## 4. Embedded Systems Programming

### Using C/C++ for Microcontrollers:

Microcontrollers are the backbone of many IoT devices, and programming them requires a solid understanding of both hardware and software. C and C++ are the predominant languages used in embedded systems programming due to their efficiency and control over hardware.

### Programming Basics

#### 1. Introduction to C/C++ for Microcontrollers:

- **C Language Basics:**
    - **Syntax and Structure:** Understanding the basic syntax, data types, variables, operators, and control structures.
    - **Functions:** Defining and calling functions, understanding scope and lifetime of variables.
    - **Pointers and Memory Management:** Using pointers, dynamic memory allocation, and managing memory effectively.
- **C++ Language Basics:**
    - **Object-Oriented Programming (OOP):** Understanding classes, objects, inheritance, polymorphism, and encapsulation.
    - **Templates and STL:** Using templates for generic programming and Standard Template Library (STL) for data structures and algorithms.

**Example: Basic C Program for LED Blinking on STM32:**

```c
#include "stm32f4xx.h"

void delay(uint32_t count) {
    while (count--) {
        __NOP();
    }
}

int main(void) {
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; // Enable clock for GPIOA
    GPIOA->MODER |= GPIO_MODER_MODER5_0; // Set PA5 as output

    while (1) {
        GPIOA->ODR ^= GPIO_ODR_OD5; // Toggle PA5
        delay(1000000); // Simple delay
    }
}
```

### 2. Development Environment:

- **IDE and Toolchains:** Overview of popular development environments such as STM32CubeIDE, Keil MDK, and IAR Embedded Workbench.
- **Project Setup:** Creating and configuring projects, including setting up linker scripts, startup files, and managing dependencies.

### 3. Writing Efficient Code:

- **Code Optimization:** Techniques for optimizing code size and execution speed.
- **Debugging:** Using debuggers and tools like GDB, OpenOCD, and integrated debugger in IDEs.
- **Testing:** Writing and running test cases to ensure code reliability and performance.

## Peripheral Control

Microcontrollers interact with the external world through various peripherals. Understanding how to configure and control these peripherals is crucial for building functional IoT devices.

### 1. General Purpose Input/Output (GPIO):

- **Configuring GPIO Pins:** Setting pins as input or output, configuring pull-up and pull-down resistors.
- **Reading and Writing to GPIO:** Using registers to read input states and write output values.

### Example: GPIO Configuration and Control on ESP32:

```c
#include "driver/gpio.h"

#define LED_PIN GPIO_NUM_2

void app_main(void) {
    gpio_pad_select_gpio(LED_PIN);
    gpio_set_direction(LED_PIN, GPIO_MODE_OUTPUT);

    while (1) {
        gpio_set_level(LED_PIN, 1); // Set LED_PIN high
        vTaskDelay(1000 / portTICK_PERIOD_MS); // Delay for 1 second
        gpio_set_level(LED_PIN, 0); // Set LED_PIN low
        vTaskDelay(1000 / portTICK_PERIOD_MS); // Delay for 1 second
    }
}
```

**2. Analog-to-Digital Converter (ADC):**

- **Configuring ADC:** Setting up the ADC module, selecting input channels, and configuring resolution.
- **Reading Analog Values:** Initiating ADC conversions and reading the digital output values.

**Example: Reading ADC Value on STM32:**

```c
#include "stm32f4xx.h"

void ADC_Init(void) {
    RCC->APB2ENR |= RCC_APB2ENR_ADC1EN; // Enable ADC1 clock
    ADC1->SQR3 = 0; // Set channel 0 as the first conversion
    ADC1->CR2 |= ADC_CR2_ADON; // Enable ADC1
}

uint16_t ADC_Read(void) {
    ADC1->CR2 |= ADC_CR2_SWSTART; // Start conversion
    while (!(ADC1->SR & ADC_SR_EOC)); // Wait for conversion to complete
    return ADC1->DR; // Read conversion result
}

int main(void) {
    ADC_Init();
    uint16_t adc_value;

    while (1) {
        adc_value = ADC_Read();
        // Use adc_value for further processing
    }
}
```

### 3. Universal Asynchronous Receiver/Transmitter (UART):

- **Configuring UART:** Setting baud rate, data bits, stop bits, and parity.
- **Sending and Receiving Data:** Using UART registers or libraries to transmit and receive data.

### Example: UART Communication on STM32:

```c
#include "stm32f4xx.h"

void UART_Init(void) {
    RCC->APB1ENR |= RCC_APB1ENR_USART2EN; // Enable USART2 clock
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; // Enable GPIOA clock

    // Configure PA2 as USART2_TX
    GPIOA->MODER |= GPIO_MODER_MODER2_1;
    GPIOA->AFR[0] |= (7 << GPIO_AFRL_AFSEL2_Pos);

    USART2->BRR = 0x683; // 9600 baud rate
    USART2->CR1 |= USART_CR1_TE | USART_CR1_RE; // Enable transmitter and receiver
    USART2->CR1 |= USART_CR1_UE; // Enable USART2
}

void UART_SendChar(char c) {
    while (!(USART2->SR & USART_SR_TXE));
    USART2->DR = c;
}

char UART_ReceiveChar(void) {
    while (!(USART2->SR & USART_SR_RXNE));
    return USART2->DR;
}

int main(void) {
    UART_Init();
    char received_char;

    while (1) {
        received_char = UART_ReceiveChar();
        UART_SendChar(received_char); // Echo received character
    }
}
```

## Real-Time Operating Systems (RTOS):

RTOS provides a framework for managing multiple tasks in embedded systems, ensuring that critical tasks are executed within their time constraints.

### Introduction to RTOS

#### 1. What is an RTOS?

- **Definition and Purpose:** An RTOS is designed to manage hardware resources, run multiple tasks, and ensure that real-time constraints are met.
- **Key Features:** Task scheduling, inter-task communication, synchronization, and resource management.

**2. Benefits of Using an RTOS:**

- **Deterministic Behavior:** Ensures predictable task execution and response times.
- **Task Management:** Simplifies the design of complex systems by breaking down functionality into smaller, manageable tasks.
- **Resource Utilization:** Efficiently manages CPU, memory, and peripherals.

**3. Common RTOS in IoT:**

- **FreeRTOS:** An open-source RTOS widely used in IoT applications.
- **Zephyr:** A scalable, open-source RTOS with extensive hardware support.
- **ThreadX:** A commercial RTOS known for its performance and reliability.

**FreeRTOS**

**1. Overview:**

- **Open Source:** FreeRTOS is open-source and widely used in various embedded applications.
- **Lightweight:** Designed to be small and efficient, suitable for microcontrollers with limited resources.
- **Community Support:** Extensive documentation and community support.

**2. Key Features:**

- **Task Management:** Create, delete, and manage tasks with priorities.
- **Inter-Task Communication:** Queues, semaphores, mutexes, and event groups for communication and synchronization.
- **Memory Management:** Dynamic and static memory allocation options.

**Example: Creating Tasks in FreeRTOS on ESP32:**

```c
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>

void Task1(void *pvParameters) {
    while (1) {
        printf("Task1 is running\n");
        vTaskDelay(1000 / portTICK_PERIOD_MS); // Delay for 1 second
    }
}

void Task2(void *pvParameters) {
    while (1) {
        printf("Task2 is running\n");
        vTaskDelay(2000 / portTICK_PERIOD_MS); // Delay for 2 seconds
    }
}

void app_main(void) {
    xTaskCreate(Task1, "Task1", 2048, NULL, 1, NULL);
    xTaskCreate(Task2, "Task2", 2048, NULL, 1, NULL);
}
```

### 3. Advanced Features:

- **Tickless Idle:** Reduce power consumption by entering a low-power state when the system is idle.
- **Trace Facility:** Debug and analyze the behavior of tasks and system performance.

## Zephyr RTOS

Zephyr is an open-source real-time operating system (RTOS) designed for resource-constrained devices, making it an excellent choice for IoT applications. This section provides a comprehensive guide to Zephyr, covering its architecture, setup, programming, and advanced features.

### Overview of Zephyr

### 1. Introduction to Zephyr:

- **Scalability:** Supports a wide range of hardware platforms, from simple microcontrollers to complex multi-core systems.
- **Modularity:** Features a highly configurable architecture that allows developers to include only the necessary components, minimizing memory footprint.
- **Community and Support:** Backed by the Zephyr Project, which is governed by the Linux Foundation, ensuring robust community support and continuous development.

**2. Key Features of Zephyr:**

- **Kernel Services:** Preemptive and cooperative multitasking, inter-thread communication, synchronization, and memory management.
- **Device Drivers:** Comprehensive support for a variety of peripherals and sensors.
- **Networking:** Support for various networking protocols, including Bluetooth, Wi-Fi, Ethernet, and LoRa.
- **Security:** Built-in security features such as secure boot, access control, and secure firmware updates.

**Setting Up the Development Environment**

**1. Prerequisites:**

- **Operating System:** Zephyr development can be done on Linux, macOS, or Windows. However, Linux is the recommended environment.
- **Toolchain:** Install the appropriate toolchain for your target hardware. Zephyr supports various toolchains such as GCC, ARM, and others.

**2. Installing Zephyr:**

**On Linux:**

```
# Install dependencies
sudo apt-get update
sudo apt-get install -y git cmake ninja-build gperf \
  ccache dfu-util device-tree-compiler wget \
  python3-pip python3-setuptools python3-wheel xz-utils file \
  make gcc gcc-multilib g++-multilib libsdl2-dev

# Clone Zephyr repository
west init zephyrproject
cd zephyrproject
west update

# Install Zephyr dependencies
pip3 install -r zephyr/scripts/requirements.txt
```

**On macOS:**

```
# Install dependencies
brew install cmake ninja dfu-util python3 gperf wget \
  device-tree-compiler ccache

# Clone Zephyr repository
west init zephyrproject
cd zephyrproject
west update

# Install Zephyr dependencies
pip3 install -r zephyr/scripts/requirements.txt
```

**On Windows:**

1. Download and install the Zephyr SDK from the Zephyr Project website.
2. Install Git and Python3.
3. Use Git Bash to clone the Zephyr repository and install dependencies:

   ```
   # Clone Zephyr repository
   west init zephyrproject
   cd zephyrproject
   west update

   # Install Zephyr dependencies
   pip3 install -r zephyr/scripts/requirements.txt
   ```

**3. Setting Up the Toolchain:**

- **Zephyr SDK:** Install the Zephyr SDK, which includes the required toolchain and tools.
- **GNU Arm Embedded Toolchain:** Alternatively, install the ARM GCC toolchain for ARM-based microcontrollers.

**Creating and Building a Zephyr Application**

**1. Project Structure:**

- **Project Directory:** Create a directory for your Zephyr project.
- **CMakeLists.txt:** Define the build instructions for the project.
- **prj.conf:** Configuration file specifying the kernel and subsystem options.
- **src/main.c:** The main application code.

**2. Example: Blinking an LED**

## Directory Structure:

```
my_project/
├── CMakeLists.txt
├── prj.conf
└── src/
    └── main.c
```

## CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.13.1)
set(BOARD nrf52840dk_nrf52840)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(blinky)
target_sources(app PRIVATE src/main.c)
```

## prj.conf:

```
CONFIG_GPIO=y
CONFIG_LOG=y
```

## src/main.c:

```c
#include <zephyr.h>
#include <device.h>
#include <drivers/gpio.h>
#include <sys/printk.h>

#define LED_PORT DT_ALIAS_LED0_GPIOS_CONTROLLER
#define LED     DT_ALIAS_LED0_GPIOS_PIN
#define SLEEP_TIME_MS   1000

void main(void) {
    struct device *dev;
    bool led_is_on = true;

    printk("Starting Blinky Example\n");

    dev = device_get_binding(LED_PORT);
    if (dev == NULL) {
        printk("Failed to get binding for LED port\n");
        return;
    }

    gpio_pin_configure(dev, LED, GPIO_OUTPUT_ACTIVE);

    while (1) {
        gpio_pin_set(dev, LED, (int)led_is_on);
        led_is_on = !led_is_on;
        k_sleep(K_MSEC(SLEEP_TIME_MS));
    }
}
```

## 3. Building and Flashing the Application:

### Building:

```
west build -b nrf52840dk_nrf52840
```

### Flashing:

```
west flash
```

**Advanced Zephyr Features**

**1. Device Drivers:**

- **Overview:** Zephyr includes a comprehensive set of drivers for various peripherals such as GPIO, I2C, SPI, UART, and more.
- **Custom Drivers:** How to write and integrate custom device drivers into Zephyr.

**Example: Custom GPIO Driver:**

```c
#include <device.h>
#include <drivers/gpio.h>

struct gpio_driver_api {
    int (*pin_set)(struct device *dev, int pin, int value);
};

static int custom_gpio_pin_set(struct device *dev, int pin, int value) {
    // Custom implementation
    return 0;
}

static const struct gpio_driver_api custom_gpio_api = {
    .pin_set = custom_gpio_pin_set,
};

DEVICE_AND_API_INIT(custom_gpio, "CUSTOM_GPIO", &custom_gpio_init,
                    NULL, NULL, POST_KERNEL, CONFIG_KERNEL_INIT_PRIORITY_DEFAULT,
                    &custom_gpio_api);
```

**2. Networking:**

- **Protocols:** Zephyr supports various networking protocols including Bluetooth, Wi-Fi, Ethernet, 6LoWPAN, and LoRa.
- **Network Stack:** Overview of Zephyr's network stack and how to configure and use it.

**Example: Bluetooth Beacon:**

```c
#include <zephyr.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/hci.h>

static const struct bt_data ad[] = {
    BT_DATA_BYTES(BT_DATA_FLAGS, (BT_LE_AD_GENERAL | BT_LE_AD_NO_BREDR)),
    BT_DATA_BYTES(BT_DATA_UUID16_ALL, 0x18, 0x0f),
};

void main(void) {
    int err;

    err = bt_enable(NULL);
    if (err) {
        printk("Bluetooth init failed (err %d)\n", err);
        return;
    }

    err = bt_le_adv_start(BT_LE_ADV_CONN, ad, ARRAY_SIZE(ad), NULL, 0);
    if (err) {
        printk("Advertising failed to start (err %d)\n", err);
        return;
    }

    printk("Beacon started\n");
}
```

**3. Power Management:**

- **Low Power Modes:** How to configure and use low power modes to extend battery life.
- **Power Management API:** Overview of the power management API and how to use it in applications.

**Example: Entering Sleep Mode:**

```
#include <zephyr.h>
#include <power/power.h>

void main(void) {
    while (1) {
        k_sleep(K_SECONDS(5)); // Simulate work
        sys_pm_force_power_state(SYS_POWER_STATE_SLEEP_1); // Enter sleep mode
    }
}
```

**4. Security:**

- **Secure Boot:** Configuring secure boot to protect the device from unauthorized firmware updates.
- **Access Control:** Implementing access control mechanisms to secure resources.

**Example: Enabling Secure Boot:**

1. Configure the bootloader to verify the firmware image signature.
2. Use Zephyr's `mcuboot` project to implement secure boot.

**Debugging and Testing**

**1. Debugging:**

- **Debugging Tools:** Using tools like GDB, OpenOCD, and integrated IDE debuggers to debug Zephyr applications.
- **Debugging Techniques:** Setting breakpoints, watching variables, and stepping through code.

**Example: Using GDB with OpenOCD:**

1. Start OpenOCD:

   ```
   openocd -f interface/stlink.cfg -f target/stm32f4x.cfg
   ```

2. Start GDB:

   ```
   arm-none-eabi-gdb build/zephyr/zephyr.elf
   ```

3. Connect to OpenOCD:

   ```
   (gdb) target remote :3333
   ```

**2. Testing:**

- **Unit Testing:** Writing unit tests for individual components using Zephyr's testing framework.
- **Integration Testing:** Testing the interaction between multiple components.

**Example: Unit Testing with Zephyr's ztest:**

```
#include <ztest.h>

static void test_addition(void) {
    zassert_equal(1 + 1, 2, "1 + 1 should equal 2");
}

void test_main(void) {
    ztest_test_suite(math_tests,
                     ztest_unit_test(test_addition));
    ztest_run_test_suite(math_tests);



}
```

**Contributing to Zephyr**

**1. Community and Contributions:**

- **Zephyr Project:** Overview of the Zephyr Project and its governance.
- **Contributing Guidelines:** How to contribute to Zephyr, including coding standards, submitting patches, and participating in discussions.

**2. Documentation and Resources:**

- **Official Documentation:** How to use Zephyr's official documentation and resources.
- **Community Resources:** Mailing lists, forums, and chat channels for community support.

By mastering Zephyr, you can develop robust and efficient IoT applications that leverage the powerful features of this scalable RTOS. Whether you are building simple sensor nodes or complex multi-core systems, Zephyr provides the tools and capabilities to meet your requirements.

**ThreadX**

**1. Overview:**

- **Commercial RTOS:** ThreadX is a commercial RTOS developed by Express Logic, now part of Microsoft.
- **High Performance:** Known for its performance and small footprint.

- **Integration:** Integrated with Microsoft Azure RTOS, providing cloud connectivity features.

**2. Key Features:**

- **Task Management:** Simple API for creating and managing tasks with time slicing and round-robin scheduling.
- **Inter-Task Communication:** Event flags, message queues, semaphores, and mutexes.
- **Memory Management:** Efficient memory management with block pools and byte pools.

**Example: Creating Tasks in ThreadX:**

```c
#include "tx_api.h"

void Task1(ULONG thread_input) {
    while (1) {
        printf("Task1 is running\n");
        tx_thread_sleep(100); // Delay for 100 ticks
    }
}

void Task2(ULONG thread_input) {
    while (1) {
        printf("Task2 is running\n");
        tx_thread_sleep(200); // Delay for 200 ticks
    }
}

void tx_application_define(void *first_unused_memory) {
    tx_thread_create(&Task1, "Task1", Task1, 0,
                     stack1, sizeof(stack1),
                     1, 1, TX_NO_TIME_SLICE, TX_AUTO_START);

    tx_thread_create(&Task2, "Task2", Task2, 0,
                     stack2, sizeof(stack2),
                     1, 1, TX_NO_TIME_SLICE, TX_AUTO_START);
}

int main(void) {
    tx_kernel_enter();
}
```

**3. Advanced Features:**

- **TraceX:** A real-time event logging, performance analysis, and visualization tool.
- **FileX:** A high-performance, FAT-compatible file system.
- **NetX:** A full-featured TCP/IP stack with support for various networking protocols.

By mastering embedded systems programming with C/C++ and understanding the use of RTOS such as FreeRTOS, Zephyr, and ThreadX, you can develop robust and efficient IoT applications. These skills are crucial for building devices that can handle complex tasks, manage resources effectively, and operate reliably in real-time environments.