Callback (Observer) Pattern in Embedded Systems and STM32 Implementation

1. Core Concepts of the Callback Pattern

The callback (or observer) pattern is a central design pattern used to notify multiple objects of state changes to an observed object. In traditional C programming, callbacks are often implemented as bare function pointers, which can lead to architectural issues in embedded systems. For a robust architecture, the following principles should be applied:

* **Encapsulating Callbacks in Structs:** Callbacks must never be defined as bare function pointers; they should always be enclosed within a `struct`.

* **Context Passing:** The callback function itself must take a pointer to this callback structure as a parameter.

* **Avoiding `void *user_data`:** Instead of inventing a `void *user_data` variable to pass data to callback functions , the `CONTAINER_OF` macro should be used to safely access the data of the main object hosting the callback.

* **Complete Decoupling:** The observed object (Subject) does not need to know the internal structure or API of the observing objects (Observers) ; it only relies on the shared callback structure.

---

2. STM32 Example Implementation

This example demonstrates how a hardware button (Subject), when pressed, triggers an LED controller (Observer) that is completely independent of the button.

2.1. Common Utility: `ContainerOf.h`

This macro is a cornerstone of object-oriented C architecture, allowing you to reach the main object via pointer arithmetic.

```
#ifndef CONTAINER_OF_H
#define CONTAINER_OF_H
#include <stddef.h>

#define CONTAINER_OF(ptr, type, member) \
    ((type *)((char *)(ptr) - offsetof(type, member)))

#endif // CONTAINER_OF_H
```

2.2. The Observed Object (Subject): `Button.h` and `Button.c`

The button object only knows the callback structure. When an event occurs, it triggers this structure.

`Button.h`

```
#ifndef BUTTON_H
#define BUTTON_H

typedef struct button_callback button_callback_t;

// Callback Struct
struct button_callback {
    void (*cb)(button_callback_t *cb_ptr);
};

// Button Object
typedef struct {
    button_callback_t *callback;
} Button_t;

void Button_Init(Button_t *self);
void Button_AddCallback(Button_t *self, button_callback_t *cb);
void Button_IRQHandler(Button_t *self);

#endif // BUTTON_H
```

`Button.c`

```c
#include "Button.h"
#include <stddef.h>

void Button_Init(Button_t *self) {
    self->callback = NULL;
}

void Button_AddCallback(Button_t *self, button_callback_t *cb) {
    self->callback = cb;
}

void Button_IRQHandler(Button_t *self) {
    if(self->callback && self->callback->cb) {
        self->callback->cb(self->callback); // Notify the event
    }
}
```

2.3. The Observer Object (Observer): `LedController.h` and `LedController.c`

The LED controller embeds the button callback structure within itself (Composition).

`LedController.h`

```c
#ifndef LED_CONTROLLER_H
#define LED_CONTROLLER_H

#include "Button.h"
#include "stm32f4xx_hal.h"

// Observer Object
typedef struct {
    uint16_t led_pin;
    GPIO_TypeDef *led_port;
    button_callback_t btn_cb; // Callback structure is embedded into the object
} LedController_t;

void LedController_Init(LedController_t *self, GPIO_TypeDef *port, uint16_t pin);
button_callback_t* LedController_GetCallback(LedController_t *self);

#endif // LED_CONTROLLER_H
```

`LedController.c`

```c
#include "LedController.h"
#include "ContainerOf.h"

// Actual Callback Function
static void LedController_OnButtonPress(button_callback_t *cb_ptr) {
    // Safe casting from callback pointer to the main LedController_t object
    LedController_t *self = CONTAINER_OF(cb_ptr, LedController_t, btn_cb);

    // Access the main object's data
    HAL_GPIO_TogglePin(self->led_port, self->led_pin);
}

void LedController_Init(LedController_t *self, GPIO_TypeDef *port, uint16_t pin) {
    self->led_port = port;
    self->led_pin = pin;
    self->btn_cb.cb = LedController_OnButtonPress;
}

button_callback_t* LedController_GetCallback(LedController_t *self) {
    return &self->btn_cb;
}
```

2.4. System Integration: `main.c`

The application layer initializes the objects and links them together.

```
#include "stm32f4xx_hal.h"
#include "Button.h"
#include "LedController.h"

// Global/Static definition to protect the life cycle of the objects
Button_t myButton;
LedController_t myLed;

int main(void) {
    HAL_Init();
    // Assuming SystemClock_Config() and MX_GPIO_Init() are called here.

    // 1. Initialize Objects
    Button_Init(&myButton);
    LedController_Init(&myLed, GPIOC, GPIO_PIN_13);

    // 2. Callback Registration
    Button_AddCallback(&myButton, LedController_GetCallback(&myLed));

    while (1) {
        // Superloop
    }
}

// STM32 EXTI Hardware Interrupt Handler
void EXTI15_10_IRQHandler(void) {
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);

    // Route the hardware interrupt to the Button object
    Button_IRQHandler(&myButton);
}
```

---

3. Architectural Warning: Memory Life Cycle

When applying this pattern, one of the biggest pitfalls is failing to manage the life cycle of the objects. If the `LedController_t` object is created locally (on the stack) inside a function, it will be destroyed when the function goes out of scope. However, the Button object will still try to call that garbage pointer, leading to an immediate system crash (HardFault). Therefore, these objects must always be defined at the `static` or global level to ensure they persist in memory.