

# 자동차 제어기 소프트웨어의 안전성 향상을 위한 Rust 언어 적용 및 신뢰성 검증

Evaluation of Rust for Automotive Embedded Controller Software

오혜빈 | 세온이앤에스



세온이앤에스  
Engineering and Solutions

# 1. 서론: 연구 배경 및 목적

## C/C++ 기반 제어기의 취약점

- SDV 전환으로 인한 SW 복잡도 기하급수적 증가
- 수동 메모리 관리에 기인한 **정의되지 않은 동작(UB)** 상시 노출

### ⚠ 임베디드 환경의 주요 UB 취약점

  
버퍼  
오버플로우

  
데이터  
결함

  
널 포인터  
역참조

→ 인접 메모리를 조용히 오염시켜 치명적 오작동 유발

- ISO 26262 ASIL-D에서 요구하는 체계적 결함 통제 난관

## Rust 언어의 도입 타당성

- **Safe by Design**: 강력한 타입 시스템을 통해  
**런타임 UB 발생을 컴파일 타임에 원천 차단**
- 가비지 컬렉터(GC) 부재로 인한 **실시간 제어 결정성** 확보
- 베어메탈(no\_std) 환경에서의 적용 가능성 및 성능 오버헤드 실증  
필요성



**"If it compiles, it is safe"**

빌드 통과 시 좌측의 치명적 메모리 결함(UB)이  
없음을 언어 스펙 차원에서 **구조적으로 보장함**

# 1.1 기존 인증 컴파일러 및 가이드라인의 한계

## 사후 검증(Post-coding check)의 맹점

- ◆Green Hills, ARM Compiler 등 기능안전 인증 컴파일러의 보편화
- ◆MISRA C 2012 / CERT C 가이드라인 적용에도 불구하고, **ISO C 표준 규격상** 발생하는 배열의 포인터 암시적 변환 (Decay) 제어 불가
- ◆외부 정적 분석 도구 의존에 따른 오탐지 해소 비용 및 인적 오류 발생 가능성 잔존

## 언어 패러다임의 전환 필요성

"도구(Tool)를 통한 결함의 발견이 아닌,  
언어 스펙(Specification)을 통한 결함의 원천 차단"

→ 컴파일러 자체 규칙이 정적 분석기를 대체하는  
**Rust의 강점**

## 2.1 메모리 안전성: 소유권(Ownership)

### 결정론적(Deterministic) 자원 관리

- **소유권 규칙:** 값의 소유자(Owner)는 특정 스코프 내 단 하나만 존재
- **자동 해제(Drop):** 스코프 이탈 시, 가비지 컬렉터(GC)의 비결정적 지연 없이 **예측 가능한 시점에** 메모리 반환
- **빌림 검사(Borrow Checker):** 다중 가변 참조(&mut) 금지로 댕글링 포인터 및 Aliasing 원천 차단

### 소유권 이동을 통한 자원 이중 사용 차단

```
// Rust: 소유권 이동(Move) 기반 버퍼 재사용 차단

fn can_drv_transmit(tx_buf: TxBuf) {
    // CAN HW Mailbox 전송 처리
}

fn can_tx_task() {

    let tx_buf = TxBuf::new();

    // 소유권이 Driver 계층으로 이동
    can_drv_transmit(tx_buf);

    // can_update_payload(&mut tx_buf);
    // ✖ 컴파일 에러!
    // 이미 이동된 버퍼 접근 불가
}
```

전송 이후 버퍼 재사용, 중복 수정, Alias 접근을  
컴파일 단계에서 구조적으로 차단

## 2.2 엄격한 타입 시스템: 배열 퇴화 방지

### Fat Pointer 기반 Sized Type

- C 언어의 배열은 함수 인자로 전달 시 크기 정보를 상실하고 메모리 주소(8바이트)만 남는 퇴화 현상 발생
- Rust의 슬라이스(&T)는 데이터 포인터와 길이(Length) 정보를 함께 지니는 Fat Pointer 구조 채택
- 컴파일 및 런타임에 크기 불일치 연산을 원천 거부

### 함수 시그니처 레벨의 안전성 보장

```
// C: 배열 크기 상실로 인한 오버플로우 위험
void process(uint8_t *arr) {
    arr[10] = 0xFF; // 크기를 몰라 컴파일 통과
}
```

```
// Rust: 크기 정보 보존 (Fat Pointer)
fn process(arr: &[u8; 8]) {
    arr[10] = 0xFF; // 컴파일 타임 에러 (크기 초과)
}
```

## 2.3 대수적 데이터 타입(ADT) 적용

### Null 포인터 역참조(Dereference) 방어

- ◆ 하드웨어 수신 버퍼(FIFO) Polling 시, 데이터 부재를 알리기 위해 C는 Null 반환
- ◆ Null 체크 누락 시 발생하는 런타임 크래시는 차량 제어기의 치명적 결함 유발
- ◆ Rust는 Null 개념을 배제하고 Option<T> 열거형(Enum) 도입

### 컴파일러의 패턴 매칭 강제

```
// Rust: 직접 구현한 CAN Interface 계층 예시
fn canif_rx_indication(rx_pdu: Option<CanFrame>) {
    match rx_pdu {
        Some(frame) => {
            canif_dispatch_rx_frame(frame);
        }
        None => {
            canif_handle_rx_empty();
        }
    }
} // frame 스코프 종료 시 자동 정리
```

C언어의 런타임 Null 참조 크래시를 컴파일러의 패턴 매칭으로  
원천 통제함

## 2.4 트레이트 기반 동시성 제어

### 데이터 경합(Data Race) 컴파일 타임 차단

- ◆ **Send Trait:** 스레드/인터럽트 간 소유권을 안전하게 이전할 수 있는 타입 명시
- ◆ **Sync Trait:** 여러 실행 흐름 간 안전하게 참조를 공유할 수 있는 타입 명시
- ◆ 공유 자원 접근 시, 개발자의 Mutex 및 **임계 구역(Critical Section)** 클로저 사용을 강제함

### 타입 시스템에 의한 안전 공유 모델 검증

```
// Send/Sync 검증 실패 예시
static mut COUNTER: u32 = 0;
// unsafe 블록 밖에서 접근 시 컴파일 에러 발생
// → 임의의 비동기적 접근 원천 봉쇄
```

임의의 비동기적 접근 원천 봉쇄

## 2.5 C언어 volatile 키워드의 구조적 한계

### Read-Modify-Write (RMW) 취약성

- ◆ C의 volatile은 컴파일러의 레지스터 캐싱 최적화만 제한할 뿐, 메모리 연산의 **원자성(Atomicity)** 보장 불가
- ◆ count++와 같은 RMW 연산 도중 인터럽트 선점(Preemption) 발생 시 데이터 훼손(Corruption)에 가능

### 상호 배제(Mutual Exclusion) 강제화

- ◆ Rust는 Send/Sync 트레이트를 통해 자원 자체에 락(Lock) 메커니즘이 결합되어야만 컴파일을 허용
- ◆ 개발자의 임계 구역 캡슐화 누락이라는 인적 오류를 문법 레벨에서 통제

## 2.6 비용 없는 추상화 (Zero-cost Abstraction)

### 고수준 문법의 로우레벨 성능 변환

- ◆ **단상화(Monomorphization):** 제네릭 타입을 컴파일 시점에 구체적 타입의 코드로 복제하여 런타임 다형성(vtable) 지연 배제
- ◆ **LLVM IR 최적화:** 고수준 이터레이터 체인이 C언어의 수동 `for` 루프와 완벽히 동등한 어셈블리어로 인라인화(Inlining)
- ◆ 가비지 컬렉터(GC) 부재로 실시간 제어의 핵심인 **실행 타이밍 결정성** 완벽 보장

### Zero-cost Abstraction의 공학적 의미

"고수준 추상화(안전성) 계층을 추가하더라도,  
C언어 수동 최적화 대비 추가적인 런타임 성능 손실(Additional Overhead)은 0이다"

## 2.7 Unsafe 블록 격리

### 신뢰성 기반 아키텍처 한정

- MCU의 하드웨어 레지스터(MMIO) 제어, FFI(Foreign Function Interface) 호출 등 필수적인 로우레벨 접근 허용
- 접근 영역을 `unsafe` 블록 안으로 명시적으로 **격리(Isolation)** 및 **래핑(Wrapping)**
- ISO 26262 준수를 위한 코드 리뷰 및 동적/정적 검증 범위를 `unsafe` 블록 내부로 대폭 축소

### 레지스터 제어의 안전한 추상화 래핑

```
// 레지스터 제어의 안전한 추상화 래핑
pub fn enable_can() {
    // 안전 구역: 비즈니스 로직
    unsafe {
        // 격리 구역: 하드웨어 직접 조작
        ptr::write_volatile(CAN_MCR, 0x1);
    }
}
```

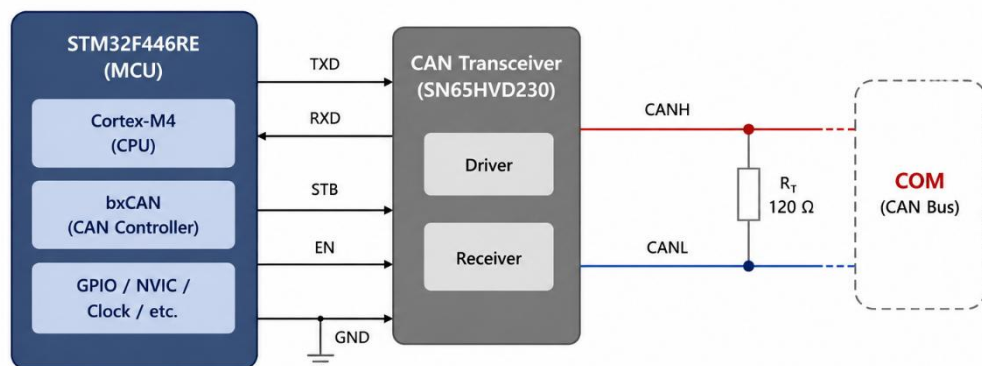
### 3. 실험 환경 및 시스템 구성

#### 하드웨어 제어 환경

- **Target MCU:** STM32F446RE (ARM Cortex-M4)
- **클럭 동기화:** 16MHz HSI 고정 (Jitter 유발 PLL 배제)
- **성능 측정:** DWT 레지스터 기반 CPU 사이클 정밀 측정

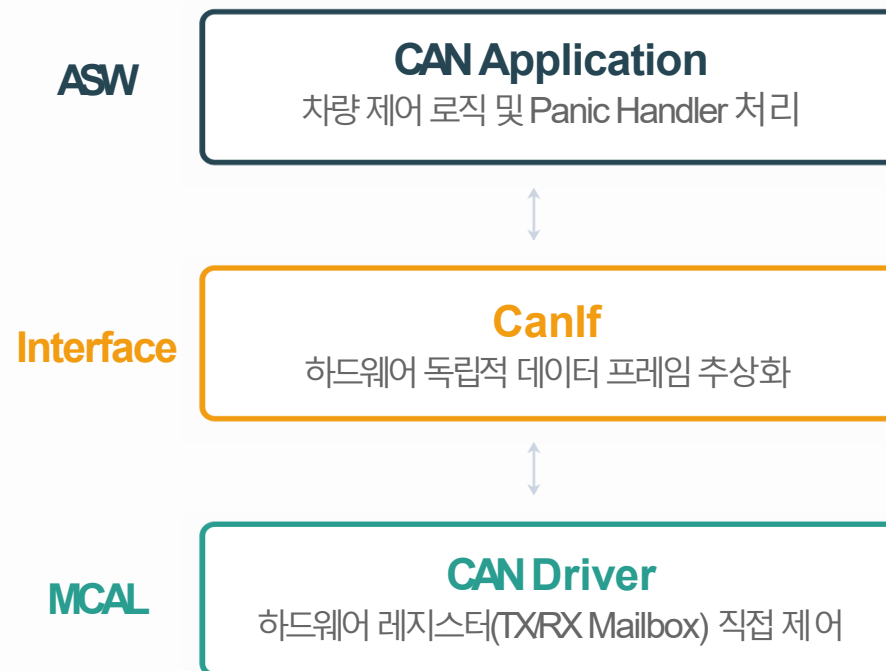
#### Hardware Configuration

STM32F446RE – CAN Transceiver (SN65HVD230) – COM



#### CAN 소프트웨어 스택 구현

"Rust no\_std 기반 CAN Full-Stack 수직 구현 완료"



## 3.1 정밀 측정 및 빌드 변인 통제

### DWT 레지스터 기반 사이클 정밀 측정

- ◆ 소프트웨어 타이머의 오차를 배제하기 위해 MCU 내장 디버그 유닛 (DWT) 활용
- ◆ DWT\_CYCNT 레지스터(0xE0001004)를 직접 폴링하여 **CPU 사이클 단위(1/16 $\mu$ s)**의 정밀한 실행 타이밍 프로파일링

### 성능 최적화 빌드 플래그

- ◆ lto = true: 링크 타임 최적화로 함수 간 인라이닝 극대화
- ◆ opt-level = 2: 임베디드 환경에 맞춘 최적 성능 컴파일
- ◆ panic = abort: 에러 발생 시 스택 되감기(Unwinding) 동작을 차단하여 비결정적 지연 방지

## 4. 실증: 메모리 결함 정적 보호

### Vulnerable (C/C++)

#### 배열 퇴화 및 경계 초과 접근

```
void process(uint8_t *ptr) {  
    ptr[8] = 0xAA; // Overflow!  
}
```

결과: 빌드 통과 후 런타임 메모리 훼손 유발



### Guaranteed (Rust)

#### 컴파일 타임 정적 무결성 검증

```
error: index out of bounds  
| data[8] = 0xAA;  
| ^^^^^^ index 8 > size 8
```

결과: 바이너리 생성 자체를 거부 (Shift-Left)

수 시간 소요되는 런타임 결함 디버깅 비용을 0초(컴파일 타임)로 단축시킴

\* 고가의 외부 정적 분석 도구 없이, 언어 스펙만으로 런타임 메모리 훼손(Memory Corruption) 차단 가능

## 5. 실증: 동시성 제어 성능 분석 - 비교 환경 구성

### Baseline: C 수동 제어

- ◆ **Interrupt Masking:** 전역 인터럽트 비활성화 명령어 (`CPSID`/`CPSIE`)를 사용하여 임계 구역을 수동으로 보호함
- ◆ **자원 접근 제어:** 공유 변수 조작 전후에 개발자가 직접 마스킹 함수를 호출해야 하며, 누락 시 데이터 경합에 무방비함

```
__disable_irq(); // 인터럽트 차단
// Critical Section: 공유 변수 조작
shared_counter++;
__enable_irq(); // 인터럽트 허용
```

### Proposed: Rust Mutex 보호

- ◆ **Token-based Access:** `CriticalSection` 토큰이 유효한 클로저 내부에서만 자원 접근을 허용하는 구조
- ◆ **컴파일 타임 안전 보장:** 명시적인 마스킹 선언 없이 변수에 접근할 경우 컴파일 에러를 발생시켜 인적 오류를 물리적으로 차단함

```
interrupt::free(|cs| {
    // cs 토큰을 소유했을 때만 내부 데이터 접근 가능
    let mut counter = COUNTER.borrow(cs).borrow_mut();
    *counter += 1;
});
```

\* 두 환경 모두 동일한 하드웨어(STM32F446) 및 클럭(16MHz) 조건에서 비교 실증 수행

## 5.1. 실증: 동시성 제어 성능 분석

순수 추가 오버헤드

14 CYC

성능 요약

Zero-cost  
Abstraction

C 수동 제어 (Base)

17 CYC

Rust Mutex 보호

31 CYC (17+14)

"측정된 14사이클(약 0.87 $\mu$ s)의 오버헤드는 C 환경에서 인터럽트 상태 보존(PRIMASK 제어)을 포함한 안전한 임계 구역 구현 시 요구되는 필수 연산 비용에 **상응하는 수준**이며, 이는 고수준 추상화(Zero-cost Abstraction) 도입이 유의미한 런타임 성능 저하를 유발하지 않음을 입증함"

## 5.2. 실증: 외부 간섭(인터럽트 중첩)에 따른 비결정성 분석

**실험 설계:** CAN 수신 인터럽트(ISR) 처리 도중 다른 요인에 의해 발생하는 **실행 지연 및 비결정성(Jitter)**을 관찰하기 위해, 의도적으로 **고우선순위 인터럽트에 의한 선점(Preemption)** 환경을 구축하고 DWT를 이용해 실행 사이클을 1,000회 반복 측정함.

### Mutex 미적용 (간섭 노출)

측정 결과 (1,000회)

**Avg 38** CYC

(Min: 30 / Max: 116 / SD: 26)

### 극심한 타이밍 지연 및 비결정성

보호받지 못한 실행 구간이 고우선순위 인터럽트에 의해 중단되며 불규칙한 Jitter 발생. 이러한 실행 시간의 **비결정성(Non-determinism)**은 실시간 제어기의 신뢰성을 직접적으로 위협함.

### Rust Mutex 적용 (원자성 보장)

측정 결과 (1,000회)

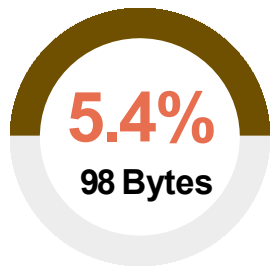
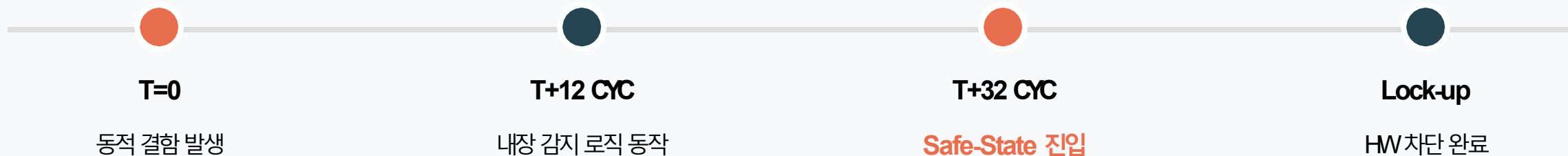
**Avg 31** CYC

(Min: 31 / Max: 31 / SD: 0)

### 완벽한 실행 결정성 (Determinism)

인터럽트 중첩을 물리적으로 차단하여 임계 구역의 실행을 원자적 (Atomic) 으로 완수. 외부 간섭 조건 하에서도 **표준편차(SD) 0**의 일정한 실시간 성능을 확보함.

## 6. 실증: Fail-safe 및 리소스 분석



### 메모리 풋프린트 점유율 (기본 구현 기준)

결함 억제를 위한 기본 패닉 핸들러(Panic Handler) 통제 로직의 풋프린트 증가는 통신 스택 대비 약 5.4%로 측정되었습니다. 이는 기초적인 안전 상태 천이 로직에 한정된 수치이나, 자원이 제한된 MCU에서도 경량화된 런타임 안전 장치의 도입이 가능함을 보여줌.

## 6.1 동적 결함 주입을 통한 Fail-safe 한계 교차 검증

### 정적 분석 우회 통제 환경

- ◆ 컴파일 타임 상수 전개 최적화를 원천 배제하기 위한 실험 설계
- ◆ 외부 하드웨어 인터럽트(GPIO)의 물리적 입력 횟수를 동적 인덱스 변수로 매핑
- ◆ 런타임 이전에는 컴파일러조차 예측 불가능한 비결정적 배열 접근 조건 조성

런타임 경계 초과 (Out-of-bounds) 발생



Rust 내장 바운드 체크 발동



32 Cycle 內 Fail-safe 처리 완료

\* 동적이고 불규칙한 환경에서도 런타임 결함 억제 장치(Containment)가 정상 작동함

## 7. 결론 및 향후 과제

### 1. 결함의 조기 차단 (Shift- Left)

배열 퇴화와 같은 체계적 결함을 컴파일 단계에서 원천 차단하여, 사후 검증과 디버깅에 소모되는 비용을 혁신적으로 절감함.

### 2. 성능 타협 없는 안전 (Zero-cost)

고수준 런타임 안전 메커니즘을 도입하더라도 필수 하드웨어 제어에 상응하는 최소 비용만 발생하며, 간섭환경에서도 엄격한 실행 결정성을 보여줌.

### 3. 저사양 MCU 자원 효율성

Fail-safe 로직 풋프린트가 전체 5.4%에 불과해, 자원이 제한된 제어기 환경에서도 런타임 안전장치의 실무적 수용성을 확인 함.

### 연구의 시사점 및 향후 과제

- 기능안전(ISO 26262) 인증을 위한 Rust 컴파일러 도구 인증(Tool Qualification) 방안 연구
- RTOS 및 이기종 멀티코어 환경으로의 소프트웨어 스택 확장 및 인터페이스 검증
- 기존 C/C++ 레거시 코드와의 상호 운용성(FFI) 시 요구되는 안전성 트레이드오프 분석
- 대규모 ECU 아키텍처 적용에 따른 빌드 복잡도 관리 가이드라인 수립

"본 연구는 *Rust* 언어가 차세대 자동차 제어기 소프트웨어의 신뢰성 향상을 위한 실무적 시스템 프로그래밍 대안이 될 수 있음을 실험적으로 뒷받침함"