clock division)
- Counter period (overflow value)
- Count direction (up/down)
- Trigger sources (internal/external)

EXAMPLE: PWM MOTOR CONTROL

```
// Configure Timer for PWM motor control
void PWM_Init(void) {
    // Timer configuration for 20kHz PWM
    TIM3->PSC = 0;          // No prescaler
    TIM3->ARR = 4199;       // 20kHz with 84MHz clock
    TIM3->CCR1 = 0;         // Initial duty cycle 0%

    // Configure channel 1 for PWM mode
    TIM3->CCMR1 |= TIM_CCMR1_OC1M_2 | TIM_CCMR1_OC1M_1;
    TIM3->CCER |= TIM_CCER_CC1E;

    // Enable timer
    TIM3->CR1 |= TIM_CR1_CEN;
}

// Update motor speed (0-100%)
void Set_Motor_Speed(uint8_t speed) {
    uint32_t duty_cycle = (speed * 4199) / 100;
    TIM3->CCR1 = duty_cycle;
}
```

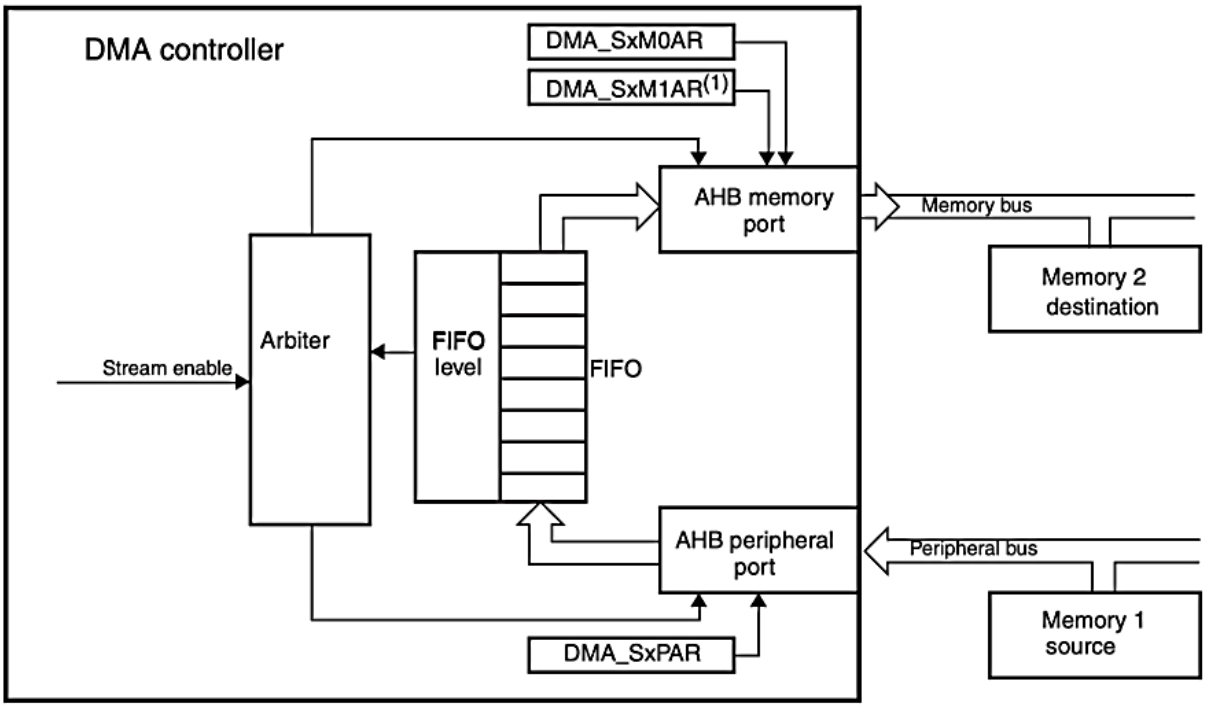
9: DIRECT MEMORY ACCESS

9.1 WHAT IS DMA?

Direct Memory Access enables data transfer between memory and peripherals without CPU intervention. DMA reduces CPU load and improves performance.

DMA CAPABILITIES:

- Memory-to-memory transfers
- Peripheral-to-memory transfers
- Memory-to-peripheral transfers
- Circular buffer management



9.2 DMA FOR EFFICIENT DATA

Firmware configures DMA controllers to handle repetitive data transfers automatically. This approach frees CPU resources for critical tasks.

DMA CONFIGURATION ELEMENTS:

- Source and destination addresses
- Transfer size and data width
- Priority levels
- Transfer completion callbacks

DMA USE CASES:

- ADC data collection
- UART data transmission
- SPI communication
- Memory initialization

EXAMPLE: ADC DATA COLLECTION WITH DMA

```c
// Configure DMA for continuous ADC data collection
void DMA_ADC_Init(void)
{
    // DMA configuration for ADC1
    DMA2_Stream0->CR = 0; // Disable DMA
    while (DMA2_Stream0->CR & DMA_SxCR_EN); // Wait for disable

    DMA2_Stream0->PAR = (uint32_t)&ADC1->DR;
    DMA2_Stream0->M0AR = (uint32_t)adc_buffer;
    DMA2_Stream0->NDTR = ADC_BUFFER_SIZE;

    // Configure DMA: circular mode, 16-bit, memory increment
    DMA2_Stream0->CR = DMA_SxCR_CIRC | DMA_SxCR_MINC |
                       DMA_SxCR_MSIZE_0 | DMA_SxCR_PSIZE_0;

    // Enable DMA
    DMA2_Stream0->CR |= DMA_SxCR_EN;
}
```

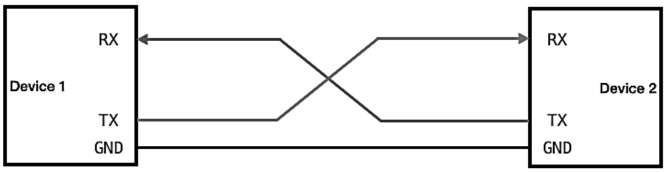
10: PERIPHERAL INTERFACES

10.1 UART / SPI / I2C / CAN

Communication peripherals enable firmware to interface with external devices and systems. Each protocol serves specific communication requirements.

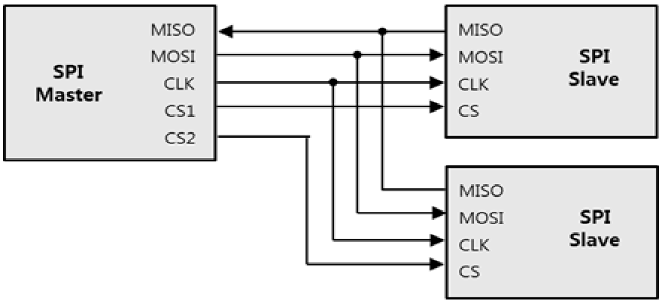
UART (UNIVERSAL ASYNCHRONOUS RECEIVER/TRANSMITTER):

- Point-to-point communication
- Asynchronous data transfer
- Configurable baud rates
- Error detection capabilities



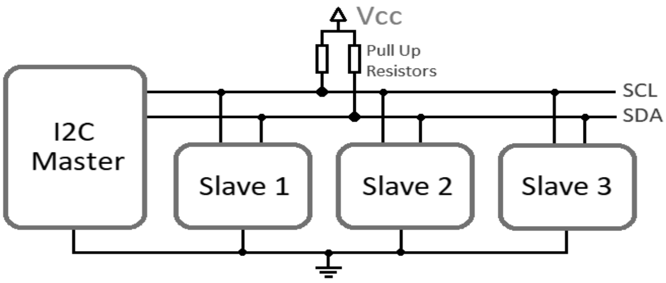
SPI (SERIAL PERIPHERAL INTERFACE):

- Master-slave communication
- Synchronous data transfer
- Full-duplex operation
- Multiple slave support



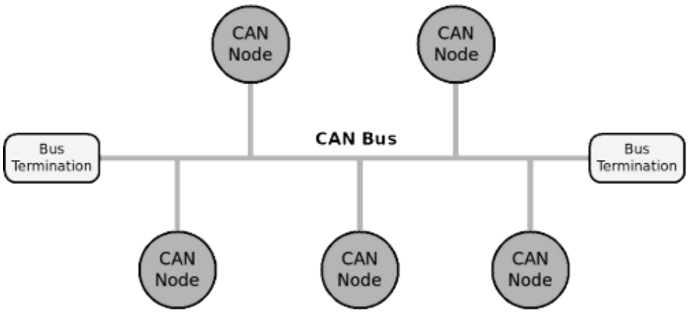
I2C (INTER-INTEGRATED CIRCUIT):

- Multi-master capability
- Two-wire interface
- Address-based COMM
- Built-in arbitration



CAN (CONTROLLER AREA NETWORK):

- Multi-master bus system
- Message-based COMM
- Built-in error detection
- Priority-based arbitration



10.2 FW ROLE IN COMMUNICATION

Firmware implements communication protocols, manages data buffers, and handles error conditions. Protocol implementation requires precise timing and state management.

COMMUNICATION MANAGEMENT TASKS:

- Protocol state machine implementation
- Buffer management and flow control
- Error detection and recovery
- Timing constraint adherence

Protocol	Key Features	Firmware Complexity	Typical Applications
UART	Simple, point-to-point	Low	Debug, GPS, Bluetooth
SPI	High speed, synchronous	Medium	Sensors, displays, memory
I2C	Multi-device, two-wire	Medium	Sensors, EEPROMs, RTCs
CAN	Robust, automotive	High	Vehicle networks, industrial

EXAMPLE: I2C SENSOR COMMUNICATION

```
// Read temperature from I2C sensor
HAL_StatusTypeDef Read_Temperature_Sensor(float *temperature) {
    uint8_t data[2];
    HAL_StatusTypeDef status;

    // Send measurement command
    uint8_t cmd = 0xE3; // Trigger temperature measurement
    status = HAL_I2C_Master_Transmit(&hi2c1, SENSOR_ADDRESS, &cmd, 1, 100);
    if (status != HAL_OK) return status;

    // Read measurement result
    status = HAL_I2C_Master_Receive(&hi2c1, SENSOR_ADDRESS, data, 2, 100);
    if (status != HAL_OK) return status;

    // Convert raw data to temperature
    uint16_t raw_temp = (data[0] << 8) | data[1];
    *temperature = -46.85 + 175.72 * (raw_temp / 65536.0);

    return HAL_OK;
}
```

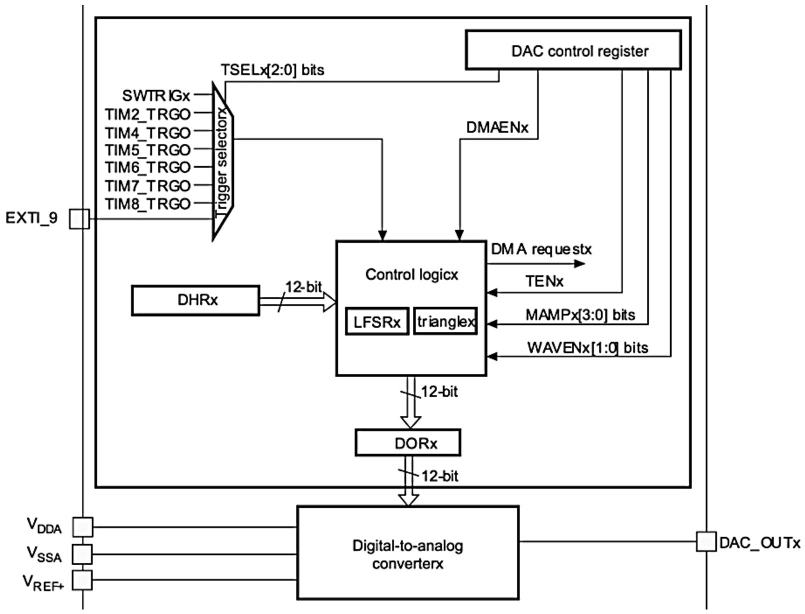
11: ANALOG FEATURES

11.1 ADC & DAC

Analog-to-Digital Converters (ADC) and Digital-to-Analog Converters (DAC) bridge the gap between analog real-world signals and digital firmware processing.

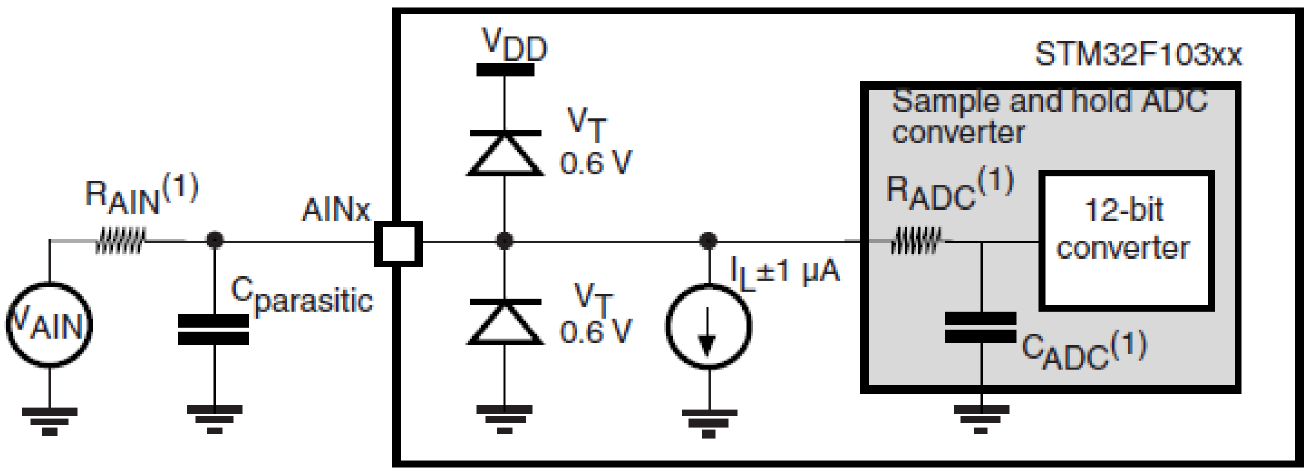
DAC CHARACTERISTICS:

- Output resolution
- Settling time
- Output drive capability
- Reference voltage options
- Waveform generation features



ADC CHARACTERISTICS:

- Resolution (8, 10, 12, 16 bits)
- Sampling rate capabilities
- Input voltage range
- Reference voltage sources
- Conversion modes



11.2 ANALOG PERIPHERALS

Firmware configures analog peripherals, manages conversion processes, and processes analog data. Proper analog handling ensures measurement accuracy and system performance.

ADC CONFIGURATION PARAMETERS:

- Channel selection and sequencing
- Sampling time settings
- Trigger sources
- Data alignment options
- Interrupt and DMA integration

ADC OPERATION MODES:

- Single conversion mode
- Continuous conversion mode
- Scan mode (multiple channels)
- Injected conversion mode

ADC Mode	Use Case	Firmware Considerations
Single	Occasional measurements	Polling or interrupt-based
Continuous	Regular monitoring	DMA for data handling
Scan	Multiple sensors	Channel management
Injected	High-priority samples	Interrupt priority setup

EXAMPLE: BATTERY VOLTAGE MONITORING

```
/// Configure ADC for battery voltage monitoring
void Battery_Monitor_Init(void) {
    // Enable ADC clock
    RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;

    // Configure ADC
    ADC1->CR2 = 0;
    ADC1->CR1 = 0;
    ADC1->SQR1 = 0; // 1 conversion
    ADC1->SQR3 = 1; // Channel 1
    ADC1->SMPR2 = ADC_SMPR2_SMP1_2; // 84 cycles sampling

    // Enable ADC
    ADC1->CR2 |= ADC_CR2_ADON;
}
// Read battery voltage with averaging
float Get_Battery_Voltage(void) {
    uint32_t adc_sum = 0;
    const uint8_t samples = 16;

    for (uint8_t i = 0; i < samples; i++) {
        // Start conversion
        ADC1->CR2 |= ADC_CR2_SWSTART;

        // Wait for conversion complete
        while (!(ADC1->SR & ADC_SR_EOC));

        adc_sum += ADC1->DR;
    }
        // Calculate average and convert to voltage
    uint16_t adc_avg = adc_sum / samples;
    float voltage = (adc_avg * 3.3f) / 4095.0f;

    // Apply voltage divider correction (R1=10k, R2=10k)
    return voltage * 2.0f;
}
```

12: FW LAYERING

12.1 HARDWARE ABSTRACTION LAYER (HAL)

The Hardware Abstraction Layer provides a standardized interface between low-level hardware and high-level application code. HAL enables firmware portability and maintainability.

HAL BENEFITS:

- Hardware independence
- Code reusability
- Faster development cycles
- Simplified debugging
- Consistent API across devices

HAL STRUCTURE:

- Hardware register definitions
- Peripheral initialization functions
- Data transfer APIs
- Error handling mechanisms
- Configuration structures

12.2 MODULAR FIRMWARE DESIGN

Modular FW design separates hardware-specific code from application logic.

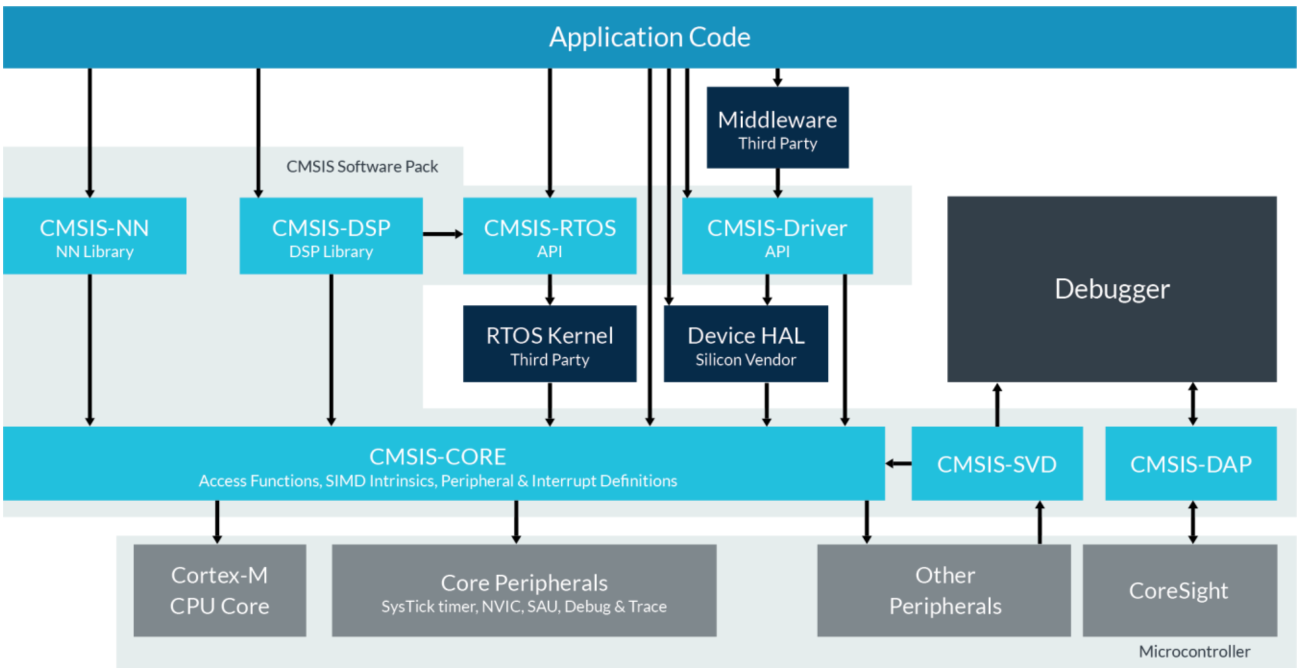
DRIVER ARCHITECTURE LAYERS:

- Board Support Package (BSP)
- Hardware Abstraction Layer (HAL)
- Device drivers
- Middleware
- Application layer

DESIGN PRINCIPLES:

- Separation of concerns
- Well-defined interfaces
- Minimal coupling
- Maximum cohesion
- Error propagation

Layer	Responsibility	Firmware Impact
BSP	Hardware initialization	Board-specific configuration
HAL	Register abstraction	Portable peripheral access
Driver	Device control	Protocol implementation
Middleware	System services	Feature integration
Application	System logic	Business requirements



EXAMPLE: MODULAR LED DRIVER

```c
// HAL layer - hardware abstraction
typedef struct
{
    GPIO_TypeDef *port;
    uint16_t pin;
    uint8_t active_state;
} LED_HW_Config_t;

// Driver layer - device control
typedef struct
{
    LED_HW_Config_t hw_config;
    uint8_t current_state;
    uint32_t blink_period;
    uint32_t last_toggle_time;
} LED_Driver_t;

// Driver API
void LED_Init(LED_Driver_t *led, GPIO_TypeDef *port, uint16_t pin);
void LED_On(LED_Driver_t *led);
void LED_Off(LED_Driver_t *led);
void LED_Toggle(LED_Driver_t *led);
void LED_Blink(LED_Driver_t *led, uint32_t period_ms);
void LED_Process(LED_Driver_t *led); // Call from main loop

// Application layer usage
LED_Driver_t status_led;
LED_Driver_t error_led;

void Application_Init(void)
{
    LED_Init(&status_led, GPIOA, GPIO_PIN_5);
    LED_Init(&error_led, GPIOB, GPIO_PIN_14);

    LED_Blink(&status_led, 1000); // 1 second blink
}
```