Інструкція зі створення модулів для проекту ModuChill

Зміст

Вступ

ModuChill - це модульна система керування холодильним обладнанням, розроблена на ESP-IDF для ESP32. Система складається з незмінного ядра (core) та набору модулів, які можна додавати або видаляти залежно від потреб конкретного проекту.

Ця інструкція описує процес створення нових модулів, які сумісні з архітектурою системи та можуть бути легко інтегровані у проект.

Структура модуля

Кожен модуль повинен мати наступну файлову структуру:

```
modules/my_module_name/
├── CMakeLists.txt              # Налаштування збірки модуля
├── my_module_name.h            # Заголовок з класом модуля
├── my_module_name.cpp          # Реалізація модуля
├── my_module_name_api.h        # Публічне API модуля (опціонально)
├── my_module_name_events.h     # Визначення подій для EventBus (опціонально)
└── my_module_name_state.h      # Структури даних для SharedState (опціонально)
```

CMakeLists.txt

Файл налаштування збірки модуля. Приклад:

cmake

```cmake
# CMakeLists.txt для модуля my_module_name
idf_component_register(
    SRCS
        "my_module_name.cpp"
    INCLUDE_DIRS
        "."
    REQUIRES
        "modules/base_module"
        "core"
        # Інші залежності...
)
```

my_module_name.h

Заголовний файл з визначенням класу модуля, який наслідує `BaseModule`:

cpp

```cpp
/**
 * @file my_module_name.h
 * @brief Опис призначення модуля
 */

#ifndef MODULES_MY_MODULE_NAME_H
#define MODULES_MY_MODULE_NAME_H

#include "base_module.h"
// Інші необхідні включення...

class MyModuleNameModule : public BaseModule {
public:
    MyModuleNameModule();
    virtual ~MyModuleNameModule();

    // Базовий інтерфейс модуля (обов'язкові методи)
    const char* getName() const override;
    esp_err_t init() override;
    void tick() override;
    void stop() override;
    esp_err_t get_ui_schema(cJSON* module_schema_parent) override;

    // Публічні методи модуля...

private:
    // Приватні члени та методи...
};

#endif // MODULES_MY_MODULE_NAME_H
```

my_module_name.cpp

Файл реалізації модуля:

cpp

```cpp
/**
 * @file my_module_name.cpp
 * @brief Реалізація модуля
 */

#include "my_module_name.h"
#include "esp_log.h"
// Інші необхідні включення...

static const char* TAG = "MyModuleName";

// Конструктор
MyModuleNameModule::MyModuleNameModule() {
    // Ініціалізація членів
}

// Деструктор
MyModuleNameModule::~MyModuleNameModule() {
    // Звільнення ресурсів
}

// Отримання назви модуля
const char* MyModuleNameModule::getName() const {
    return "my_module_name";
}

// Ініціалізація модуля
esp_err_t MyModuleNameModule::init() {
    ESP_LOGI(TAG, "Ініціалізація модуля");

    // Завантаження конфігурації
    // Ініціалізація ресурсів
    // Підписка на події

    return ESP_OK;
}

// Періодичне оновлення модуля
void MyModuleNameModule::tick() {
    // Виконання періодичних операцій
}

// Зупинка модуля
void MyModuleNameModule::stop() {
    ESP_LOGI(TAG, "Зупинка модуля");
```

```cpp
    // Звільнення ресурсів
    // Скасування підписок на події
}

// Створення схеми UI для модуля
esp_err_t MyModuleNameModule::get_ui_schema(cJSON* module_schema_parent) {
    if (!module_schema_parent) {
        return ESP_ERR_INVALID_ARG;
    }

    // Створення схеми UI для веб-інтерфейсу

    return ESP_OK;
}

// Реалізація інших методів...
```

my_module_name_api.h (опціонально)

Файл з описом публічного API модуля:

cpp

```cpp
/**
 * @file my_module_name_api.h
 * @brief Публічний API модуля для використання іншими модулями
 */

#ifndef MODULES_MY_MODULE_NAME_API_H
#define MODULES_MY_MODULE_NAME_API_H

#include "esp_err.h"
// Інші необхідні включення...

namespace my_module_name_api {

// Структури даних API

// Функції API
esp_err_t some_function(parameters);
esp_err_t another_function(parameters);

} // namespace my_module_name_api

#endif // MODULES_MY_MODULE_NAME_API_H
```

my_module_name_events.h (опціонально)

Файл з описом подій, які генерує або на які підписується модуль:

```cpp
/**
 * @file my_module_name_events.h
 * @brief Визначення подій модуля
 */

#ifndef MODULES_MY_MODULE_NAME_EVENTS_H
#define MODULES_MY_MODULE_NAME_EVENTS_H

#include <string>
// Інші необхідні включення...

namespace my_module_name_events {

// Константи імен подій
static const char* const EVENT_SOMETHING_HAPPENED = "my_module.something_happened";

// Структури даних подій
struct SomethingHappenedEvent {
    // Поля події...
    uint64_t timestamp;
};

} // namespace my_module_name_events

#endif // MODULES_MY_MODULE_NAME_EVENTS_H
```

my_module_name_state.h (опціонально)

Файл з описом даних, які модуль зберігає в SharedState:

cpp

```cpp
/**
 * @file my_module_name_state.h
 * @brief Структури даних для стану модуля
 */

#ifndef MODULES_MY_MODULE_NAME_STATE_H
#define MODULES_MY_MODULE_NAME_STATE_H

#include <string>
// Інші необхідні включення...

namespace my_module_name_state {

// Ключі для SharedState
static const char* const KEY_SOME_VALUE = "my_module/some_value";

// Структури даних для стану
struct ModuleStatus {
    // Поля стану...
};

} // namespace my_module_name_state

#endif // MODULES_MY_MODULE_NAME_STATE_H
```

Основні класи та інтерфейси

BaseModule

Базовий клас для всіх модулів:

cpp

```cpp
class BaseModule {
public:
    virtual ~BaseModule() = default;
    virtual const char* getName() const = 0;
    virtual esp_err_t init() = 0;
    virtual void tick() = 0;
    virtual void stop() = 0;
    virtual esp_err_t get_ui_schema(cJSON* module_schema_parent) = 0;
};
```

Життєвий цикл модуля

1. **Конструювання**: Створення об'єкту модуля
2. **Реєстрація**: `ModuleManager::register_module(new MyModuleNameModule())`
3. **Ініціалізація**: `module->init()` (викликається з `ModuleManager::init_modules()`)
4. **Виконання**: `module->tick()` (викликається періодично з головного циклу)
5. **Зупинка**: `module->stop()` (викликається перед вимкненням або перезавантаженням)
6. **Деструкція**: Видалення об'єкту модуля (при завершенні роботи програми)

Взаємодія з ядром

ConfigLoader

Використовується для завантаження та збереження налаштувань:

```cpp
// Отримання значення
float temperature = ConfigLoader::get<float>("my_module/temperature", 4.0f); // 4.0f - значення

// Збереження значення
ConfigLoader::set<float>("my_module/temperature", 5.0f);
```

EventBus

Використовується для публікації подій та підписки на них:

```cpp
// Публікація події
my_module_name_events::SomethingHappenedEvent event = {
    .timestamp = static_cast<uint64_t>(time(nullptr))
};
EventBus::publish(my_module_name_events::EVENT_SOMETHING_HAPPENED, &event);

// Підписка на подію
EventBus::subscribe("SystemStarted", [this](const void* data) {
    ESP_LOGI(TAG, "Отримано подію SystemStarted");
    // Обробка події...
});
```

SharedState

Використовується для збереження стану та обміну даними:

cpp

```cpp
// Збереження значення
SharedState::set<float>(my_module_name_state::KEY_SOME_VALUE, 42.0f);

// Отримання значення
float value = SharedState::get<float>(my_module_name_state::KEY_SOME_VALUE, 0.0f);

// Підписка на зміну значення
SharedState::subscribe(my_module_name_state::KEY_SOME_VALUE, [this](const std::any& new_value)
    float value = std::any_cast<float>(new_value);
    ESP_LOGI(TAG, "Значення змінено на: %.1f", value);
    // Обробка зміни...
});
```

Взаємодія з іншими модулями

Через API модулів

cpp

```cpp
// Пошук модуля
OtherModule* other_module = static_cast<OtherModule*>(ModuleManager::find_module("other_module"

// Виклик методів модуля
if (other_module) {
    other_module->some_method();
}
```

Через EventBus

cpp

```cpp
// Публікація події, яка може бути оброблена іншими модулями
EventBus::publish("my_module.event", &data);

// Підписка на події інших модулів
EventBus::subscribe("other_module.event", [this](const void* data) {
    // Обробка події...
});
```

Через SharedState

```cpp
// Збереження даних, які можуть бути використані іншими модулями
SharedState::set<int>("my_module/value", 42);

// Отримання даних, збережених іншими модулями
int other_value = SharedState::get<int>("other_module/value", 0);
```

Рекомендації з розробки

Іменування

1. **Імена модулів**: нижній_регістр_з_підкресленнями (snake_case)
2. **Імена класів**: ВерхнійРегістрСлова (PascalCase)
3. **Імена методів**: нижнійРегістрСлова (camelCase) або нижній_регістр_з_підкресленнями
4. **Імена констант**: ВЕРХНІЙ_РЕГІСТР_З_ПІДКРЕСЛЕННЯМИ

Обробка помилок

1. **Методи ініціалізації** мають повертати `esp_err_t`
2. **Всі помилки** мають логуватися з відповідним рівнем важливості
3. **Критичні помилки** мають публікуватися через EventBus

Потокобезпечність

1. **Доступ до спільних ресурсів** має бути захищений мьютексами
2. **EventBus та SharedState** вже є потокобезпечними
3. **Довгі операції** мають виконуватися в окремих задачах FreeRTOS

Документація

1. **Всі публічні методи** мають бути задокументовані
2. **API модуля** має містити докладний опис функцій та структур
3. **Файл events.h** має описувати всі події, які генерує модуль

Інтеграція у збірку

Для додавання модуля у проект, потрібно виконати наступні кроки:

1. **Створити директорію** модуля в `modules/my_module_name/`
2. **Додати файли** модуля згідно з описаною структурою
3. **Зареєструвати модуль** у функції `register_all_modules()` в `main/main.cpp`:

cpp

```cpp
void register_all_modules() {
    ESP_LOGI(TAG, "Реєстрація модулів...");

    // Реєстрація існуючих модулів
    ModuleManager::register_module(new FridgeControllerModule());

    // Реєстрація нового модуля
    ModuleManager::register_module(new MyModuleNameModule());

    ESP_LOGI(TAG, "Модулі зареєстровано");
}
```

Шаблон модуля

Для спрощення створення нових модулів, можна використовувати наступний шаблон:

my_module_name.h

cpp

```cpp
/**
 * @file my_module_name.h
 * @brief Опис призначення модуля
 */

#ifndef MODULES_MY_MODULE_NAME_H
#define MODULES_MY_MODULE_NAME_H

#include "base_module.h"
// Інші необхідні включення...

class MyModuleNameModule : public BaseModule {
public:
    MyModuleNameModule();
    virtual ~MyModuleNameModule();

    // Базовий інтерфейс модуля (обов'язкові методи)
    const char* getName() const override;
    esp_err_t init() override;
    void tick() override;
    void stop() override;
    esp_err_t get_ui_schema(cJSON* module_schema_parent) override;

    // Публічні методи модуля...

private:
    // Приватні члени та методи...

    // Прапорець ініціалізації
    bool initialized_;
};

#endif // MODULES_MY_MODULE_NAME_H
```

my_module_name.cpp

cpp

```cpp
/**
 * @file my_module_name.cpp
 * @brief Реалізація модуля
 */

#include "my_module_name.h"
#include "esp_log.h"
// Інші необхідні включення...

static const char* TAG = "MyModuleName";

// Конструктор
MyModuleNameModule::MyModuleNameModule()
    : initialized_(false) {
    // Ініціалізація членів
}

// Деструктор
MyModuleNameModule::~MyModuleNameModule() {
    stop();
}

// Отримання назви модуля
const char* MyModuleNameModule::getName() const {
    return "my_module_name";
}

// Ініціалізація модуля
esp_err_t MyModuleNameModule::init() {
    ESP_LOGI(TAG, "Ініціалізація модуля");

    if (initialized_) {
        ESP_LOGW(TAG, "Модуль вже ініціалізовано");
        return ESP_OK;
    }

    // Завантаження конфігурації
    // Ініціалізація ресурсів
    // Підписка на події

    EventBus::subscribe("SystemStarted", [this](const void* data) {
        ESP_LOGI(TAG, "Отримано подію SystemStarted");
        // Обробка події...
    });

    initialized_ = true;
```

```
    ESP_LOGI(TAG, "Модуль успішно ініціалізовано");

    return ESP_OK;
}

// Періодичне оновлення модуля
void MyModuleNameModule::tick() {
    if (!initialized_) {
        return;
    }

    // Виконання періодичних операцій
}

// Зупинка модуля
void MyModuleNameModule::stop() {
    if (!initialized_) {
        return;
    }

    ESP_LOGI(TAG, "Зупинка модуля");

    // Звільнення ресурсів
    // Скасування підписок на події

    initialized_ = false;
}

// Створення схеми UI для модуля
esp_err_t MyModuleNameModule::get_ui_schema(cJSON* module_schema_parent) {
    if (!module_schema_parent) {
        return ESP_ERR_INVALID_ARG;
    }

    // Створення схеми UI для веб-інтерфейсу
    cJSON* module_obj = cJSON_CreateObject();
    if (!module_obj) {
        return ESP_ERR_NO_MEM;
    }

    // Додаємо метаінформацію
    cJSON_AddStringToObject(module_obj, "name", "Назва модуля");
    cJSON_AddStringToObject(module_obj, "description", "Опис модуля");

    // Додаємо об'єкт модуля до батьківського об'єкта
    cJSON_AddItemToObject(module_schema_parent, "my_module_name", module_obj);
```

```
    return ESP_OK;
}
```

CMakeLists.txt

```cmake
# CMakeLists.txt для модуля my_module_name

# Реєстрація компонента
idf_component_register(
    # Вихідні файли
    SRCS
        "my_module_name.cpp"

    # Публічні включення
    INCLUDE_DIRS
        "."

    # Залежності
    REQUIRES
        "modules/base_module"
        "core"
        # Інші залежності...
)
```

Цей шаблон можна використовувати як основу для створення нових модулів, змінюючи імена та реалізацію відповідно до потреб.