MODULE -2

PROGRAMMING MSP430

1. Development Environment for Embedded Systems

Embedded systems development requires specialized environments that include hardware, software, and debugging tools tailored to the embedded hardware. A typical embedded development environment includes:

- **Toolchain**: This includes the compiler (for C or Assembly), linker, and debugger. A popular toolchain is **GCC (GNU Compiler Collection)**, used with a **cross-compiler** for the target embedded platform.
- **IDE (Integrated Development Environment)**: IDEs like **Keil µVision**, **IAR Embedded Workbench**, or **Eclipse** with plugins like **GNU ARM Eclipse** are commonly used.
- **Debugging and Simulation Tools**: Debugging tools like **JTAG** or **SWD** and simulators for the target hardware (e.g., **Proteus**, **QEMU**) are essential for running code on actual hardware or virtual environments.
- **Real-time Operating Systems (RTOS)**: Some embedded systems use an RTOS like **FreeRTOS**, **EmbOS**, or **ChibiOS** to handle multitasking and scheduling.

2. Aspects of C for Embedded Systems

The C language is widely used in embedded systems for its balance of control and abstraction. Some key aspects when using C for embedded systems:

- **Low-Level Access**: C provides direct access to memory, which is essential for embedded systems.
- **Portability**: C allows the code to be portable across different platforms, making it easier to scale embedded systems to different microcontrollers.
- **Efficient Memory Management**: Embedded systems often have limited resources, so developers need to use memory efficiently (e.g., avoid dynamic memory allocation).
- **Optimization**: Writing optimized code is crucial for performance in embedded systems.

Example in C for Embedded Systems:

```
#include <avr/io.h> // Header for AVR microcontroller

#include <util/delay.h> // Delay function


void initLED() {
```

```
    DDRB |= (1 << PB0); // Set PB0 as output
}

void toggleLED() {
    PORTB ^= (1 << PB0); // Toggle PB0 (LED)
}

int main() {
    initLED(); // Initialize LED

    while(1) {
        toggleLED(); // Toggle the LED
        _delay_ms(500); // Delay for 500 ms
    }

    return 0;
}
```

3. Assembly Language for Embedded Systems

- **Assembly Language** provides direct control over hardware and is often used in situations requiring optimization or direct interaction with hardware peripherals.
- It is hardware-specific and written for a particular processor architecture (e.g., ARM, AVR).
- **Advantages**: Speed, small memory footprint, fine control over processor resources.
- **Disadvantages**: Complexity, harder to maintain.

Example Assembly Code (AVR):

```
; Turn on an LED connected to pin PB0 on an AVR microcontroller
LDI     R16, 0x01       ; Load immediate value 0x01 into register R16
OUT     DDRB, R16       ; Set DDRB register to configure PB0 as output
LOOP:
    SBI     PORTB, 0    ; Set bit 0 of PORTB (turn on LED)
```

```
NOP             ; No operation (delay)
CBI   PORTB, 0  ; Clear bit 0 of PORTB (turn off LED)
NOP             ; No operation (delay)
RJMP  LOOP      ; Jump back to LOOP
```

4. Register Organization in Embedded Systems

In microcontrollers, **registers** are used to hold data and control various hardware components. Registers can be classified as:

- **General-purpose registers**: Used for storing data temporarily during execution.
- **Special function registers (SFRs)**: Used for configuring hardware peripherals (e.g., timers, interrupts).

Common registers include:

- **Program Counter (PC)**: Holds the address of the next instruction.
- **Status Register (SR)**: Holds flags (e.g., carry, zero, etc.).
- **Stack Pointer (SP)**: Points to the top of the stack.

Example (AVR):

```
; Load value into a register
LDI   R16, 0xFF   ; Load 0xFF into register R16
```

5. Addressing Modes in Embedded Systems

Addressing modes define how operands (values) are accessed during instruction execution. Common addressing modes include:

- **Immediate Addressing**: Operand is provided as a constant in the instruction (e.g., LDI).
- **Direct Addressing**: Operand is in a specific memory location (e.g., MOV R0, 0x20).
- **Indirect Addressing**: Operand is accessed through a pointer stored in a register (e.g., LD R0, (R1)).
- **Indexed Addressing**: Combines a base address with an index (e.g., LD R0, 0x10(R1)).

6. Constant Generators and Emulated Instructions

- **Constant Generators**: These are built-in circuits in microcontrollers that generate specific constant values (e.g., power-of-2 values) without needing to use actual code to calculate them.

- **Emulated Instructions**: Some complex instructions may not exist in the instruction set, so they are emulated using multiple simpler instructions.

7. Instruction Set of Embedded Processors

- **RISC (Reduced Instruction Set Computing)**: Uses a small, highly optimized instruction set (e.g., ARM Cortex-M).
- **CISC (Complex Instruction Set Computing)**: Uses a larger set of instructions (e.g., x86 architecture).

Example Instruction Set (ARM):

- **MOV**: Move data between registers.
- **ADD**: Add two values.
- **CMP**: Compare two values.

8. Example Programs for Embedded Systems

- **Light an LED**: This simple program turns on an LED connected to a microcontroller.

```
#include <avr/io.h>
int main() {
   DDRB |= (1 << PB0); // Set PB0 as output
   while(1) {
      PORTB |= (1 << PB0); // Turn on LED
   }
   return 0;
}
```

- **Read Input from a Switch**: A program to read a switch input and perform an action.

```
#include <avr/io.h>
int main() {
   DDRC &= ~(1 << PC0); // Set PC0 as input (switch)
   DDRB |= (1 << PB0);  // Set PB0 as output (LED)

   while(1) {
      if (PINC & (1 << PC0)) { // If switch is pressed
```

```
            PORTB |= (1 << PB0); // Turn on LED
        } else {
            PORTB &= ~(1 << PB0); // Turn off LED
        }
    }
    return 0;
}
```

- **Flashing Light with Delay**: Example of using a simple delay function to flash an LED.

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

int main() {
    DDRB |= (1 << PB0); // Set PB0 as output

    while(1) {
        PORTB |= (1 << PB0); // Turn on LED
        _delay_ms(500);      // Delay for 500ms
        PORTB &= ~(1 << PB0); // Turn off LED
        _delay_ms(500);      // Delay for 500ms
    }
    return 0;
}
```

9. Interrupts and Low Power Modes

- **Interrupts**: Interrupts are used to handle asynchronous events (e.g., a button press or timer overflow). When an interrupt occurs, the current execution is paused, and the interrupt service routine (ISR) is executed.
- **Low Power Modes**: Embedded systems often need to conserve power. Microcontrollers typically have several low-power modes:
    - **Sleep Mode**: The CPU is halted, but peripherals can still run.
    - **Idle Mode**: The CPU runs at a reduced clock speed.

- **Power Down**: The CPU and most peripherals are disabled.

Example of Interrupt Handling:

```
ISR(INT0_vect) {
    // Interrupt service routine for INT0
    PORTB ^= (1 << PB0); // Toggle LED
}

int main() {
    DDRB |= (1 << PB0); // Set PB0 as output
    EIMSK |= (1 << INT0); // Enable external interrupt INT0
    sei(); // Enable global interrupts
    while(1);
    return 0;
}
```