White Paper

차량용 임베디드 소프트웨어에서의 Rust 적용 전략

FFI 기반 AUTOSAR 통합을 중심으로

세온이앤에스
Seon ENS, Inc.

Why Now, Rust?

'소프트웨어 정의 차량' 시대로
전환되며

차량용
임베디드 소프트웨어의
안전성과 신뢰성은
그 어느 때보다
중요해졌습니다

점점 높아져가는 제품 복잡도
그리고 이에 따른
소프트웨어 개발에 대한
전방위적인 도전에 대해서
진지한 고민이
필요합니다

차량용 임베디드 소프트웨어에서의 Rust 적용 전략

FFI 기반 AUTOSAR 통합을 중심으로

세온이앤에스
Seon ENS, Inc.

목차

1. 서론: 왜 지금 Rust 인가

1.1 차량용 소프트웨어 복잡도 증가와 메모리 안전성 위기

1.2 C/C++ 기반 개발의 구조적 한계

1.3 글로벌 동향

2. Rust 가 자동차 임베디드에 적합한 이유

2.1 소유권 시스템과 컴파일 타임 메모리 안전성 보장

2.2 Zero-cost Abstraction: 안전성과 성능의 양립

2.3 no_std 베어메탈 지원과 임베디드 생태계

2.4 Unsafe 격리를 통한 단계적 안전성 관리

3. 성능 검증: ARM Cortex-M4 에서 C 와 Rust 는 동등한가

3.1 실험 설계 — 동일 LLVM 백엔드, 어셈블리 수준 화이트박스 분석

3.2 16 개 벤치마크 결과 요약

3.3 Rust 우위 요인: noalias 최적화, 루프 언롤링, 레지스터 할당

3.4 C 우위 요인: 프레임 포인터 생략, ITE 조건부 실행

3.5 핵심 통찰 — 특정 조건에서 소유권 모델이 컴파일러 최적화에도 기여

4. 안전성 검증: 결함 주입과 실시간 신뢰성 실증

4.1 CAN 통신 스택 Rust 구현

4.2 컴파일 타임 결함 차단 — 컴파일 단계로의 결함 검출 시점 이동

4.3 ISR-메인 루프 공유 자원 무결성

4.4 Panic Handler Fail-safe

5. FFI 를 통한 AUTOSAR Classic 통합 — 핵심 전략

5.1 왜 FFI 인가: 기존 C 코드베이스와의 공존 전략

5.2 통합 아키텍처

5.3 FFI 인터페이스 설계 패턴

5.4 AUTOSAR SWC Runnable 에서 Rust 함수 호출 구조

5.5 mobilgene 플랫폼 구동 결과

6. 도입 로드맵: 3 단계 점진적 적용 제안

7. 결론

참고 문헌

Disclaimer

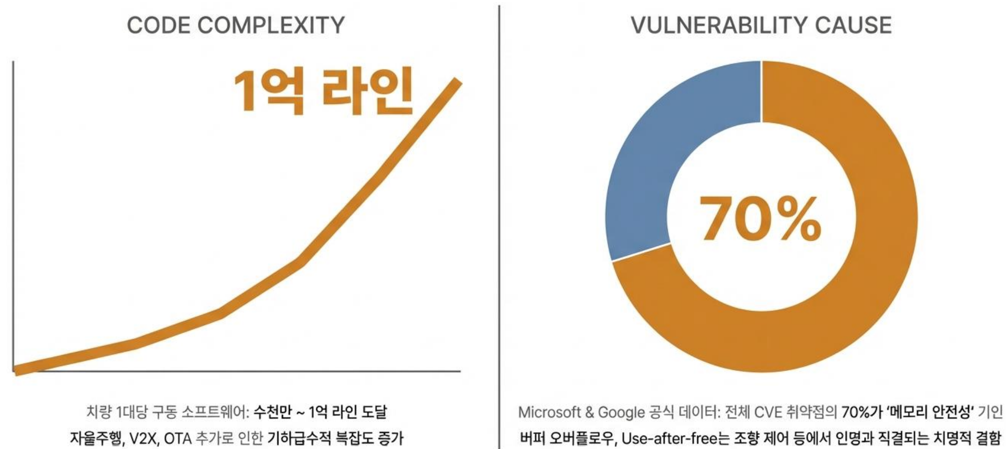
1. 서론: 왜 지금 Rust 인가

1.1 차량용 소프트웨어 복잡도 증가와 메모리 안전성 위기

오늘날 자동차 한 대에 탑재되는 소프트웨어 코드는 수천만에서 1 억 라인에 이르는 것으로 거론됩니다. SDV(Software Defined Vehicle) 전환이 가속화되면서, ECU 의 수는 줄어드는 대신 단일 제어기가 처리해야 할 기능은 기하급수적으로 늘어나고 있습니다. 자율주행, V2X 통신, OTA 업데이트 등 새로운 기능이 추가될수록 소프트웨어의 복잡도는 높아지고, 이는 곧 결함 발생 확률의 증가로 이어집니다.

Microsoft 는 자사에 할당된 CVE(Common Vulnerabilities and Exposures)의 약 70%가 메모리 안전성 문제에서 기인한다고 밝힌 바 있습니다[1]. 버퍼 오버플로우, Use-after-free, 데이터 레이스와 같은 결함은 IT 시스템에서는 보안 패치로 대응할 수 있지만, 차량용 임베디드 시스템에서는 인명 안전과 직결됩니다. 조향 제어 ECU 의 메모리 접근 오류가 주행 중 발생한다면, 그 결과는 돌이킬 수 없습니다. Google 의 Android 팀은 메모리 안전 언어로 작성된 새 코드의 비율을 높인 결과, 메모리 안전 취약점의 비중이 2019 년 76%에서 2024 년 24%로 급감했다는 분석 결과를 공식 블로그를 통해 발표했습니다[2].

그렇다면 수십 년간 자동차 소프트웨어의 근간이 되어온 C 언어만으로 이 문제를 해결할 수 있을까요?



1.2 C/C++ 기반 개발의 구조적 한계

C 언어는 하드웨어에 대한 직접 제어, 예측 가능한 성능, 광범위한 툴체인 지원 등 임베디드 개발에 최적화된 강점을 가지고 있습니다. 수십 년간 자동차 산업이 C 를 선택해 온 데에는 충분한 이유가 있으며, 그 결과로 축적된 코드 자산과 엔지니어링 경험의 가치는 결코 작지 않습니다. 그러나 C 의 설계 철학 자체가 프로그래머에게 메모리 관리의 전적인 책임을 부여하며, 이것이 구조적 한계의 근본 원인입니다.

MISRA C 는 이러한 한계를 코딩 규칙으로 보완하려는 업계의 오랜 노력입니다. 그러나 MISRA C 는 근본적으로 "하지 말아야 할 것"의 목록이라는 점에서 태생적 제약을 갖습니다. 컴파일러가 강제하는 것이 아니라 정적 분석 도구와 코드 리뷰에 의존하므로, 규칙 위반이 분석 도구의 한계나 리뷰어의 실수를 통해 런타임까지 살아남을 수 있습니다. 특히 복잡한 포인터 앨리어싱이나 멀티코어 환경의 데이터 레이스와 같이, MISRA C 의 규칙과 정적 분석 도구만으로는 완전히 방지하기 어려운 결함 유형이 존재합니다. 규칙의 수를 아무리 늘려도, 언어 자체가 허용하는 위험한 동작을 코딩 가이드라인만으로 완벽하게 제어하기는 어렵습니다.

결국 문제는 도구나 규칙이 아닌, 언어 자체의 설계에 있습니다. 프로그래머의 주의력에 의존하는 대신, 컴파일러 수준에서 메모리 안전성을 보장하는 언어가 필요합니다.

검증 항목	C / MISRA C 생태계	근본적 한계
메모리 관리	프로그래머 전적 책임 (수동 제어)	**멀티코어 환경의 데이터 레이스 및 포인터 앨리어싱 구조적 방지 불가**
규칙 강제성	정적 분석 도구와 코드 리뷰어에 의존	**컴파일러의 강제성 부재로 리뷰어 실수 시 런타임까지 결함 생존**
결함 발견 시점	런타임 및 시스템 통합 테스트 단계	**발견 지연으로 인한 수정 비용의 기하급수적 폭증**

1.3 글로벌 동향

이러한 인식은 이미 산업 전반으로 빠르게 확산되고 있습니다. 가장 주목할 만한 변화는 컴파일러 툴체인 인증 영역에서 나타나고 있습니다. 독일의 자동차용 컴파일러 전문 기업 HighTec 은 Infineon AURIX 와 ARM 기반 Stellar 를 대상으로 ISO 26262 ASIL D 수준의 도구 인증(Tool Qualification)을 획득한 Rust 컴파일러를 제공하고 있습니다[3]. 이는 Rust 언어 자체의 안전 인증이 아닌, 특정 MCU/툴체인 조합에서 안전 개발 경로가 마련되었음을 의미합니다. HighTec 은 이미 자동차 업계에서 C/C++ 안전 인증 컴파일러로 광범위한 실적을 보유하고 있어, 기존 고객사가 동일한 신뢰 기반 위에서 Rust 를 도입할 수 있는 경로를 제시하고 있습니다. 이전까지 "Rust 는 안전 인증 컴파일러가 없다"는 것이 도입의 주요 장벽이었는데, HighTec 이 이 장벽을 허물고 있습니다.

정부 차원의 움직임도 주목할 필요가 있습니다. 미국 백악관 ONCD 는 2024 년 2 월 "Back to the Building Blocks: A Path Toward Secure and Measurable Software" 보고서를 통해 메모리 안전 언어로의 전환을 국가적 차원에서 권고했습니다[4]. 이는 특정 산업이 아닌 소프트웨어 생태계 전반에 대한 권고이지만, 안전이 최우선인 자동차 산업에 주는 시사점은 더욱

큽니다. Rust 는 자동차 소프트웨어 영역에서도 실무적 검토 대상이 된 시스템 프로그래밍 언어입니다.

본 백서에서 제시하는 성능 비교, 안전성 검증, AUTOSAR FFI 통합 실험은 모두 세온이앤에스와 자비스*가 직접 코드를 작성하고 실제 MCU 보드 위에서 수행한 결과입니다. 이론적 가능성이 아닌, 자동차 임베디드 현장에서 직접 부딪히며 얻은 데이터와 경험을 바탕으로 합니다.

** 자비스(JARBIS, jarbis.co.kr)는 2025 년 현대자동차 그룹의 사내벤처 육성프로그램을 통해서 선발된 현대자동차 그룹의 사내 스타트업입니다. 자비스는 차량용 소프트웨어 개발의 복잡성을 줄이고, 표준 기반 개발 프로세스를 자동화하기 위한 솔루션을 개발하고 있습니다. 즉 AUTOSAR Classic 환경에서 요구되는 다양한 설정 작업과 코드 생성 과정을 효율화함으로써, 개발 생산성과 품질을 동시에 향상시키는 것을 목표로 합니다.*

2. Rust 가 자동차 임베디드에 적합한 이유

2.1 소유권 시스템과 컴파일 타임 메모리 안전성 보장

Rust 의 가장 핵심적인 차별점은 소유권(Ownership) 시스템입니다. Rust 에서 모든 값은 반드시 하나의 소유자만 가지며, 소유자가 스코프를 벗어나면 해당 값의 메모리가 자동으로 해제됩니다. 이것만으로도 C 에서 빈번하게 발생하는 이중 해제(double free)와 메모리 누수를 원천적으로 방지할 수 있습니다.

여기에 더해 빌림 검사기(Borrow Checker)가 컴파일 타임에 참조의 유효성을 강제합니다. 핵심 규칙은 두 가지입니다. 첫째, 어떤 값에 대해 하나의 가변 참조(`&mut T`)만 존재하거나, 여러 개의 불변 참조(`&T`)만 존재할 수 있습니다. 둘째, 참조는 소유자보다 오래 살 수 없습니다. 이 규칙을 위반하는 코드는 컴파일 자체가 불가능합니다. C 에서 정적 분석 도구가 "경고"로 알려주는 것을 Rust 컴파일러는 "에러"로 차단합니다. 결함이 빌드 단계에서 제거되므로, 테스트나 현장에서 발견되는 메모리 결함을 원천적으로 방지할 수 있습니다.

이러한 특성은 소프트웨어 품질 관리 관점에서 컴파일 단계로 결함 검출 시점을 앞당기는 효과라 할 수 있습니다. 결함의 발견 시점이 테스트 단계에서 빌드 단계로, 즉 개발 프로세스의 가장 왼쪽으로 이동하는 것입니다. 결함을 조기에 발견할수록 수정 비용이 기하급수적으로 줄어든다는 것은 소프트웨어 공학의 기본 원리이며, Rust 는 이를 언어 설계 수준에서 실현합니다.

2.2 Zero-cost Abstraction: 안전성과 성능의 양립

"안전한 코드는 느리다"는 통념이 있습니다. Java 나 Go 처럼 가비지 컬렉션을 사용하는 언어들은 메모리 안전성을 런타임에 보장하는 대가로 예측하기 어려운 지연 시간이 발생합니다. 마이크로초 단위의 결정론적 응답이 요구되는 자동차 실시간 제어 시스템에서 이러한 런타임 오버헤드는 허용될 수 없습니다.

Rust 는 이 딜레마에 대해 근본적으로 다른 접근을 취합니다. Zero-cost Abstraction 원칙에 따라, 추가 런타임 계층 없이, 적절한 조건에서 수작업 최적화 코드에 근접한 성능을 지향합니다. Rust 의 모든 안전성 검사는 전적으로 컴파일 타임에 수행되며, 런타임에는 가비지 컬렉터가 존재하지 않습니다. 트레이트(Trait)를 이용한 정적 디스패치(제네릭) 기반의 다형성은 컴파일 시점에 단형화(Monomorphization)되어, 가상 함수 테이블을 거치지 않고 인라인됩니다. 결과적으로 생성되는 바이너리는 동등한 C 코드와 구조적으로 유사하며, 경우에 따라서는 더 효율적인 기계어 코드가 생성됩니다.

2.3 no_std 베어메탈 지원과 임베디드 생태계

자동차 ECU 에서 동작하는 소프트웨어는 대부분 운영체제 없이 베어메탈 위에서 직접 실행됩니다. Rust 는 no_std 환경을 공식적으로 지원하여, 표준 라이브러리의 힙 할당이나 OS 의존적 기능 없이 베어메탈 위에서 직접 동작할 수 있습니다. ARM Cortex-M 시리즈에 대해서는 Tier 2 타겟 지원을 제공하여, 자동차 ECU 에서 사용되는 주요 마이크로컨트롤러(thumbv7em-none-eabihf 등)에서 Rust 를 바로 실행할 수 있습니다. Tier 2 는 빌드가 보장되며 공식 바이너리가 제공되는 수준으로, 개발 및 검증 단계의 실무 사용에 충분합니다[5]. 다만 양산 적용을 위해서는 ISO 26262[7]에 따른 도구 인증(Tool Qualification)이 별도로 필요합니다.

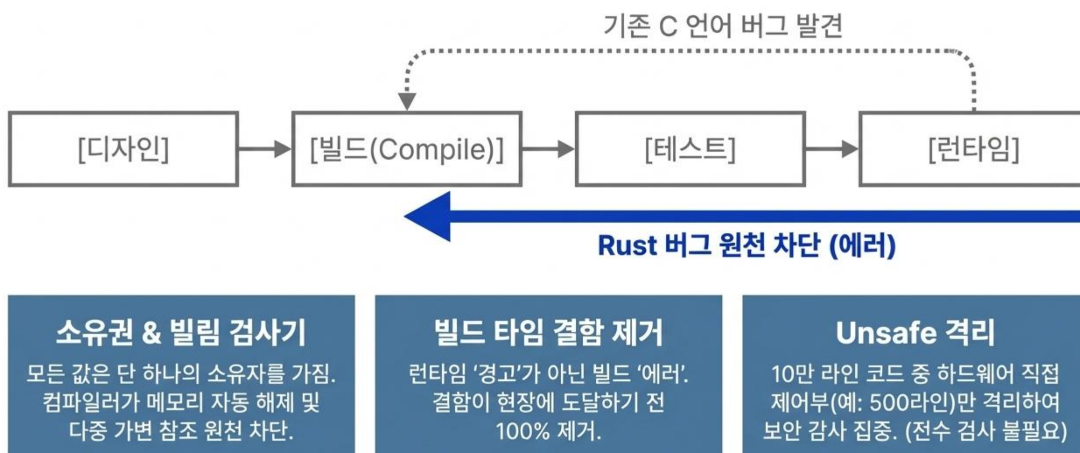
임베디드 Rust 생태계도 빠르게 성숙하고 있습니다. `cortex-m` 크레이트는 ARM Cortex-M 프로세서의 핵심 페리페럴에 대한 안전한 추상화를 제공하고, `embedded-hal` 은 하드웨어 추상화 레이어 트레이트를 정의하여 다양한 MCU 에서 동일한 드라이버 코드를 재사용할 수 있게 합니다. `defmt` 는 임베디드 환경에 최적화된 로깅 프레임워크를, `probe-rs` 는 디버깅과 플래싱 도구를

제공합니다. 기존 C 임베디드 개발자가 Rust 로 전환할 때 필요한 인프라가 이미 상당 수준으로 갖추어져 있는 것입니다.

2.4 Unsafe 격리를 통한 단계적 안전성 관리

현실적으로 임베디드 시스템에서는 하드웨어 레지스터 직접 접근, 인터럽트 핸들링, DMA 전송 설정 등 메모리 안전성을 컴파일러가 정적으로 보장할 수 없는 영역이 반드시 존재합니다. Rust 는 이러한 코드를 `unsafe` 블록으로 명시적으로 격리합니다.

여기서 unsafe 라는 키워드가 주는 인상과 달리, 이것은 "위험한 코드"를 의미하는 것이 아닙니다. "프로그래머가 컴파일러 대신 안전성을 보증하겠다고 선언한 코드"를 의미합니다. 핵심은 이러한 코드가 전체 코드베이스의 극히 일부에만 집중되며, 코드 리뷰와 안전 감사(audit)의 범위를 명확히 한정할 수 있다는 점입니다. 예를 들어 10 만 라인의 ECU 소프트웨어에서 unsafe 코드가 500 라인이라면, 안전성 검증의 초점을 그 500 라인에 집중할 수 있습니다. 물론 실무에서는 unsafe 코드의 줄 수 자체보다, 해당 코드가 형성하는 추상화 경계와 불변식(invariant)의 정확성을 검증하는 것이 핵심입니다. MISRA C 가 전체 코드에 대해 규칙 준수를 검증해야 하는 것과 비교하면, 이는 근본적으로 다른 수준의 효율성을 제공합니다.



3. 성능 검증: ARM Cortex-M4 에서 C 와 Rust 는 동등한가

Rust 의 안전성은 이론적으로 매력적이지만, 자동차 임베디드 엔지니어에게 가장 중요한 질문은 결국 이것입니다. "성능은 괜찮은가?" 아무리 안전해도 실시간 제어 요구사항을 충족하지 못한다면 자동차 ECU 에서는 사용할 수 없습니다. 본 장에서는 ARM Cortex-M4 기반 실제 자동차 ECU 환경에서 수행된 C 와 Rust 의 성능 비교 분석 결과를 소개합니다.

3.1 실험 설계 — 동일 LLVM 백엔드, 어셈블리 수준 화이트박스 분석

기존의 C vs Rust 벤치마크 연구들은 대부분 GCC 로 C 를, LLVM 으로 Rust 를 컴파일하여 비교했습니다. 이 경우 성능 차이가 언어 자체의 특성에서 비롯된 것인지, 아니면 GCC 와 LLVM 이라는 서로 다른 컴파일러 백엔드의 차이에서 비롯된 것인지를 구분할 수 없다는 근본적 한계가 있었습니다.

본 연구에서는 이 문제를 해결하기 위해 C 와 Rust 모두 동일한 LLVM 20.1.7 백엔드를 사용했습니다. C 는 Clang/LLVM 으로, Rust 는 기본 LLVM 백엔드로 컴파일하여, 컴파일러 백엔드의 차이를 제거하고 언어 프론트엔드의 차이만을 순수하게 비교할 수 있는 실험 조건을 구성했습니다. 실험 대상 MCU 는 자동차 ECU 에 널리 사용되는 ARM Cortex-M4 아키텍처 기반의 STM32F446RE 이며, 클럭은 PLL 을 사용하지 않는 16MHz HSI 로 설정하여 외부 변수를 최소화했습니다. 성능 측정은 DWT(Data Watchpoint and Trace) 사이클 카운터를 사용하여 CPU 사이클 수를 직접 측정하는 방식으로 진행했으며, 각 벤치마크는 30 회 반복 수행되었습니다.

특히 주목할 점은 단순히 실행 시간만을 비교하는 블랙박스 벤치마크를 넘어, 컴파일러가 생성한 어셈블리 코드를 명령어 수준에서 비교 분석하는 화이트박스 접근을 적용했다는 것입니다. 이를 통해 성능 차이의 "결과"뿐 아니라 "원인"까지 규명할 수 있었습니다.

3.2 16 개 벤치마크 결과 요약

ECU 소프트웨어에서 사용되는 대표적인 연산 패턴을 반영하여 6 개 카테고리, 16 개 벤치마크를 설계했습니다. 기본 산술(정수, 비트, 고정소수점), 메모리 접근(배열, DMA 메모리), 제어흐름(조건분기, 함수호출), 알고리즘(정렬, 검색, 암호화), 통신(링버퍼, 시그널 인코딩, 디지털 필터), 수학 연산(행렬, 부동소수점/삼각함수) 등 ECU 가 실제로 수행하는 연산을 포괄적으로 다루었습니다.

결과를 종합하면, 16 개 벤치마크 중 Rust 가 우위를 보인 것이 7 개, C 가 우위를 보인 것이 8 개, 동등한 것이 1 개였습니다. 본 실험 조건에서 Rust 와 C 는 전반적으로 대등한 성능을 보였으며, 특정 영역에서는 Rust 가 우위를 보이기도 하여 "Rust 가 C 보다 느리다"는 우려가 일반적이지 않음을 시사합니다.

본 벤치마크는 각 언어에서 실무적으로 사용되는 관용적 코드(idiomatic code)의 성능을 비교한 것입니다. 예를 들어 C 에서 restrict 키워드를 명시적으로 사용하면 Rust 와 유사한 최적화가 가능하나, 실무에서 restrict 를 일상적으로 사용하지 않는 관행을 반영하여 본 비교에서는 사용하지 않았습니다. 따라서 본 결과는 '관용적 구현 + 기본 설정'에서의 실무적 경향으로 해석되어야 합니다.

더 흥미로운 것은 개별 벤치마크의 결과입니다. Rust 가 가장 큰 우위를 보인 DMA 메모리 벤치마크에서는 C 대비 68% 빠른 성능을 기록했고, 행렬 연산에서는 57%, 배열 접근에서는 33% 빠른 결과를 보였습니다. 반대로 C 가 가장 큰 우위를 보인 함수 호출 벤치마크에서는 C 가 83%

빠른 성능을 기록했습니다. 이러한 극단적 차이의 원인을 이해하는 것이 벤치마크 수치 자체보다 더 중요한 통찰을 제공합니다.

3.3 Rust 우위 요인: noalias 최적화, 루프 언롤링, 레지스터 할당

어셈블리 분석 결과, Rust 가 C 를 앞서는 벤치마크에서 세 가지 핵심적인 최적화 패턴이 확인되었습니다.

첫째, 소유권 기반 noalias 최적화입니다. Rust 의 소유권 시스템은 컴파일러에게 "이 참조는 다른 참조와 메모리 영역이 겹치지 않는다"는 강력한 보증을 제공합니다. 이는 LLVM 백엔드의 `noalias` 최적화를 자동으로 활성화하여, 컴파일러가 메모리 접근 순서를 자유롭게 재배치하고 중복 로드를 제거할 수 있게 합니다. C 에서는 프로그래머가 `restrict` 키워드를 명시적으로 사용해야만 가능한 최적화가, Rust 에서는 언어의 기본 규칙으로 인해 자동으로 적용되는 것입니다.

둘째, 적극적인 루프 언롤링입니다. DMA 메모리 벤치마크에서 Rust 컴파일러는 메모리 복사 루프를 4-wide 로 언롤링하여 사이클당 처리량을 극대화한 반면, C 컴파일러는 바이트 단위

복사를 수행했습니다. 이로 인해 Rust 가 68%나 빠른 결과가 나타났습니다. 행렬 연산(57% 빠름)과 배열 접근(33% 빠름)에서도 유사한 루프 최적화 패턴이 관찰되었습니다. Rust 컴파일러가 소유권 정보를 활용하여 메모리 앨리어싱이 없음을 확신할 수 있었기 때문에, 더 공격적인 루프 변환을 안전하게 적용할 수 있었습니다. 다만 C 측에서 memcpy() 표준 라이브러리를 사용했다면 유사한 최적화 경로를 기대할 수 있으므로, 이 수치는 언어 자체의 차이뿐 아니라 관용적 코드 작성 방식의 차이도 반영하고 있습니다.

셋째, 라이브러리 함수 대체입니다. Rust 컴파일러는 `__aeabi_memcpy4` 와 같은 범용 런타임 라이브러리 호출을 인라인 최적화된 코드로 대체하여 함수 호출 오버헤드를 제거했습니다. 이는 Rust 의 제네릭 시스템과 단형화(Monomorphization) 메커니즘이 임베디드 환경에서도 실질적인 성능 이점으로 작용함을 보여줍니다.

3.4 C 우위 요인: 프레임 포인터 생략, ITE 조건부 실행

반대로 C 가 앞선 벤치마크에서는 두 가지 요인이 확인되었습니다.

함수 호출 벤치마크에서 C 가 83%나 빠른 결과를 보인 것은 프레임 포인터 처리 방식의 차이 때문이었습니다. Clang/LLVM 은 기본적으로 프레임 포인터를 생략하여 함수 진입과 종료 시 `push/pop` 명령어를 제거하는 반면, Rust 컴파일러는 기본적으로 프레임 포인터를 유지합니다. 함수 호출이 빈번한 코드에서 이 차이가 누적되어 큰 성능 격차로 나타난 것입니다. 다만 이는 Rust 컴파일러 옵션(`-C force-frame-pointers=no`)으로 조정 가능한 부분이므로, 언어의 본질적 한계라기보다는 기본 설정의 차이로 보는 것이 적절합니다.

제어흐름 벤치마크에서 C 가 36% 빠른 결과를 보인 것은 ARM 의 IT(If-Then) 조건부 실행 명령어 활용 차이에서 비롯되었습니다. C 컴파일러는 조건 분기를 IT 명령어로 변환하여 파이프라인 플러시 없이 조건부 실행을 처리한 반면, Rust 컴파일러는 동일한 상황에서 일반적인 분기 명령어(B)를 생성하여 분기 시마다 파이프라인 플러시가 발생했습니다. 이는 Rust LLVM

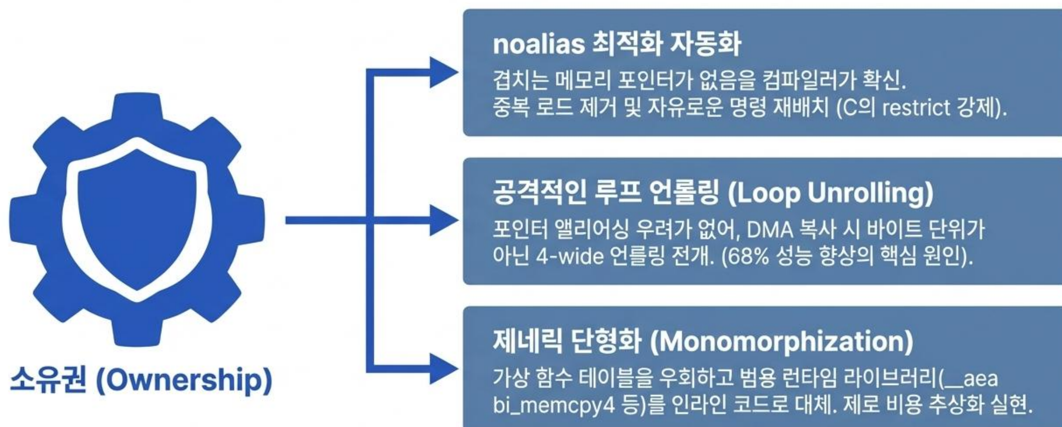
프론트엔드의 코드 생성 패턴 차이로, 향후 컴파일러 업데이트를 통해 개선될 여지가 있는 부분입니다.

3.5 핵심 통찰 — 특정 조건에서 소유권 모델이 컴파일러 최적화에도 기여

본 벤치마크의 가장 중요한 발견은 수치 그 자체가 아닙니다. Rust 의 소유권 시스템이 안전장치이면서 동시에, 특정 조건에서 컴파일러 최적화의 기반으로도 작용한다는 사실입니다.

C 에서는 포인터 앨리어싱의 가능성이 항상 열려 있습니다. 두 포인터가 같은 메모리를 가리킬 수 있기 때문에, 컴파일러는 메모리 접근 순서를 바꾸거나 중복 로드를 제거하는 등의 최적화를 보수적으로 수행할 수밖에 없습니다. 반면 Rust 의 소유권 규칙은 컴파일러에게 메모리 접근 패턴에 대한 강력한 보증을 제공하여, 더 공격적인 최적화를 안전하게 적용할 수 있게 합니다.

일부 조건에서 안전성 강화와 성능 유지 또는 개선이 동시에 가능하다는 것 — 이것이 본 벤치마크에서 확인된 Rust 의 주목할 만한 특성입니다. "안전성과 성능 중 하나를 선택해야 한다"는 기존의 이분법이 Rust 에서는 반드시 성립하지 않을 수 있음을 시사합니다.



4. 안전성 검증: 결함 주입과 실시간 신뢰성 실증

성능이 동등하다면, 다음 질문은 자연스럽게 이것이 됩니다. Rust 의 안전성은 실제로 차량용 임베디드 환경에서 효과가 있을까요? 이론적 안전성이 아니라, ARM Cortex-M4 베어메탈 위에서, 실제 자동차 통신 프로토콜을 구현한 코드에서, 정량적으로 측정 가능한 수준의 안전성을 제공하는지가 핵심입니다. 본 장에서는 Rust 로 CAN 통신 스택을 구현하고, 의도적 결함 주입을 통해 안전성을 실증한 결과를 소개합니다.

4.1 CAN 통신 스택 Rust 구현

검증의 현실성을 확보하기 위해, 단순 알고리즘이 아닌 실제 자동차 통신 프로토콜인 CAN 스택을 Rust 로 구현했습니다. 아키텍처는 AUTOSAR 계층 구조를 참조하여 설계했습니다. 최하위 MCAL 계층에서는 STM32F446RE 의 bxCAN 페리페럴 레지스터를 직접 제어하며, 중간의 CanIf 계층에서는 CAN 인터페이스 추상화와 메시지 라우팅을 담당하고, 최상위 Application 계층에서는 비즈니스 로직과 CAN 메시지 송수신을 처리합니다.

전체 시스템은 no_std 베어메탈 환경에서 staticlib 형태로 빌드되어, 운영체제나 외부 런타임 없이 MCU 위에서 직접 동작합니다. 이 환경은 실제 자동차 ECU 와 유사한 베어메탈 MCU 실행 환경이므로, 검증 결과의 기초적 타당성을 확보할 수 있습니다.

4.2 컴파일 타임 결함 차단 — 컴파일 단계로의 결함 검출 시점 이동

C 와 동일한 결함 패턴을 Rust 코드에 의도적으로 주입하여, Rust 컴파일러의 결함 차단 능력을 검증했습니다. 첫 번째 결함은 타입 불일치입니다. `[u8; 8]` 타입(8 바이트 배열)을 요구하는 CAN 메시지 전송 함수에 `[u8; 4]`(4 바이트 배열)를 전달하는 코드를 작성했습니다. C 에서는 배열이 함수 인자로 전달될 때 포인터로 퇴화(decay)되면서 크기 정보가 소실되므로, 이 코드는 경고 없이 빌드를 통과합니다. 런타임에 함수가 8 바이트를 읽으려 할 때 4 바이트를 초과하여 인접 메모리를 읽는 버퍼 오버리드가 발생하며, 이는 CAN 메시지의 데이터 오염으로 이어집니다. 반면 Rust 에서는 `[u8; 4]`와 `[u8; 8]`이 서로 다른 타입이므로 컴파일 에러가 발생하여 빌드 자체가 불가능합니다.

두 번째 결함은 경계 초과 접근입니다. 8 바이트 CAN 데이터 버퍼에서 인덱스 8(9 번째 원소)에 접근하는 코드를 작성했습니다. C 에서는 컴파일러 경고 없이 빌드되며, 런타임에 버퍼 경계를 넘어 인접 메모리를 읽어 예측 불가능한 동작이 발생합니다. Rust 에서는 상수 인덱스의 경우 컴파일 타임에, 동적 인덱스의 경우 런타임 바운드 체크에 의해 안전하게 차단됩니다.

이러한 결과가 의미하는 바는 명확합니다. 기존에 단위 테스트, 통합 테스트, 혹은 최악의 경우 필드에서 발견되던 결함이 빌드 단계에서 차단됩니다. 결함의 수정 비용은 발견 시점이 늦어질수록 기하급수적으로 증가한다는 것은 소프트웨어 공학의 기본 원리이며, Rust 는 이 원리에 따라 결함 발견 시점을 개발 프로세스의 가장 앞 단계로 이동시킵니다.

4.3 ISR-메인 루프 공유 자원 무결성

자동차 ECU 에서 인터럽트 서비스 루틴(ISR)과 메인 루프 간 공유 데이터의 무결성은 핵심 안전 요구사항입니다. 예를 들어 CAN 수신 ISR 이 메시지 버퍼를 업데이트하는 도중에 메인 루프가 동일 버퍼를 읽으면, 절반만 갱신된 데이터를 읽게 되어 데이터 정합성이 깨질 수 있습니다. C 에서는 이러한 문제를 인터럽트 비활성화나 크리티컬 섹션으로 수동 관리해야 하며, 프로그래머가 보호를 누락하면 간헐적이고 재현이 극히 어려운 데이터 레이스 버그가 발생합니다.

Rust 의 임베디드 크레이트가 제공하는 `Mutex` 를 적용하여 공유 자원을 보호하고, DWT 사이클 카운터로 오버헤드를 정밀 측정했습니다. Mutex 를 적용한 경우 CAN 메시지 처리에 평균 31 사이클이 소요되었으며, 표준편차가 0, 최솟값과 최댓값이 모두 31 로 완벽한 결정론적 동작을 보였습니다. Mutex 를 적용하지 않고 ISR 간섭이 발생한 경우에는 평균 38 사이클이 소요되었지만, 표준편차가 26 으로 크게 흔들렸고 최솟값 30 에서 최댓값 116 까지 비결정적 지터가 관찰되었습니다.

외부 간섭이 없는 통제 환경에서 Mutex 미적용 시 기본 실행 시간은 평균 17 사이클이며, Mutex 적용 시 31 사이클로 측정되었습니다. 즉 Mutex 로 인한 안전성 오버헤드는 단 14 사이클입니다. 16MHz 클럭 기준으로 1 마이크로초에도 미치지 않는 시간입니다. 이 14 사이클은 C 에서 수동으로 __disable_irq() / __enable_irq()를 호출하여 임계 구역을 보호할 때도 동일하게 지불해야 하는 본질적 비용입니다. 즉 Rust 의 Mutex 추상화 계층이 추가적인 성능 비용을 발생시키지 않으며, C 에서 수작업으로 작성한 코드와 동등한 기계어가 생성됩니다. 그리고 더 중요한 것은, 적절한 동시성 추상화와 타입 규칙을 사용하고 unsafe 경계를 올바르게 관리하는 경우, Rust 에서는 보호되지 않은 공유 자원 접근을 컴파일 단계에서 차단할 수 있다는 점입니다. C 에서 프로그래머의 기억력에 의존하던 것이, Rust 에서는 컴파일러가 강제하는 규칙이 됩니다.

4.4 Panic Handler Fail-safe

컴파일 타임 검사를 통과한 코드라 하더라도, 런타임에 동적 인덱스로 배열에 접근하는 경우에는 바운드 체크가 수행됩니다. 이때 Rust 는 C 와 결정적으로 다른 선택을 합니다. C 에서 배열 경계를 넘어선 접근은 정의되지 않은 동작(Undefined Behavior)을 유발하여, 프로그램이 아무 일도 없었다는 듯이 잘못된 데이터로 계속 실행될 수 있습니다. Rust 에서는 이러한 위반이 발생하면 `panic` 을 트리거하여, 정의되지 않은 동작 대신 제어된 실패를 보장합니다.

본 실험에서 구현한 Panic Handler 는 패닉 발생을 감지하면 즉시 CAN 하드웨어 송신을 차단하고 안전 대기 상태(Safe State)로 전환합니다. 측정 결과 패닉 감지부터 안전 상태 진입까지 31 사이클이 소요되었으며, 이는 16MHz 기준 약 2 마이크로초에 해당합니다. Panic

Handler 전체의 메모리 풋프린트는 98 바이트로, Panic Handler 로직 66 바이트, Panic Dispatcher 16 바이트, 안전 보조 심볼 16 바이트로 구성됩니다. 이는 본 테스트 구현체의 전체 코드(.text 섹션, 1,808 바이트) 대비 약 5.4%에 불과합니다.

ISO 26262 의 Fail-safe 설계 취지와 부합하는 메커니즘을 98 바이트, 31 사이클이라는 소규모 풋프린트로 구현할 수 있음을 보여줍니다. 이는 자원이 극히 제한된 자동차 ECU 환경에서도 Rust 의 런타임 안전성 메커니즘이 실무적으로 적용 가능함을 시사합니다. 정의되지 않은 동작으로 잘못된 제어 명령이 계속 전달되는 것보다, 2 마이크로초 만에 안전 상태로 전환되는 것이 기능안전 관점에서 압도적으로 나은 선택일 수 있습니다.

5. FFI 를 통한 AUTOSAR Classic 통합 — 핵심 전략

앞선 장에서 Rust 의 성능과 안전성을 자동차 임베디드 환경에서 확인했습니다. 그렇다면 현실적인 문제가 남습니다. 수십 년간 축적된 AUTOSAR 기반 C 코드베이스를 어떻게 Rust 와 통합할 것인가? 기존 코드를 모두 버리고 Rust 로 재작성하는 것은 비현실적이며, 그럴 필요도 없습니다. 본 장에서는 FFI(Foreign Function Interface)를 통한 AUTOSAR Classic 통합 전략과, 현업 AUTOSAR 플랫폼 기반의 구동 검증 결과를 소개합니다.

5.1 왜 FFI 인가: 기존 C 코드베이스와의 공존 전략

AUTOSAR BSW(Basic Software) 스택은 수백만 라인의 검증된 C 코드로 이루어져 있습니다. OS, 통신 스택, 진단, 메모리 관리 등 수십 년에 걸쳐 검증되고 인증된 코드 자산이며, 각 OEM 과 Tier-1 의 요구에 맞게 커스터마이징된 결과물입니다. 여기에 오랜 기간 축적된 테스트 자산과 안전 인증 이력까지 고려하면, 이 모든 것을 포기하고 처음부터 다시 작성한다는 것은 기술적으로나 사업적으로나 현실성이 없습니다.

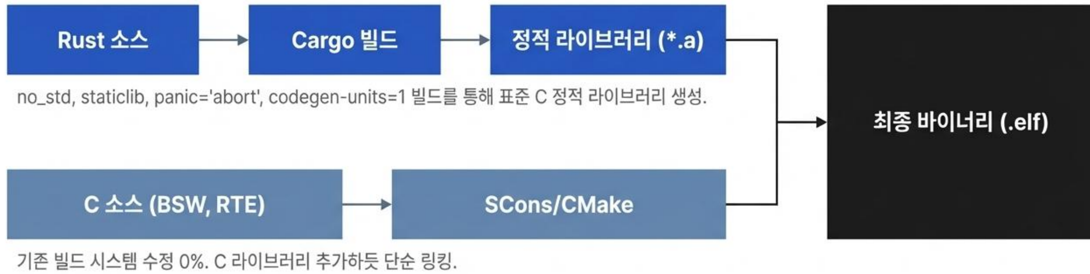
FFI 는 이 현실적 제약에 대한 답입니다. 기존 C 코드베이스를 그대로 유지하면서, 새로운 기능이나 안전이 특히 중요한 모듈만 Rust 로 작성하여 통합하는 점진적 전환 전략을 가능하게 합니다. Rust 의 FFI 는 C ABI 와 직접 호환되며, Java 의 JNI 나 Python 의 ctypes 와 달리 추가적인 런타임이나 중간 계층이 필요하지 않습니다. 컴파일된 바이너리 수준에서 C 함수를 호출하고 C 에서 Rust 함수를 호출하는 것이 추가적인 런타임 계층이 없어 함수 호출 수준의 오버헤드는 매우 작습니다. 다만 ABI 호환, 데이터 레이아웃 설계, 빌드 통합 등 시스템 수준의 엔지니어링 비용은 별도로 고려해야 합니다.

5.2 통합 아키텍처

AUTOSAR 플랫폼에 Rust 를 통합하기 위한 빌드 파이프라인은 크게 두 단계로 나뉩니다. 먼저 Rust 소스 코드를 Cargo 를 통해 크로스 컴파일하여 정적 라이브러리(.a 파일)를 생성합니다. 이때 타겟은 `thumbv7em-none-eabihf`(ARM Cortex-M 하드 플로트)로 지정하고, 크레이트 타입은 `staticlib`, 패닉 전략은 `abort`, 코드 생성 단위는 `1` 로 설정합니다. `staticlib` 은 OS 없이 링킹 가능한 표준 C 정적 라이브러리를 생성하며, `panic = "abort"`는 패닉 시 스택 언와인딩 없이 즉시 중단하여 임베디드 환경에 적합하고, `codegen-units = 1` 은 단일 코드 생성 단위로 전체 최적화를 극대화합니다.

생성된 정적 라이브러리는 AUTOSAR 빌드 시스템(SCons 또는 CMake)에서 기존 C 소스, BSW, RTE 와 함께 링킹되어 최종 바이너리(.elf)가 됩니다. 이 방식의 핵심적인 장점은 AUTOSAR 빌드 시스템을 전혀 수정하지 않아도 된다는 것입니다. Rust 코드는 사전에 빌드된 정적 라이브러리로 제공되므로, 기존 빌드 시스템 입장에서는 일반적인 C 라이브러리를 하나 추가하는 것과 다를 바 없습니다. 이는 기존 CI/CD 파이프라인이나 빌드 인프라에 미치는 영향을 최소화합니다.

통합 아키텍처: AUTOSAR 시스템의 무결한 결합



통합 실행 계층
#[no_mangle], extern 'C', #[repr(C)] 인터페이스 적용.
개별 SWC의 Runnable 단위로 분할하여 C 래퍼를 통해 Rust 함수 호출.
기존 AUTOSAR BSW와 스케줄링 타이밍 변경 불필요.

5.3 FFI 인터페이스 설계 패턴

Rust 와 C 간 FFI 인터페이스는 세 가지 핵심 어트리뷰트의 조합으로 구성됩니다.

`#[no_mangle]` 어트리뷰트는 Rust 컴파일러의 이름 맹글링을 비활성화합니다. Rust 는 기본적으로 함수명에 해시를 추가하여 고유한 심볼명을 생성하는데, 이를 비활성화해야 C 에서 함수명으로 직접 호출할 수 있습니다. `extern "C"`는 C 호출 규약(calling convention)을 사용하도록 지정하여, 인자 전달 순서와 스택 정리 방식이 C 와 동일하게 동작합니다. `#[repr(C)]`는 구조체의 메모리 레이아웃을 C 와 동일하게 보장하여, C 와 Rust 간에 구조체를 안전하게 공유할 수 있게 합니다.

이 세 가지를 조합하면 Rust 측에서 다음과 같이 FFI 함수를 정의할 수 있습니다. `#[repr(C)]`가 적용된 `CanMessage` 구조체는 C 의 동일한 구조체와 메모리 레이아웃이 일치하며, `#[no_mangle] pub extern "C" fn rust_process_can_message()`으로 선언된 함수는 C 코드에서 `extern` 선언만으로 호출할 수 있습니다. C 측 AUTOSAR SWC 에서는 일반적인 외부 함수 호출과

동일한 방식으로 Rust 함수를 사용하므로, 기존 C 개발자가 별도의 학습 없이도 Rust 모듈을 활용할 수 있습니다.

5.4 AUTOSAR SWC Runnable 에서 Rust 함수 호출 구조

AUTOSAR Classic 아키텍처에서 Rust 코드가 삽입되는 지점은 SWC(Software Component)의 Runnable 내부입니다. AUTOSAR 에서 Runnable 은 RTE(Runtime Environment)에 의해 스케줄링되는 최소 실행 단위로, 주기적으로 또는 이벤트에 의해 호출됩니다.

이 통합 방식에서 AUTOSAR 의 계층 구조는 그대로 유지됩니다. MCAL, BSW, RTE 는 기존 C 코드가 그대로 동작하며, SWC 의 Runnable 내부에서만 C 래퍼 함수를 통해 Rust FFI 함수가 호출됩니다. RTE 의 스케줄링 메커니즘을 그대로 활용하므로 기존 AUTOSAR 타이밍 설계를 변경할 필요가 없고, BSW 스택은 전혀 수정이 필요하지 않습니다. 가장 중요한 것은 Runnable 단위로 개별 전환이 가능하다는 점입니다. 하나의 SWC 내에서도 일부 Runnable 만 Rust 로 전환하고 나머지는 C 를 유지할 수 있으므로, 리스크를 최소화하면서 점진적으로 Rust 코드의 비중을 늘려갈 수 있습니다. 또한 기존 SWC 레벨의 테스트 프레임워크를 그대로 재사용할 수 있어, 테스트 자산의 연속성도 보장됩니다.

5.5 mobilgene 플랫폼 구동 결과

이론적 아키텍처를 넘어, 양산 플랫폼 계열의 AUTOSAR 환경에서 구동을 검증한 결과를 소개합니다. 현대오토에버의 mobilgene 플랫폼(AUTOSAR Classic 4.4, R44)과 NXP S32K312 EVB 보드를 사용하여 Rust FFI 통합 테스트를 수행했습니다[6][8]. NXP S32K312 는 ARM Cortex-M7 기반으로 최대 120MHz 클럭, 2MB Flash, 256KB SRAM 을 제공하는 차량용 마이크로컨트롤러이며, 디버거로는 Lauterbach TRACE32 를 사용했습니다.

테스트는 ECU 소프트웨어에서 사용되는 연산 패턴을 포괄적으로 다루도록 7 개 카테고리, 총 26 개의 FFI 함수로 설계되었습니다. 기본 연산(Basic Operations) 카테고리에서는 정수, 실수, 비트, 고정소수점 연산을, 메모리 접근(Memory Access)에서는 배열, 구조체, 메모리 복사 패턴을, 제어흐름(Control Flow)에서는 조건 분기, 상태 머신, 반복문, 함수 호출 패턴을 검증했습니다. 통신 프로토콜(Communication Protocol) 카테고리에서는 CAN 프레임 처리, CRC 계산, 링버퍼, 시그널 인코딩을, 실시간 태스크(Real-Time Tasks)에서는 스케줄링, 세마포어, 메시지 큐, 타이머, 리소스 락 등을, 알고리즘(Algorithms)에서는 정렬, 검색, 필터, 룩업 테이블, PID 제어, 비트 패킹 등을 다루었습니다. 마지막으로 하드웨어 인터페이스(Hardware Interface)에서는 시뮬레이션 메모리 맵 방식으로 GPIO, 레지스터, 인터럽트, 타이머 접근 패턴을 검증했습니다.

측정의 신뢰성을 확보하기 위해 각 FFI 함수에 대해 1,000 회 이상의 반복 측정을 수행했으며, 처음 100 회는 워밍업으로 제외했습니다. 통계 지표로 평균, 표준편차, 최솟값, 최댓값, 95% 신뢰구간, 변동계수(CV)를 산출했고, 통계적 유의성은 t-test 또는 Mann-Whitney U 검정($p < 0.05$)으로 확인했습니다. 또한 기본 설정, LTO 비활성화, 인라인 비활성화의 세 가지 빌드 구성에서 각각 측정하여 최적화 옵션의 영향까지 분석했습니다.

결과적으로 26 개 전체 FFI 함수가 mobilgene AUTOSAR Classic 4.4 플랫폼에서 정상 구동되었습니다. 이 결과가 의미하는 바는 세 가지입니다. 첫째, no_std 와 staticlib 조합으로 AUTOSAR BSW 스택과 무결한 통합이 가능합니다. 둘째, ECU 에서 사용되는 주요 연산 패턴 — 산술, 메모리, 제어흐름, 통신, 실시간, 알고리즘, 하드웨어 인터페이스 — 을 Rust FFI 로 구현하고 실행할 수 있습니다. 셋째, 기존 AUTOSAR 빌드 시스템(SCons)과 Cargo 빌드를 최소한의 수정으로 통합할 수 있습니다. 이 결과는 Rust-AUTOSAR FFI 통합이 연구실 수준의 개념 증명이 아니라, 양산 플랫폼 계열 환경에서 구동이 확인된 실현 가능한 전략임을 보여줍니다.

6. 도입 로드맵: 3 단계 점진적 적용 제안

Rust 를 자동차 임베디드 환경에 도입할 때, 전면적 전환은 현실적이지도 않고 바람직하지도 않습니다. 기존 자산을 보호하면서 리스크를 관리할 수 있는 점진적 접근이 필요하며, 다음의 3 단계 로드맵을 제안합니다.

첫 번째 단계는 유틸리티와 알고리즘 모듈에서 시작하는 것입니다. CRC 계산, 룩업 테이블, 디지털 필터, 데이터 인코딩/디코딩 등 AUTOSAR BSW 나 하드웨어에 대한 의존성이 없는 순수 연산 모듈이 가장 안전한 출발점입니다. 이러한 모듈은 입력과 출력이 명확하고, 기존 C 구현과의 동치성을 검증하기 용이하며, 문제가 발생하더라도 C 구현으로 즉시 롤백할 수 있습니다. 이 단계의 진정한 목적은 Rust 코드 자체보다는, 팀의 Rust 역량 확보와 빌드 파이프라인 구축, FFI 인터페이스 설계 패턴 수립, 테스트 방법론 정립 등 이후 단계의 기반을 마련하는 데 있습니다.

두 번째 단계는 Safety-critical SWC 로의 확장입니다. 1 단계에서 축적된 경험과 인프라를 바탕으로, ASIL 등급이 부여된 안전 관련 모듈에 Rust 를 적용합니다. CAN/LIN 통신 처리, 센서 데이터 유효성 검증, 진단(Diagnostics) 모듈 등 메모리 안전성이 기능안전 요구사항과 직결되는 영역이 대상입니다. 이 단계에서 Rust 의 컴파일 타임 안전성 보장은 ISO 26262 가 요구하는 소프트웨어 안전성 확보에 유리한 특성을 제공할 수 있습니다. 다만 실제 ASIL 등급 개발에의 적용에는 별도의 안전 논증(Safety Case)과 도구 인증이 수반되어야 합니다. 아울러 이 단계의 진입을 위해서는 Rust 컴파일러 및 관련 툴체인에 대한 ISO 26262 Part 8 기반의 도구 인증(Tool Qualification) 확보가 선행되어야 합니다.

세 번째 단계는 Rust 네이티브 AUTOSAR 바인딩의 구축입니다. 앞선 두 단계에서 개별 함수 수준의 FFI 를 넘어, AUTOSAR BSW API 에 대한 체계적인 Rust 바인딩을 구축하여 SWC 전체를 Rust 로 작성할 수 있는 환경을 마련합니다. `bindgen` 을 활용한 자동 바인딩 생성과, Rust 타입 시스템을 활용한 안전한 래퍼 API 설계가 이 단계의 핵심 과제입니다. 이 단계에 이르면 Rust 가 AUTOSAR 생태계의 일급 시민(first-class citizen)으로 자리 잡게 됩니다.

각 단계의 전환은 반드시 이전 단계에서의 검증된 결과를 기반으로 이루어져야 하며, 어느 단계에서든 기존 C 코드와의 공존이 보장됩니다. "모든 것을 한 번에 바꾸겠다"는 접근이 아니라, "증명된 만큼만 확장하겠다"는 접근이 성공적인 Rust 도입의 열쇠입니다.

7. 결론

본 백서에서는 차량용 임베디드 소프트웨어에 Rust 를 적용하기 위한 기술적 근거와 실증적 검증 결과, 그리고 FFI 기반 AUTOSAR 통합 전략을 제시했습니다.

세 가지 독립적인 연구를 통해 성능, 안전성, 통합 가능성이라는 세 축을 검증했습니다. ARM Cortex-M4 에서 동일 LLVM 백엔드 기반으로 수행된 비교 실험에서 Rust 는 C 와 대등한 성능을 보였으며, 소유권 시스템에 기반한 noalias 최적화를 통해 특정 영역에서는 C 를 상회했습니다. 안전성 검증에서는 컴파일 타임 결함 차단, 14 사이클의 Mutex 오버헤드, 31 사이클의 Panic Handler 레이턴시를 통해, Rust 의 안전성 메커니즘이 실시간 제약이 엄격한 ECU 환경에서도 실용적임을 정량적으로 입증했습니다. 그리고 현대오토에버의 mobilgene 플랫폼에서 26 개 FFI 함수가 정상 구동됨을 확인하여, AUTOSAR Classic 환경과의 실제적 통합이 가능함을 실증했습니다.

Rust 는 C 의 성능을 유지하면서 메모리 안전성을 컴파일러 수준에서 강하게 보장하는 대표적인 시스템 프로그래밍 언어입니다. FFI 를 통한 점진적 도입 전략은 기존 자산을 보호하면서도 소프트웨어 품질을 구조적으로 향상시킬 수 있는 현실적 경로를 제공합니다. 기존 AUTOSAR 빌드 시스템을 수정하지 않고, BSW 스택에 영향을 주지 않으면서, Runnable 단위로 하나씩 전환해 나갈 수 있다는 것은 실무적으로 매우 높은 도입 가능성을 의미합니다.

차량용 소프트웨어의 복잡도가 계속 증가하고, SDV 전환이 가속화되는 현실에서, Rust 는 더 이상 "관심을 가져볼 만한 언어"가 아닌 "도입 타당성을 본격적으로 검토해야 할 기술"입니다. 본 백서에서 제시한 실증 데이터와 도입 전략이, 그 준비의 첫 걸음에 구체적인 방향을 제시할 수 있기를 바랍니다.

참고 문헌

[1] "A proactive approach to more secure code," MSRC Blog

[2] "Eliminating Memory Safety Vulnerabilities at the Source," Google Online Security Blog

[3] "Infineon expands Rust ecosystem for AURIX with HighTec's ISO 26262 ASIL D qualified Rust compiler"

[4] "Back to the Building Blocks: A Path Toward Secure and Measurable Software," Technical Report

[5] https://doc.rust-lang.org/rustc/platform-support.html

[6] https://www.hyundai-autoever.com

[7] ISO26262

[8] https://www.autosar.org/standards/classic-platform

Disclaimer

본 백서는 정보 제공을 목적으로 작성된 자료로, 어떠한 경우에도 법적, 재정적, 투자적 조언을 제공하기 위한 문서가 아닙니다.

회사는 본 백서의 내용 또는 그 사용으로 인해 발생할 수 있는 직접적, 간접적, 부수적, 결과적 손해에 대해 어떠한 법적 책임도 지지 않습니다.

또한, 본 백서의 어떠한 부분도 계약적 권리나 법적 의무를 발생시키지 않으며, 회사 또는 관련 기관의 공식적인 제안, 약속, 보증으로 해석되어서는 안 됩니다.

백서의 모든 정보는 “있는 그대로(as is)” 제공되며, 독자는 본 문서의 내용을 참고함에 있어 자신의 판단과 책임하에 행동해야 합니다.

Lean Tech. Fast Results.
SDV 시대, 빠르게 대응하는 통합형 기술 파트너
세온이앤에스

세온이앤에스는 자동차 전장 소프트웨어와 기능안전, Automotive SPICE®, 사이버보안 등 미래차 핵심 분야에 특화된 전문 기업입니다. 검증된 기술력과 컨설팅 경험을 기반으로 고객의 빠른 시장 대응을 지원하며, 공인 교육기관으로서 글로벌 표준을 이끄는 역량을 갖추고 있습니다.

끊임없는 혁신과 전문성을 바탕으로, SDV 시대의 든든한 파트너가 되겠습니다.

세온이앤에스에 대해서 더 자세히 알아 보세요!
https://www.seonens.com