Virtual API and Factory Pattern in Embedded Systems (Polymorphism in C)

1. Core Concepts of the Virtual API Pattern

In modern embedded software engineering, directly coupling your application logic (e.g., a sensor reading algorithm) to specific hardware registers (e.g., STM32 UART or SPI) makes the code rigid and impossible to test on a PC. The **Virtual API Pattern** solves this by bringing C++ style Polymorphism (Interfaces and Virtual Tables) into standard C.

For a robust, MISRA-compliant architecture, the following principles apply:

- **The Virtual Table (VTable):** A `struct` containing only function pointers. This acts as the "contract" or "interface" that the hardware driver must fulfill.
- **The Interface Object:** A base `struct` that contains a `const` pointer to the VTable.
- **Inheritance via Composition:** The concrete hardware driver (e.g., `UartDriver_t`) embeds the interface object as its very first member.
- **The Factory Function:** A function that initializes the hardware but returns a pointer to the generic *Interface*, completely hiding the hardware details from the application layer.

---

2. STM32 Example Implementation

In this example, we create a generic `I_Serial` communication interface. The application layer will use this interface to send data, completely unaware of whether the underlying hardware is UART, SPI, or a USB Virtual COM port.

2.1. The Interface: `I_Serial.h`

This defines the generic contract. Notice how the application will only interact with the inline wrapper functions.

```
#ifndef I_SERIAL_H
#define I_SERIAL_H

#include <stdint.h>
```

```
// 1. Forward declaration of the VTable
typedef struct I_Serial_VTable I_Serial_VTable_t;

// 2. The Virtual Table (List of function pointers)
struct I_Serial_VTable {
    void (*Write)(void *const me, uint8_t data);
    uint8_t (*Read)(void *const me);
};

// 3. The Base Interface Structure
typedef struct {
    const I_Serial_VTable_t *vptr; // Pointer to the ROM-stored VTable
} I_Serial_t;

// 4. Inline Wrapper Functions (For clean application code)
static inline void I_Serial_Write(I_Serial_t *const me, uint8_t data) {
    me->vptr->Write(me, data); // Dereference and call
}

static inline uint8_t I_Serial_Read(I_Serial_t *const me) {
    return me->vptr->Read(me);
}

#endif // I_SERIAL_H
```

2.2. The Concrete Implementation: `UartDriver.c`

This file implements the `I_Serial` interface for a specific STM32 UART peripheral.

```
#include "I_Serial.h"
#include "stm32f4xx_hal.h"

// 1. The Concrete Class (Inherits I_Serial_t via Composition)
typedef struct {
    I_Serial_t base;           // MUST be the first member for safe typec
    UART_HandleTypeDef *huart; // Hardware-specific data
} UartDriver_t;

// 2. Static Memory Pool (Zero Dynamic Memory Allocation)
static UartDriver_t uart1_instance;
```

```
// 3. Concrete Implementation of the Write Function
static void Uart_Write_Impl(void *const me, uint8_t data) {
    // Cast the generic 'me' pointer back to the concrete UartDriver_t
    UartDriver_t *self = (UartDriver_t *)me;

    // Hardware-specific action
    HAL_UART_Transmit(self->huart, &data, 1, HAL_MAX_DELAY);
}

// 4. Concrete Implementation of the Read Function
static uint8_t Uart_Read_Impl(void *const me) {
    UartDriver_t *self = (UartDriver_t *)me;
    uint8_t rx_data = 0;
    HAL_UART_Receive(self->huart, &rx_data, 1, HAL_MAX_DELAY);
    return rx_data;
}

// 5. The VTable Definition (Stored in ROM/Flash via 'const')
static const I_Serial_VTable_t uart_vtable = {
    .Write = Uart_Write_Impl,
    .Read  = Uart_Read_Impl
};

// 6. The Factory Function
// Initializes the UART but returns the generic I_Serial_t interface poir
I_Serial_t* UartDriver_Create(UART_HandleTypeDef *huart_handle) {
    uart1_instance.base.vptr = &uart_vtable; // Link the virtual table
    uart1_instance.huart = huart_handle;     // Set hardware context

    // Safely cast up to the base interface
    return (I_Serial_t*)&uart1_instance;
}
```

2.3. System Integration: `main.c`

The application logic is completely decoupled from the STM32 HAL. You could easily pass a `MockSerial_Create()` interface during PC Unit Testing (Software-in-the-Loop).

```
#include "stm32f4xx_hal.h"
#include "I_Serial.h"

// External declaration of the Factory function
```

```
extern I_Serial_t* UartDriver_Create(UART_HandleTypeDef *huart_handle);

extern UART_HandleTypeDef huart1;

// --- APPLICATION LAYER ---
// App_Run does NOT know about UART, STM32, or HAL.
// It only knows the I_Serial_t contract.
void App_Run(I_Serial_t *comm_channel) {
    // Send a boot message
    I_Serial_Write(comm_channel, 0xAA);
    I_Serial_Write(comm_channel, 0xBB);
}
// -------------------------

int main(void) {
    HAL_Init();
    // SystemClock_Config() and MX_USART1_UART_Init() assumed here.

    // 1. Create the hardware driver via the Factory
    I_Serial_t *mySerial = UartDriver_Create(&huart1);

    // 2. Pass the interface to the Application Layer (Dependency Injecti
    App_Run(mySerial);

    while (1) {
        // Superloop
    }
}
```

---

3. Architectural Warning: ROM and Overhead Considerations

While the Virtual API pattern is essential for scalable architectures and unit testing, it comes with a minor trade-off:

- **ROM Footprint:** Every VTable consumes a small amount of Flash memory (ROM).
- **Call Overhead:** Calling a function via a pointer (`me->vptr->Write()`) requires reading the address from memory before branching to it, which takes a few extra CPU cycles compared to a direct function call.
- **Best Practice:** Use this pattern for module boundaries (e.g., Communication Buses, Display Drivers, File Systems). Do **not** use it for extremely high-frequency ISRs (Interrupt Service

Routines) or basic GPIO toggling where microsecond precision is required.