Báo cáo Kỹ thuật: Thiết kế Module SystemStateManager An toàn cho Bộ Điều khiển Robot Công nghiệp 6 Trục tuân thủ ISO 10218-1

1. Tổng quan Điều hành

Báo cáo này trình bày chi tiết thiết kế kiến trúc và chiến lược hiện thực hóa cho module SystemStateManager (Trình quản lý trạng thái hệ thống) nằm trong lõi C++ (C++ Core) của bộ điều khiển robot công nghiệp 6 trục. Mục tiêu cốt lõi của báo cáo là thiết lập một Máy trạng thái hữu hạn (Finite State Machine - FSM) mang tính quyết định (deterministic) và đảm bảo an toàn nghiêm ngặt (safety-critical), đóng vai trò là cơ quan quản lý trung tâm cho mọi hành vi vận hành của robot.

Thiết kế này được xây dựng dựa trên sự tuân thủ tuyệt đối các tiêu chuẩn quốc tế, đặc biệt là ISO 10218-1:2011 (Yêu cầu an toàn cho robot công nghiệp) và ISO 13849-1 (Các bộ phận liên quan đến an toàn của hệ thống điều khiển). Kiến trúc đề xuất loại bỏ các cấu trúc switch-case lồng nhau cứng nhắc, khó bảo trì để chuyển sang Mô hình Trạng thái dựa trên Bảng dữ liệu (Table-Driven State Pattern) sử dụng các tính năng của C++ hiện đại (C++17/20). Cách tiếp cận này không chỉ nâng cao khả năng kiểm toán (auditability) của mã nguồn—một yêu cầu bắt buộc trong các chứng nhận an toàn—mà còn đơn giản hóa việc kiểm thử đơn vị và đảm bảo thời gian phản hồi có thể dự đoán được đối với các sự kiện an toàn nguy cấp.

Báo cáo bao gồm việc tích hợp sâu rộng các khóa liên động an toàn (Safety Interlocks) như Dừng khẩn cấp (E-Stop), Màn chắn sáng (Light Curtains), Công tắc hành trình (Deadman Switches), quản lý các trạng thái chuyển động theo chuẩn PLCopen, và các chi tiết cài đặt C++ cụ thể để làm cầu nối giữa Giao diện người dùng (UI) trên PC và vòng lặp điều khiển thời gian thực.

2. Cơ sở Tiêu chuẩn và Yêu cầu An toàn Chức năng

Trước khi đi sâu vào thiết kế phần mềm, điều tối quan trọng là phải thiết lập khung pháp lý và kỹ thuật mà SystemStateManager phải tuân thủ. Robot công nghiệp 6 trục là thiết bị có mức độ rủi ro cao, có khả năng gây thương tích nghiêm trọng hoặc tử vong.[1] Do đó, phần mềm điều khiển không chỉ đơn thuần là thực thi lệnh mà còn là lớp bảo vệ cuối cùng.

2.1. ISO 10218-1: An toàn cho Robot Công nghiệp

Thiết kế của FSM bị ràng buộc chặt chẽ bởi ISO 10218-1, quy định cụ thể về các chế độ vận

hành và các danh mục dừng (Stop Categories).

2.1.1. Các Chế độ Vận hành (Operational Modes)

ISO 10218-1 yêu cầu robot phải có các chế độ vận hành riêng biệt với các hành vi an toàn loại trừ lẫn nhau. FSM phải thực thi nghiêm ngặt các trạng thái này, đảm bảo không bao giờ có sự nhầm lẫn giữa logic của chế độ Tự động và Chế độ Bằng tay.[2]

- **Chế độ Tự động (AUTO):** Robot thực hiện các tác vụ đã được lập trình sẵn mà không cần sự can thiệp của con người. Trong chế độ này, hàng rào an toàn và màn chắn sáng phải được kích hoạt. Bất kỳ sự vi phạm nào vào không gian an toàn (ví dụ: mở cửa lồng robot) phải kích hoạt Dừng bảo vệ (Protective Stop) ngay lập tức.[1] FSM phải từ chối mọi lệnh "Jog" (chạy nhấp) thủ công trong trạng thái này.
- **Chế độ Bằng tay Giảm tốc (Manual Reduced Speed - T1):** Được sử dụng cho việc dạy (teaching) và lập trình. Tốc độ tại Điểm Trung tâm Dụng cụ (Tool Center Point - TCP) phải được giới hạn cứng ở mức 250 mm/s. Người vận hành được phép vào trong vùng làm việc, nhưng chuyển động chỉ khả thi khi Thiết bị Cho phép (Enabling Device/Deadman Switch) được giữ ở vị trí trung gian.[1] FSM phải giám sát tín hiệu tốc độ và sẵn sàng ngắt động lực nếu vượt ngưỡng.
- **Chế độ Bằng tay Tốc độ Cao (Manual High Speed - T2):** Cho phép xác minh chương trình ở tốc độ đầy đủ. Đây là chế độ rủi ro cao nhất, yêu cầu thiết bị cho phép và thường cần một hành động xác nhận bổ sung từ người vận hành. FSM phải có các điều kiện bảo vệ (Guard Conditions) cực kỳ chặt chẽ cho chế độ này.[6]

2.1.2. Danh mục Dừng (Stop Categories theo IEC 60204-1)

FSM phải xử lý ba cơ chế dừng riêng biệt dựa trên mức độ nghiêm trọng của vi phạm an toàn [7]:

Danh mục Dừng	Mô tả Kỹ thuật	Hành vi của SystemStateManager	Ví dụ Kích hoạt
Category 0	**Dừng không kiểm soát (Uncontrolled Stop).** Ngắt ngay lập tức nguồn điện tới các bộ truyền động (actuators). Phanh cơ khí đóng lại ngay lập tức.	FSM chuyển ngay lập tức sang trạng thái ESTOP_ACTIVE. Bỏ qua mọi logic giảm tốc mềm. Gửi tín hiệu ngắt contactor.	Nhấn nút E-Stop; Mất điện nguồn điều khiển.

Category 1	**Dừng có kiểm soát (Controlled Stop).** Nguồn điện được duy trì để robot giảm tốc theo quỹ đạo dừng tối ưu, sau đó mới ngắt nguồn khi robot đã dừng hẳn.	FSM chuyển sang trạng thái STOPPING_CAT1. Gửi lệnh ramp-down velocity tới Motion Core. Giám sát vận tốc về 0, sau đó ngắt Enable Drives.	Vi phạm màn chắn sáng trong chế độ AUTO; Nhả công tắc Deadman trong chế độ MANUAL.
Category 2	**Dừng có kiểm soát, duy trì nguồn (Controlled Stop, Power Kept).** Nguồn điện được duy trì để giữ vị trí.	FSM chuyển sang trạng thái HOLD hoặc PAUSED. Các vòng lặp servo vẫn hoạt động để giữ robot tại vị trí, chống trôi.	Lệnh Tạm dừng (Pause) từ UI; Lệnh Wait trong kịch bản.

Dữ liệu nghiên cứu chỉ ra rằng việc nhầm lẫn giữa Cat 0 và Cat 1 là nguyên nhân phổ biến gây hư hỏng cơ khí cho robot (do phanh gấp ở tốc độ cao trong Cat 0) hoặc gây mất an toàn (do robot không dừng lại ngay khi mất điện trong Cat 1 giả lập sai). FSM phải phân định rạch ròi hai luồng xử lý này.

2.2. ISO 13849-1: Mức Hiệu suất (Performance Levels - PL)

Hệ thống điều khiển robot 6 trục thường yêu cầu đạt Mức Hiệu suất d (PLd) với Kiến trúc Loại 3 (Category 3).[9] Điều này đặt ra các yêu cầu cụ thể cho phần mềm FSM:

1. **Tính dư thừa (Redundancy):** Phần mềm không được tin tưởng một biến đơn lẻ cho các tín hiệu an toàn. FSM phải xử lý các cặp tín hiệu (Dual-channel monitoring). Ví dụ, tín hiệu E-Stop không phải là một bit bool eStop, mà là một cấu trúc kiểm tra sự nhất quán của hai kênh vật lý Input_A và Input_B.
2. **Phạm vi chẩn đoán (Diagnostic Coverage - DC):** Phần mềm phải phát hiện sự không khớp (mismatch) giữa hai kênh an toàn. Nếu Kênh A báo "Mở" nhưng Kênh B báo "Đóng" quá một khoảng thời gian cho phép (ví dụ: 50ms - Discrepancy Time), FSM phải chuyển sang trạng thái HARDWARE_FAULT và ngăn chặn mọi hoạt động khởi động lại.[12]
3. **Yêu cầu Reset thủ công:** ISO 13849-1 quy định rõ ràng rằng việc giải phóng thiết bị dừng khẩn cấp (ví dụ: xoay nút E-Stop ra) *không được phép* làm robot khởi động lại ngay lập tức. FSM phải yêu cầu một hành động Reset riêng biệt và có chủ ý từ người vận hành (ví dụ: nhấn nút "Reset" trên UI hoặc tủ điện) để chuyển từ trạng thái ESTOP_RECOVERABLE sang IDLE.[13]

2.3. Tích hợp Chuẩn PLCopen Motion Control

Để đảm bảo tính tương thích công nghiệp và dễ dàng tích hợp với các hệ thống PLC bên ngoài, logic trạng thái chuyển động bên trong FSM nên phản ánh Sơ đồ trạng thái của PLCopen.[15] SystemStateManager sẽ đóng vai trò là "Người giám sát" (Supervisor) cấp cao, quản lý các trạng thái an toàn và chế độ, trong khi một máy trạng thái con (Sub-FSM) hoặc Module Motion Planner sẽ quản lý chi tiết các trạng thái chuyển động như StandStill, DiscreteMotion, ContinuousMotion, ErrorStop. Sự phân tách này giúp mã nguồn C++ Core dễ bảo trì và mở rộng.

3. Kiến trúc Hệ thống và Luồng Dữ liệu

3.1. Phân chia Miền (Domain Separation)

Hệ thống điều khiển được chia thành hai miền riêng biệt, với SystemStateManager nằm tại giao điểm quan trọng nhất:

1. **Miền Phi Thời gian thực (Non-Real-Time Domain - PC/Windows):** Nơi chứa giao diện người dùng C# (WPF/WinForms). Nhiệm vụ của nó là hiển thị trạng thái, nhận lệnh từ người dùng, và trực quan hóa 3D. Nó *không* chịu trách nhiệm về an toàn máy móc. Nếu Windows bị treo, robot *phải* vẫn an toàn.
2. **Miền Thời gian thực (Real-Time Domain - C++ Core):** Chạy trên một nhân RTOS (như RTX64, INtime) hoặc một thread có độ ưu tiên cực cao trên Linux (Preempt-RT). Đây là nơi SystemStateManager cư trú. Nó phải đảm bảo chu kỳ điều khiển (cycle time) ổn định (thường là 1ms hoặc 0.5ms).[17]

3.2. Cơ chế Giao tiếp (IPC) và Xử lý Lệnh

Giao tiếp giữa C# UI và C++ Core thường sử dụng Shared Memory (Bộ nhớ chia sẻ) hoặc TCP/IP cục bộ để giảm độ trễ. Tuy nhiên, FSM không được xử lý lệnh một cách mù quáng.

- **Bộ đệm Lệnh (Command Buffer):** UI gửi các "Yêu cầu" (Requests), ví dụ: Req_SetMode(AUTO), Req_ServoOn, Req_JogAxis. Các yêu cầu này được đưa vào hàng đợi không khóa (lock-free queue) để tránh vi phạm thời gian thực.
- **Xử lý trong Chu kỳ Quét (Scan Cycle):** Tại mỗi chu kỳ 1ms, SystemStateManager sẽ kiểm tra hàng đợi lệnh.
- **Thẩm định Lệnh (Command Validation):** Đây là bước quan trọng. FSM sẽ so sánh lệnh yêu cầu với Trạng thái Hiện tại và các Khóa liên động. Ví dụ: Nếu nhận lệnh Req_JogAxis nhưng trạng thái hiện tại là AUTO, FSM sẽ từ chối lệnh và trả về mã lỗi cho UI, thay vì thực thi.[19]

4. Thiết kế Hệ thống Khóa liên động An toàn (Safety Interlock System)

Trước khi định nghĩa các trạng thái của FSM, ta phải định nghĩa các tín hiệu đầu vào điều khiển sự chuyển đổi trạng thái. Một module SafetySignalManager sẽ được thiết kế để trừu tượng hóa phần cứng I/O, cung cấp dữ liệu sạch cho FSM.

4.1. Danh sách Tín hiệu An toàn Phần cứng

Dựa trên phân tích các hệ thống robot công nghiệp tiêu chuẩn (Fanuc, ABB, Mitsubishi) [6], các tín hiệu sau là bắt buộc:

1. **Dừng Khẩn cấp (Emergency Stop - ES):** 2 kênh, tiếp điểm thường đóng (NC). Ngắt trực tiếp nguồn động lực qua Safety Relay/Contactor. FSM cần giám sát trạng thái phụ trợ của relay này để biết khi nào E-Stop đã được kích hoạt.
2. **Bảo vệ An toàn (Safeguard/Gate):** 2 kênh, NC. Thường là khóa cửa lồng hoặc màn chắn sáng. Tín hiệu này chỉ được phép bỏ qua (bypass) trong chế độ MANUAL (T1/T2) nhưng bắt buộc phải hoạt động trong chế độ AUTO.
3. **Thiết bị Cho phép (Enabling Device / Deadman Switch):** Công tắc 3 vị trí (Off-On-Off). Đây là "trái tim" của an toàn trong chế độ Bằng tay.
   - Vị trí 1 (Nhả ra): Off -> Ngắt chuyển động.
   - Vị trí 2 (Giữ giữa): On -> Cho phép chuyển động.
   - Vị trí 3 (Bóp chặt): Off (Panic) -> Ngắt chuyển động ngay lập tức (Cat 0 hoặc Cat 1).
4. **Bộ chọn Chế độ (Mode Selector):** Công tắc khóa hoặc RFID chọn T1, T2, AUTO. Tín hiệu này phải là duy nhất (không được phép cả T1 và AUTO cùng = 1).
5. **Phản hồi Servo (Servo Ready / Drive Fault):** Đọc từ EtherCAT Status Word của các servo driver.

4.2. Logic Xác thực Tín hiệu (Dual Channel Discrepancy Check)

Để đạt PLd, SystemStateManager không đọc trực tiếp chân I/O mà thông qua lớp xử lý logic dư thừa.

C++

```
// C++17 Implementation Snippet: Safety Signal Validation
struct SafetyInput {
    bool channel_A;
    bool channel_B;
    uint32_t mismatch_timer_ms;
    bool is_valid;
    bool logical_state; // State after validation
};
```

```
class SafetySignalManager {
public:
    // Chạy trong mỗi chu kỳ 1ms
    void Update(bool raw_A, bool raw_B, SafetyInput& input) {
        input.channel_A = raw_A;
        input.channel_B = raw_B;

        if (input.channel_A == input.channel_B) {
            input.mismatch_timer_ms = 0;
            input.is_valid = true;
            input.logical_state = input.channel_A; // Active Low or High based on config
        } else {
            input.mismatch_timer_ms++;
            // ISO 13849-1 cho phép sai lệch ngắn hạn (VD: do cơ khí công tắc không đồng đều)
            if (input.mismatch_timer_ms > MAX_DISCREPANCY_TIME_MS) { // e.g., 50ms
                input.is_valid = false;
                // Kích hoạt lỗi phần cứng nghiêm trọng
                FlagCriticalError(HardwareError::SAFETY_MISMATCH);
            }
        }
    }
};
```

Đoạn mã trên minh họa cách phần mềm loại bỏ nhiễu và phát hiện lỗi phần cứng tiềm ẩn (ví dụ: một dây E-Stop bị đứt hoặc công tắc bị dính tiếp điểm), tuân thủ nguyên tắc chẩn đoán của ISO 13849.

5. Thiết kế Chi tiết Máy trạng thái SystemStateManager

5.1. Tại sao chọn Table-Driven thay vì Switch-Case?

Trong phát triển hệ thống nhúng an toàn (safety-critical), việc sử dụng switch-case lồng nhau (nested switch-case) thường bị chỉ trích vì [22]:

1. **Độ phức tạp Cyclomatic cao:** Khó kiểm thử hết các nhánh.
2. **Khó bảo trì:** Logic chuyển đổi bị trộn lẫn với logic hành động.
3. **Thiếu tính trực quan:** Không thể hiện rõ ràng ma trận trạng thái-sự kiện.
4. **Rủi ro về ngăn xếp (Stack Usage):** Các khối lệnh lồng nhau sâu có thể gây tràn stack trong các vi điều khiển hạn chế tài nguyên.

Thay vào đó, thiết kế này sử dụng **Table-Driven Approach** kết hợp với **State Pattern** (tinh chỉnh cho C++ hiện đại, tránh cấp phát động new/delete trong runtime để tuân thủ MISRA

C++). Bảng chuyển đổi (Transition Table) hoạt động như một "cấu hình" cứng của hệ thống, giúp các kỹ sư an toàn dễ dàng kiểm toán (audit) logic mà không cần đọc từng dòng mã xử lý.

5.2. Sơ đồ Trạng thái (State Diagram)

Hệ thống được mô hình hóa theo cấu trúc phân cấp (Hierarchical State Machine - HSM) để quản lý độ phức tạp.

5.2.1. Các Trạng thái Chính (Major States)

1. **STATE_BOOT**: Khởi động hệ thống, kiểm tra toàn vẹn bộ nhớ (RAM/Flash), khởi tạo EtherCAT bus.
2. **STATE_ERROR_LOCKOUT**: Trạng thái lỗi nghiêm trọng (ví dụ: lỗi phần cứng an toàn, mất kết nối EtherCAT). Yêu cầu Power Cycle để thoát.
3. **STATE_ESTOP_ACTIVE**: E-Stop đang được nhấn. Nguồn động lực bị ngắt (Cat 0).
4. **STATE_ESTOP_RESET_NEEDED**: E-Stop đã được nhả, nhưng hệ thống chờ xác nhận Reset từ người dùng.[13]
5. **STATE_IDLE**: Hệ thống không lỗi, nguồn điều khiển có, nhưng chưa cấp nguồn động lực (Servo Off). Brakes đang đóng.
6. **STATE_ARMING**: Quá trình chuyển tiếp. Kiểm tra interlocks -> Đóng Contactor nguồn -> Bật Servo -> Mở phanh -> Chờ phản hồi "Ready".
7. **STATE_OPERATIONAL**: Robot đã sẵn sàng, Servo On, Phanh mở. Đây là trạng thái cha (Super-state) của các chế độ vận hành.
    - **SUBSTATE_AUTO_IDLE**: Chờ lệnh chạy chương trình.
    - **SUBSTATE_AUTO_RUNNING**: Đang thực thi G-code/Script.
    - **SUBSTATE_MANUAL_IDLE**: Chờ lệnh Jog.
    - **SUBSTATE_MANUAL_JOGGING**: Đang di chuyển theo lệnh Jog của người dùng (yêu cầu Deadman).
8. **STATE_STOPPING**: Đang thực hiện quy trình dừng (Cat 1 hoặc Cat 2). Chuyển tiếp về IDLE hoặc ESTOP sau khi hoàn tất.

5.3. Ma trận Chuyển đổi và Điều kiện Bảo vệ (Transitions & Guards)

Dưới đây là bảng mô tả logic cốt lõi của FSM, ánh xạ sự kiện đầu vào tới trạng thái kế tiếp:

Trạng thái Hiện tại	Sự kiện (Event)	Điều kiện Bảo vệ (Guard)	Trạng thái Kế tiếp	Hành động (Action)
ANY	EV_ESTOP_ASSERTED	None	ESTOP_ACTIVE	Ngắt PWM, Ngắt Contactor, Khóa phanh

ESTOP_ACTIVE	EV_ESTOP_RELEASED	Safety_Mismatch == False	ESTOP_RESET_NEEDED	Bật đèn báo "Reset Required"
ESTOP_RESET_NEEDED	CMD_SYS_RESET	None	IDLE	Xóa lỗi phần mềm
IDLE	CMD_SERVO_ON	`No_Faults && (Mode==AUTO		
Deadman==ON)`	ARMING	Kích hoạt tuần tự nguồn		
ARMING	EV_SERVO_READY	None	OPERATIONAL	Log "System Ready"
OPERATIONAL	CMD_START_AUTO	Mode==AUTO && Gate==CLOSED	AUTO_RUNNING	Bắt đầu nội suy quỹ đạo
AUTO_RUNNING	EV_GATE_OPEN	None	STOPPING	Dừng Cat 1 (Ramp down)
OPERATIONAL	CMD_JOG_START	Mode==MANUAL && Deadman==ON	MANUAL_JOGGING	Bắt đầu nội suy Jog
MANUAL_JOGGING	EV_DEADMAN_OFF	None	STOPPING	Dừng Cat 1 nhanh

6. Hiện thực hóa bằng C++ Hiện đại (Modern C++ Implementation)

Chúng ta sẽ sử dụng C++17 với std::variant để biểu diễn các trạng thái (cho phép mỗi trạng thái mang dữ liệu riêng, ví dụ trạng thái lỗi mang mã lỗi) và một bảng chuyển đổi constexpr nếu có thể, hoặc một cấu trúc dữ liệu tĩnh (static const).

6.1. Định nghĩa Sự kiện và Trạng thái

C++

```cpp
#include <variant>
#include <optional>
#include <functional>
#include <iostream>

// Định nghĩa các Sự kiện (Events)
enum class EventType {
    ESTOP_TRIGGERED,
    ESTOP_RELEASED,
    CMD_RESET,
    CMD_SERVO_ON,
    CMD_SERVO_OFF,
    HARDWARE_FAULT,
    SAFETY_GATE_OPENED,
    DEADMAN_PRESSED,
    DEADMAN_RELEASED,
    PROCESS_COMPLETE // Sự kiện nội bộ khi hoàn thành quy trình (VD: Stopping xong)
};

struct Event {
    EventType type;
    // Có thể thêm dữ liệu sự kiện (payload) nếu cần
};

// Định nghĩa các Trạng thái (States) sử dụng struct để dễ mở rộng
struct StateBoot {};
struct StateEstopActive { uint64_t timestamp; };
struct StateEstopResetNeeded {};
struct StateIdle {};
struct StateArming { int progress_percent; };
struct StateOperational {
    enum class SubMode { AUTO, MANUAL_T1, MANUAL_T2 } current_mode;
};
struct StateStopping {
    enum class Category { CAT0, CAT1, CAT2 } stop_type;
};
struct StateError { int error_code; };
```

```
// Variant chứa tất cả các trạng thái có thể
using SystemState = std::variant<
    StateBoot,
    StateEstopActive,
    StateEstopResetNeeded,
    StateIdle,
    StateArming,
    StateOperational,
    StateStopping,
    StateError
>;
```

6.2. Cấu trúc Bảng Chuyển đổi (Transition Table)

Để tránh switch-case, ta định nghĩa một cấu trúc bảng. Trong môi trường nhúng, ta ưu tiên std::function hoặc con trỏ hàm trần (raw function pointers) để tránh overhead.

C++

```
// Alias cho hàm Guard và Action
using GuardFunc = std::function<bool(const SystemState&)>;
using ActionFunc = std::function<void(SystemState&)>;

struct Transition {
    // Chúng ta cần một cách để định danh kiểu trạng thái trong bảng
    // Một cách đơn giản là dùng index của variant hoặc enum song song
    size_t src_state_index;
    EventType trigger_event;

    // Hàm tạo trạng thái mới (Factory function)
    std::function<SystemState(const SystemState&)> next_state_factory;

    GuardFunc guard;
    ActionFunc action;
};
```

Tuy nhiên, việc ánh xạ std::variant vào bảng tra cứu phẳng hơi phức tạp. Một mẫu thiết kế (Design Pattern) hiệu quả hơn cho C++17 là **Overloaded Visitor** kết hợp với bảng tra cứu cục bộ hoặc sử dụng một thư viện FSM siêu nhẹ (header-only) tuân thủ chuẩn. Nhưng để tuân thủ

yêu cầu "tự thiết kế", ta sẽ xây dựng một bộ xử lý sự kiện (Event Handler) sử dụng std::visit.

6.3. Triển khai SystemStateManager với std::visit

Cách tiếp cận này tận dụng tính năng "Pattern Matching" của C++17, cho phép trình biên dịch kiểm tra tính đầy đủ của các trường hợp và tối ưu hóa tốt hơn switch-case thủ công.

C++

```
class SystemStateManager {
private:
    SystemState current_state = StateBoot{};
    SafetySignalManager& safety_manager;

public:
    SystemStateManager(SafetySignalManager& mgr) : safety_manager(mgr) {}

    void ProcessEvent(const Event& event) {
        // Xử lý Sự kiện Toàn cục (Global Transitions) - Ưu tiên cao nhất
        if (event.type == EventType::ESTOP_TRIGGERED) {
            if (!std::holds_alternative<StateEstopActive>(current_state)) {
                TransitionTo(StateEstopActive{ GetTimestamp() });
                // Thực hiện hành động Cat 0 Stop ngay lập tức
                Hardware::CutPower();
            }
            return;
        }

        // Dispatch sự kiện dựa trên trạng thái hiện tại
        std::visit([this, &event](auto&& state) {
            this->HandleStateEvent(state, event);
        }, current_state);
    }

private:
    // --- Các bộ xử lý cho từng trạng thái cụ thể ---

    // 1. Xử lý cho StateIdle
    void HandleStateEvent(const StateIdle&, const Event& event) {
        if (event.type == EventType::CMD_SERVO_ON) {
```

```
        if (safety_manager.IsSystemSafe()) { // Guard Condition
            TransitionTo(StateArming{0});
            Hardware::EnableContactors(); // Action
        } else {
            LogWarning("Servo On Rejected: Safety Interlocks Not Met");
        }
    }
}

// 2. Xử lý cho StateArming
void HandleStateEvent(const StateArming& state, const Event& event) {
    if (event.type == EventType::PROCESS_COMPLETE) {
        // Giả sử process complete nghĩa là servo đã ready
        TransitionTo(StateOperational{ DetectMode() });
    }
    else if (event.type == EventType::HARDWARE_FAULT) {
        TransitionTo(StateError{ 101 }); // Error 101: Arming Failed
    }
}

// 3. Xử lý cho StateOperational (Trạng thái phức tạp nhất)
void HandleStateEvent(const StateOperational& state, const Event& event) {
    // Kiểm tra an toàn liên tục
    if (state.current_mode == StateOperational::SubMode::AUTO) {
        if (event.type == EventType::SAFETY_GATE_OPENED) {
            TransitionTo(StateStopping{ StateStopping::Category::CAT1 });
            MotionCore::InitiateRampDown();
            return;
        }
    }
    else if (state.current_mode == StateOperational::SubMode::MANUAL_T1) {
        if (event.type == EventType::DEADMAN_RELEASED) {
            TransitionTo(StateStopping{ StateStopping::Category::CAT1 });
            MotionCore::InitiateQuickStop();
            return;
        }
    }

    // Xử lý lệnh di chuyển...
}

// 4. Xử lý cho StateEstopActive
void HandleStateEvent(const StateEstopActive&, const Event& event) {
```

```
        if (event.type == EventType::ESTOP_RELEASED) {
            // Không được tự động về IDLE, phải về RESET_NEEDED
            TransitionTo(StateEstopResetNeeded{});
        }
    }

    // 5. Xử lý cho StateEstopResetNeeded
    void HandleStateEvent(const StateEstopResetNeeded&, const Event& event) {
        if (event.type == EventType::CMD_RESET) {
            if (safety_manager.IsEStopCleared()) {
                TransitionTo(StateIdle{});
                LogInfo("System Reset by Operator");
            }
        }
    }

    // Default handler cho các trạng thái không quan tâm sự kiện này
    template <typename T>
    void HandleStateEvent(const T&, const Event&) {
        // Ignore event
    }

    // Hàm chuyển đổi trạng thái chung
    template <typename NewState>
    void TransitionTo(NewState&& new_state) {
        // Có thể thêm logic Exit() cho trạng thái cũ và Entry() cho trạng thái mới ở đây
        current_state = std::forward<NewState>(new_state);
        // Gửi thông báo cập nhật trạng thái lên UI qua IPC
        NotifyUI(current_state);
    }

    StateOperational::SubMode DetectMode() {
        // Đọc từ phần cứng
        return safety_manager.GetMode();
    }
};
```

6.4. Phân tích Ưu điểm của Thiết kế C++ này

1. **Type Safety (An toàn kiểu dữ liệu):** std::variant đảm bảo rằng current_state luôn chứa một và chỉ một trạng thái hợp lệ. Không có rủi ro con trỏ null hay trạng thái rác (undefined state).
2. **Tránh Switch-Case Lồng nhau:** Thay vì một hàm khổng lồ với 2 tầng switch (Switch

State -> Switch Event), ta dùng std::visit để định tuyến (dispatch) trực tiếp đến hàm xử lý (HandleStateEvent) tương ứng với kiểu dữ liệu của trạng thái hiện tại. Điều này chia nhỏ code thành các hàm nhỏ, dễ đọc và dễ test.

3. **Encapsulation (Tính đóng gói):** Dữ liệu riêng của từng trạng thái (ví dụ timestamp trong EstopActive hay progress trong Arming) được gói gọn trong struct của trạng thái đó, không làm ô nhiễm không gian biến toàn cục của class.
4. **Khả năng mở rộng:** Thêm một trạng thái mới chỉ cần định nghĩa struct mới, thêm vào std::variant, và viết hàm overload HandleStateEvent. Trình biên dịch sẽ báo lỗi nếu ta quên xử lý trạng thái đó trong std::visit (nếu không dùng template default).

7. Tích hợp và Kiểm thử Tuân thủ (Verification & Validation)

7.1. Giao tiếp với Motion Core và HAL

SystemStateManager không trực tiếp điều khiển động cơ. Nó hoạt động như một nhạc trưởng.

- Khi chuyển sang StateStopping, nó gọi MotionCore::InitiateRampDown().
- Nó liên tục truy vấn MotionCore::GetVelocity() để xác nhận robot đã dừng hẳn trước khi chuyển từ Stop Cat 1 sang ngắt nguồn (để tối ưu hóa tuổi thọ phanh cơ khí).

7.2. Kiểm thử Tuân thủ MISRA C++ và ISO

Để đảm bảo mã nguồn này an toàn cho robot công nghiệp:

- **Không cấp phát động:** std::variant (trong hầu hết các cài đặt thư viện chuẩn hiện đại như libstdc++ hoặc libc++) thường không sử dụng heap cho các kiểu dữ liệu nhỏ (Small Object Optimization). Tuy nhiên, để tuân thủ nghiêm ngặt MISRA trong môi trường nhúng không có heap, ta có thể cần sử dụng một phiên bản variant tùy chỉnh hoặc etl::variant (Embedded Template Library) để đảm bảo 100% stack-based.
- **Kiểm tra độ bao phủ (Coverage Testing):** Mọi dòng code trong các hàm HandleStateEvent phải được cover 100% (MC/DC coverage) trong unit test.
- **Fault Injection:** Cần mô phỏng các tình huống như: Tín hiệu E-Stop chập chờn (bouncing), người dùng nhấn Reset khi E-Stop chưa nhả, hoặc chuyển mode khi robot đang chạy tốc độ cao. Hệ thống phải luôn rơi vào trạng thái an toàn (STOPPING hoặc ESTOP).

8. Kết luận

Thiết kế SystemStateManager được trình bày ở trên cung cấp một nền tảng vững chắc cho việc phát triển bộ điều khiển robot 6 trục an toàn và tin cậy. Bằng cách kết hợp các nguyên tắc an toàn nghiêm ngặt của ISO 10218-1/ISO 13849-1 với sức mạnh diễn đạt và an toàn kiểu của C++ hiện đại, chúng ta loại bỏ được sự mong manh của các phương pháp lập trình truyền thống. Kiến trúc này không chỉ đáp ứng yêu cầu chức năng hiện tại mà còn tạo điều kiện

thuận lợi cho việc mở rộng tính năng (như thêm Cobot mode - ISO/TS 15066) và vượt qua các bài kiểm tra chứng nhận an toàn khắt khe trong tương lai. Bước tiếp theo là triển khai POC (Proof of Concept) trên phần cứng thực tế và thực hiện đo đạc thời gian phản hồi (response time analysis) để đảm bảo các ràng buộc thời gian thực được thỏa mãn.

Works cited

1. ISO 10218: Ensuring Safety in Industrial Robotics - Jama Software, accessed February 1, 2026, https://www.jamasoftware.com/requirements-management-guide/industrial-manufacturing-development/iso-10218-ensuring-safety-in-industrial-robotics/
2. The New ISO 10218:2023 Standards: Enhancing Robotic Safety in Industrial Environments, accessed February 1, 2026, https://www.liveelectronicsgroup.com/technical-news/enhancing-robot-safety/
3. CR800 series controller Robot Safety Option Instruction Manual - Mitsubishi Electric, accessed February 1, 2026, https://www.mitsubishielectric.com/dl/fa/document/manual/robot/bfp-a3531/bfp-a3531p.pdf
4. Safety Connection | New ISO 10218:2025: Industrial Robots - YouTube, accessed February 1, 2026, https://www.youtube.com/watch?v=IWS1ifmb32U
5. OSHA Technical Manual (OTM) - Section IV: Chapter 4 | Occupational Safety and Health Administration, accessed February 1, 2026, https://www.osha.gov/otm/section-4-safety-hazards/chapter-4
6. FANUC Robot SAFETY HANDBOOK, accessed February 1, 2026, https://www.fanuc.eu/~/media/files/pdf/products/robots/educational%20cell/safety%20manual%20for%20fanuc%20educational%20cell.pdf?la=en
7. Product specification - Robot stopping distances according to ISO 10218-1 - ABB, accessed February 1, 2026, https://search.abb.com/library/Download.aspx?DocumentID=3HAC048645-001&LanguageCode=en&DocumentPartId=&Action=Launch
8. Universal Robots Top 10 Frequently Asked Questions - CrossCo, accessed February 1, 2026, https://www.crossco.com/resources/technical/robotics-faq/
9. Safety FAQ - Universal Robots, accessed February 1, 2026, https://www.universal-robots.com/articles/ur/safety/safety-faq/
10. 5.3 - Safety Functions - Gt-Engineering, accessed February 1, 2026, https://www.gt-engineering.it/en/technical-standards/en-iso-standards/en-iso-10218-1-safety-requirements-for-industrial-robots/5-4-safety-related-control-system-performance/
11. An Introduction to Machine Safety Standard ISO 13849 - EZ Spotlight - EngineerZone, accessed February 1, 2026, https://ez.analog.com/ez-blogs/b/engineerzone-spotlight/posts/an-introduction-to-machine-safety-standard-iso-13849
12. CR800 series controller Robot Safety Option Instruction Manual - Contentstack, accessed February 1, 2026, https://eu-assets.contentstack.com/v3/assets/blt5412ff9af9aef77f/bltc6e2fc1cdfff

fb19/61f64ee26354aa63b7f51752/3bed1944-b3bc-11eb-91e1-b8ca3a62a094_bfp-a3531d.pdf
13. Using an HMI for reset and start - Safety Products - ABB, accessed February 1, 2026, https://new.abb.com/low-voltage/products/safety-products/using-an-hmi-for-reset-and-start
14. Implementation of safety requirements from applicable standards - Schneider Electric, accessed February 1, 2026, https://product-help.schneider-electric.com/Machine%20Expert/V1.1/en/Preventa_SafeMotion_FBFUN/topics/safetyrequirements_motionfb.htm
15. PLCopen State Diagram - Schneider Electric, accessed February 1, 2026, https://product-help.schneider-electric.com/Machine%20Expert/V1.1/en/MotCoLib/MotCoLib/General_Description_of_Motion_Control_Libraries/General_Description_of_Motion_Control_Libraries-4.htm
16. PLCopen State Machine, accessed February 1, 2026, https://webhelp.kollmorgen.com/kas3.07/Content/3.UnderstandKAS/PLCopen%20state%20machine.htm
17. Design Patterns for Safety-Critical Embedded Systems - RWTH Publications, accessed February 1, 2026, https://publications.rwth-aachen.de/record/51773/files/3273.pdf
18. Implementing a Real-Time State Machine in Modern C++ - honeytreeLabs, accessed February 1, 2026, https://honeytreelabs.com/posts/real-time-state-machine-in-cpp/
19. Functional safety of machine controls – Application of EN ISO 13849 – (IFA Report 2/2017e) - DGUV, accessed February 1, 2026, https://www.dguv.de/medien/ifa/en/pub/rep/pdf/reports-2019/report0217e/rep0217e.pdf
20. SAFETY MANUAL - Mitsubishi Electric, accessed February 1, 2026, https://www.mitsubishielectric.com/dl/fa/document/manual/robot/bfp-a3541/bfp-a3541k.pdf
21. MSEL Controller - Instruction Manual Fifth Edition - ATB Automation, accessed February 1, 2026, https://atbautomation.eu/uploads/IAI_MSEL_PG_24V-stepper_vier-assige-controller_handleiding.pdf
22. How not to code a state machine in C++ for an embedded system, accessed February 1, 2026, https://fjrg76.wordpress.com/2020/05/14/how-not-to-code-a-state-machine-in-c-for-an-embedded-system/
23. What is your way to go for state machines in C++? : r/embedded - Reddit, accessed February 1, 2026, https://www.reddit.com/r/embedded/comments/10o3i2x/what_is_your_way_to_go_for_state_machines_in_c/
24. C state-machine design [closed] - Stack Overflow, accessed February 1, 2026, https://stackoverflow.com/questions/1647631/c-state-machine-design