Introduction to Embedded System

Embedded System: An embedded system is a specialized computing system that performs dedicated functions or tasks within a larger mechanical or electrical system. Unlike general-purpose computers, embedded systems are designed to perform specific tasks and are integrated into the hardware they control.

Characteristics:

- **Dedicated Functionality:** Designed to perform a specific function or set of functions.
- **Real-time Operation:** Often required to meet real-time constraints.
- **Resource Constraints:** Limited in terms of memory, processing power, and energy consumption.
- **Reliability and Stability:** Must operate reliably over long periods.
- **Embedded Software:** Runs embedded software tailored for specific tasks.

Examples:

- **Consumer Electronics:** Smartphones, MP3 players, and digital cameras.
- **Automotive Systems:** Anti-lock braking systems (ABS), engine control units (ECUs).
- **Industrial Automation:** Programmable logic controllers (PLCs), robotic systems.
- **Healthcare Devices:** Pacemakers, MRI machines.
- **Home Appliances:** Washing machines, microwave ovens.

Factors for Selecting the Embedded Programming Language

Choosing the right programming language for embedded systems depends on various factors:

1. **Performance Requirements:**
    - **Real-time constraints:** Languages that provide precise control over timing and execution, such as C or assembly.
    - **Speed:** High-level languages like C or C++ are preferred for performance-critical applications.
2. **Resource Constraints:**
    - **Memory Footprint:** Languages with low memory overhead are preferred, such as C or assembly.
    - **Processing Power:** Efficient languages that generate minimal overhead and maximize processor utilization.
3. **Development Efficiency:**
    - **Ease of Use:** High-level languages like Python or C++ can speed up development time.
    - **Toolchain Support:** Availability of compilers, debuggers, and IDEs for the chosen language.
4. **Portability:**

    - **Cross-Platform Compatibility:** C and C++ are highly portable and widely supported across different platforms.
5. **Community and Ecosystem:**
    - **Library Support:** Availability of libraries and frameworks.
    - **Community Support:** Active community for troubleshooting and support.
6. **Safety and Reliability:**
    - **Safety-Critical Applications:** Languages with strong type checking and error handling, such as Ada or Rust.
    - **Reliability:** Languages that provide predictable and reliable performance.
7. **Cost and Licensing:**
    - **Open Source vs. Proprietary:** Cost of development tools and licenses.
    - **Maintenance:** Long-term maintenance and support costs.

Difference Between C and Embedded C

C:

- **General Purpose:** Designed for general-purpose programming.
- **Standard Library:** Rich set of standard libraries for various applications.
- **Portability:** Highly portable across different platforms and operating systems.

Embedded C:

- **Specialized:** Designed for programming embedded systems.
- **Hardware Interaction:** Provides direct access to hardware and I/O operations.
- **Resource Constraints:** Optimized for low memory and processing power.
- **Real-time Performance:** Supports real-time programming with precise timing control.

Key Differences:

1. **Standard Libraries:**
    - C: Includes standard libraries like `stdio.h`, `stdlib.h`.
    - Embedded C: Minimal or no standard libraries, with libraries specific to hardware.
2. **Memory Management:**
    - C: Dynamic memory allocation using `malloc` and `free`.
    - Embedded C: Often avoids dynamic memory allocation due to resource constraints.
3. **I/O Operations:**
    - C: File and console I/O using standard functions.
    - Embedded C: Direct hardware access and low-level I/O operations.
4. **Code Optimization:**
    - C: General optimization for performance.
    - Embedded C: Optimized for minimal memory usage and efficient execution.

Keywords and Data Types

Keywords:

- **Keywords in C:** `int`, `char`, `float`, `if`, `else`, `while`, `for`, `return`, etc.
- **Keywords in Embedded C:** Includes all C keywords, with additional keywords specific to embedded programming, such as `__interrupt`, `__far`, `__near`.

Data Types:

- **Basic Data Types:** `int`, `char`, `float`, `double`, `void`.
- **Derived Data Types:** Arrays, pointers, structures, unions.
- **Embedded-Specific Data Types:** Fixed-width integers (`int8_t`, `uint8_t`, `int16_t`, `uint16_t`, `int32_t`, `uint32_t`), bit-fields.

Components of Embedded Program

1. **Header Files:**
    - Contain declarations of functions, macros, and data types.
    - Examples: `#include <stdio.h>`, `#include <stdint.h>`.
2. **Main Function:**
    - Entry point of the program.
    - Example: `int main(void) { ... }`.
3. **Initialization Code:**
    - Sets up hardware, initializes variables, configures peripherals.
    - Example: `init_uart(); init_timer();`.
4. **Infinite Loop:**
    - Ensures the program runs continuously.
    - Example: `while (1) { ... }`.
5. **Interrupt Service Routines (ISR):**
    - Handles interrupts and provides real-time response.
    - Example: `void __interrupt isr(void) { ... }`.
6. **Peripheral Configuration:**
    - Configures and manages peripherals like GPIO, UART, ADC.
    - Example: `setup_gpio(); setup_adc();`.

```c
#include <stdio.h>  // Header Files

// Function Prototypes
void init_peripherals(void);
void main_loop(void);

// Global Variables
int global_variable = 0;

int main(void) {
    init_peripherals();  // Initialization Code
    while (1) {          // Infinite Loop
        main_loop();     // Main Functionality
    }
    return 0;
}

void init_peripherals(void) {
    // Peripheral Initialization
}

void main_loop(void) {
    // Main Functionality
}
```

Basic Concepts of Embedded Programming

1. **Real-time Systems:**
    - Systems that must respond to inputs within a specific time frame.
    - Examples: Automotive control systems, industrial automation.
2. **Interrupts:**
    - Mechanism for handling asynchronous events.
    - ISR (Interrupt Service Routine) executes in response to an interrupt.
3. **Memory Management:**
    - Efficient use of limited memory resources.
    - Static vs. dynamic memory allocation.
4. **I/O Operations:**
    - Direct interaction with hardware using memory-mapped I/O or port I/O.

    - Example: Reading/writing to GPIO pins.
5. **Concurrency:**
    - Managing multiple tasks concurrently.
    - Techniques: Polling, interrupts, RTOS (Real-Time Operating System).
6. **Power Management:**
    - Techniques to reduce power consumption.
    - Examples: Sleep modes, low-power states.
7. **Communication Protocols:**
    - Protocols for data exchange between devices.
    - Examples: UART, SPI, I2C, CAN.
8. **Error Handling:**
    - Detecting and handling errors gracefully.
    - Techniques: Watchdog timers, fail-safe mechanisms.

Example:

Embedded C Program to Toggle an LED

Assumptions:

- The microcontroller is an AVR ATmega328P (like the one used in Arduino Uno).
- The LED is connected to pin PB0 (pin 8 on the Arduino Uno).

```c
#include <avr/io.h>
#include <util/delay.h>

#define LED_PIN PB0    // Define the LED pin
#define DELAY_MS 1000  // Define delay time in milliseconds

int main(void) {
    // Set the LED pin as an output
    DDRB |= (1 << LED_PIN);

    while (1) {
        // Toggle the LED pin
        PORTB ^= (1 << LED_PIN);

        // Wait for DELAY_MS milliseconds
        _delay_ms(DELAY_MS);
    }

    return 0;
}
```

Explanation of the above code:

1. **Include Headers:**
   - `#include <avr/io.h>`: Provides macros for port and register names specific to the AVR microcontroller.
   - `#include <util/delay.h>`: Provides the `_delay_ms()` function for creating delays.
2. **Define Macros:**
   - `#define LED_PIN PB0`: Defines `LED_PIN` as `PB0`, which is the bit number for pin 8 on the Arduino Uno.
   - `#define DELAY_MS 1000`: Defines a delay of 1000 milliseconds (1 second).
3. **Setup LED Pin:**
   - `DDRB |= (1 << LED_PIN)`: Sets the direction of the `LED_PIN` to output. `DDRB` is the Data Direction Register for port B. The `|= (1 << LED_PIN)` sets the bit corresponding to `LED_PIN` to 1, configuring it as an output.
4. **Main Loop:**
   - `PORTB ^= (1 << LED_PIN)`: Toggles the state of the `LED_PIN`. `PORTB` is the Data Register for port B. The `^= (1 << LED_PIN)` operation flips the bit corresponding to `LED_PIN`, toggling the LED.
   - `_delay_ms(DELAY_MS)`: Delays execution for 1000 milliseconds.