



RUST 의 AUTOSAR 적용을 위한 실제적인 성능 평가검증 사례

발표자 이력

I. (주) 자비스(2026년 3월 설립)

- 대표
 - 차량 SW 개발을 쉽고 빠르게 하는 다양한 솔루션

II. 현대자동차(1년)

- 사내스타트업

III. 현대오토에버(6년)

- AUTOSAR Classic mobilgene platform 개발

III. 현대오트론(10년)

- 간접방식 TPMS 국내 최초 양산 개발
- 전기차 LDC 제어기 개발



C와의 성능 비교로 Rust의 임베디드 환경에서 사용성을 검증



Rust

메모리 안전성을 컴파일 타임에 보장
임베디드 성능은 미지수



C 언어

수십 년간 임베디드에서
검증된 사실상 표준



Hard Real-Time 제약 속에서 Rust가 C만큼의 성능을 낼 수 있는가?
→ 이미 검증된 C와 직접 비교하여 '임베디드 사용성'을 정량 검증한다

세션 1

C언어와 RUST의 성능 비교

```
#include <stdio.h>
int main() {
    int count = 0;
    for (int i = 0; i < N; i++) {
        count += i;
    }
    return count;
}
```



vs

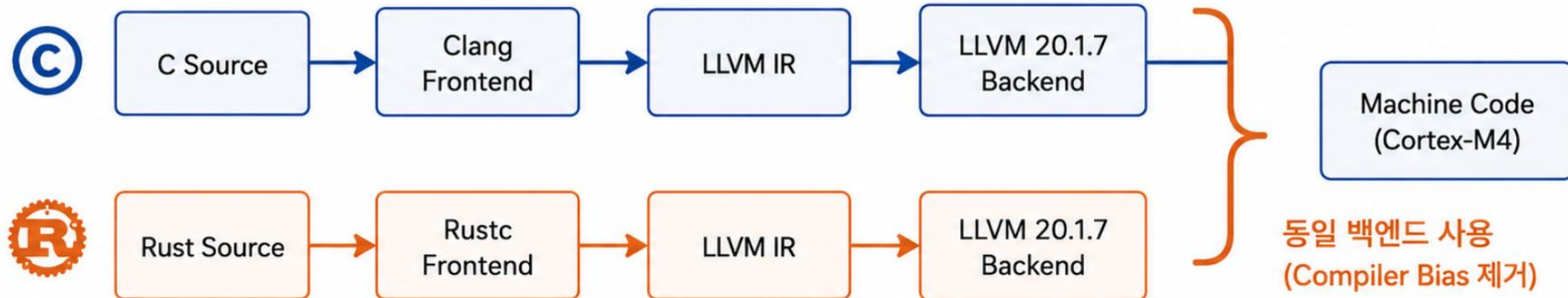


```
fn main() {
    let mut count = 0;
    for i in 0..N {
        count += i;
    }
    println!("{}", count);
}
```

컴파일러 편향 제거: 동일 LLVM 20.1.7 백엔드로 언어(프론트엔드) 차이만 분리



컴파일 파이프라인 (Compilation Pipeline)



하드웨어 설정 (Hardware Setup)

1	대상 MCU	STM32F446RE (ARM Cortex-M4F)
2	클럭 속도	16MHz
3	측정 방법	DWT Cycle Counter (CPU 사이클 정밀 측정)



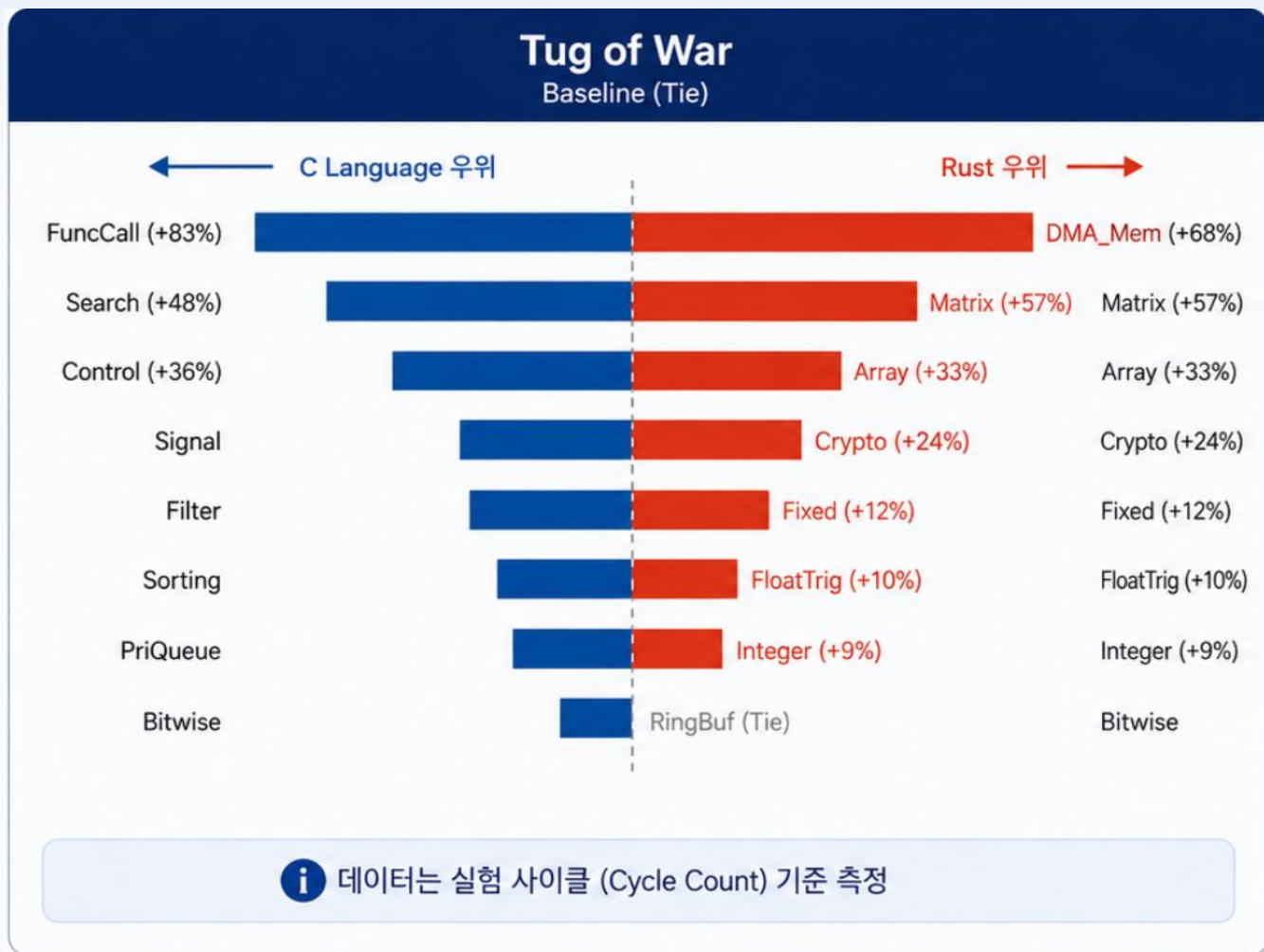
빌드 설정 (Build Configuration)

1	C (Clang)	Opt -O2, LTO Full, Loop Unrolling Off
2	Rust (rustc)	Opt-level 2, LTO True, Panic abort, Release mode

기본연산·메모리·알고리즘·신호처리·제어·수학 — 총 16종을 차량 Use case에 매핑

카테고리	대표 벤치마크	차량 ECU Use case
 기본연산	Integer, Bitwise	센서 raw 데이터 처리
 메모리	DMA_Mem, Array, RingBuf	버퍼 복사·록업·링버퍼
 알고리즘	Search, Sorting, PriQueue	진단·이벤트 스케줄
 신호처리	Signal, Filter, FloatTrig	센서 신호 필터링
 제어	Control, FuncCall	상태기계·제어 로직
 수학연산	Matrix, Crypto, Fixed	ADAS 행렬·보안·고정소수

영역별 성능 차이 분석: Rust 7개 : C언어 8개 : 동등 1개



벤치마크 결과 (The Score)

총 16개 항목 테스트 결과



7

Rust Wins



1

Tie



8

C Wins

● 7 Rust Wins

Rust가 더 우수한 성능을 보인 영역
DMA_Mem, Matrix, Matrix, Array,
Crypto, Fixed, FloatTrig, Integer

● 1 Tie

동등한 성능을 보인 영역
RingBuf

● 8 C Wins

C언어가 더 우수한 성능을 보인 영역
FuncCall, Search, Control, Signal,
Filter, Sorting, PriQueue, Bitwise

Rust 우위 영역: 메모리 및 수학 연산

+68%
DMA_Mem
(메모리 복사)

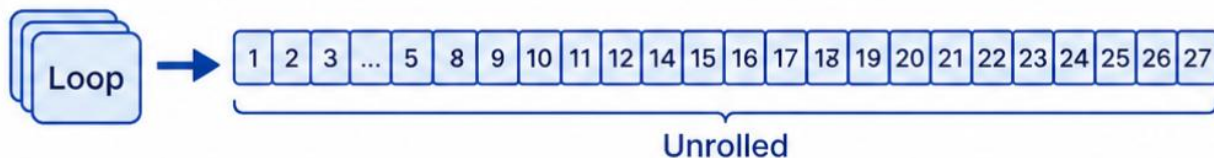
01 성능 향상 원인 (The Mechanism)

- Diagram 1: LLVM Library Substitution



- ✓ Rust의 슬라이스 복사 패턴을 LLVM이 인식하여 최적화된 ARM 라이브러리로 자동 변환

+57%
Matrix
(행렬 연산)

02 Diagram 2: Aggressive Loop Unrolling

- ✓ Matrix(3x3) 연산에서 루프 제어 분기(Branch)를 제거하고 명령어 27개를 순차 나열



Zero-Cost Abstractions: 추상화된 Iterator 코드가 수동 루프보다 효율적인 기계어로 번역됨

C 언어 상위 영역: 제어 흐름 및 함수 호출

+83%

FuncCall (함수 호출)

01 성능 저하 원인 (The Mechanism)

- Diagram 1: Stack Frame Overhead

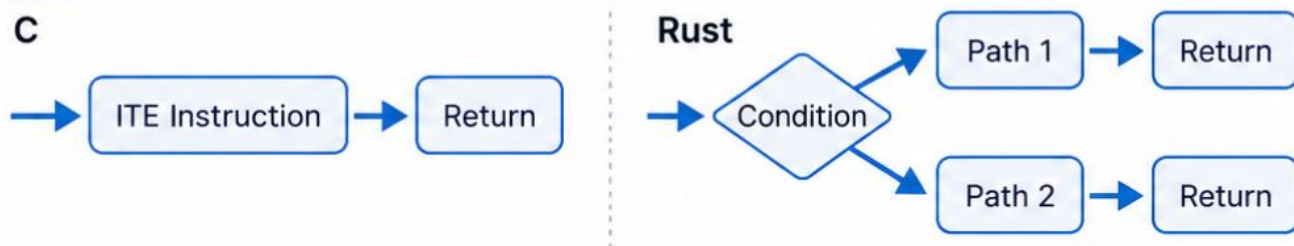


- ✓ C는 속도를 위해 프레임 포인터를 생략하나, Rust는 디버깅/패닉 복구를 위해 유지함

+36%

Control (제어 로직)

02 Diagram 2: Branching Strategy

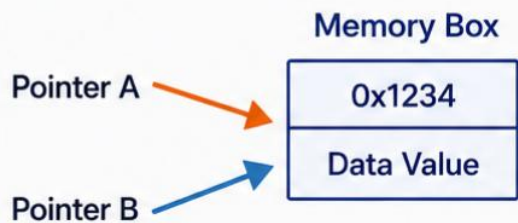


- ✓ C는 ARM의 ITE(If-Then-Else) 명령어로 분기를 최소화하지만, Rust는 다중 복귀점을 생성

안전성 보너스(The Safety Bonus): 제약이 속도를 만든다

Aliasing & Optimization

C Language Context

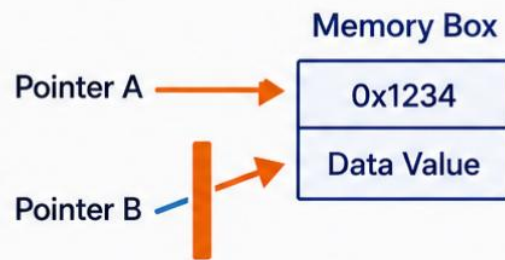


Potential Aliasing
(별칭 가능성)



컴파일러가
보수적으로 최적화
(데이터 충돌 우려)

Rust Language Context



&mut Exclusion
(배타적 소유권)



No Alias 보장

최적화 결과 (Consequences)



Loop Vectorization (SIMD)

데이터 의존성 걱정 없이 병렬 처리

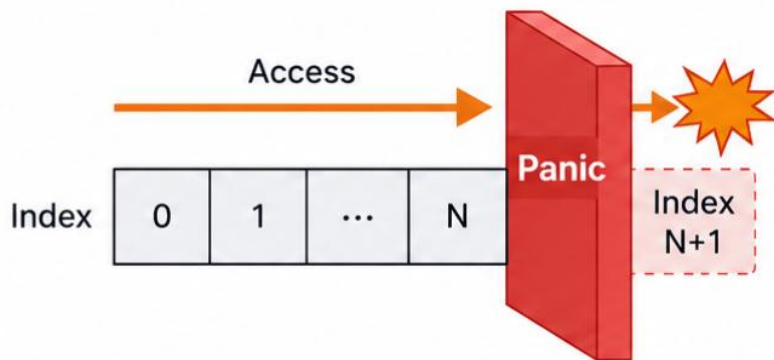


Memory Reordering

효율적인 메모리 접근 순서로 재배열

안전성 세금(The Safety Tax): 보호를 위한 비용

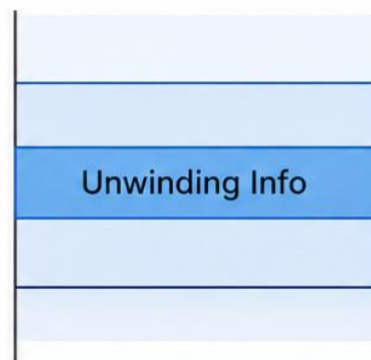
01 Cost 1: Bounds Checking (경계 검사)



Search(이진 탐색) 벤치마크: -46% 성능 저하

- ✓ 매 배열 접근마다 'cmp' 명령어로 범위를 확인.
실패 시 프로그램 중단(Panic)으로 메모리 오염 방지.

02 Cost 2: Panic Safety (스택 관리)



FuncCall 벤치마크: 오버헤드 발생

- ✓ 예외 상황(HardFault)에서 정확한 원인 추적을 위해
프레임 포인터를 유지하는 비용 발생
(C는 이를 생략하여 속도 확보).



Mitigation: '**unsafe**' 블록이나 '**Iterator**'를 사용하면 경계 검사를 제거할 수 있음

(단, 개발자 주의 요망)

동일한 성능이라면, 안전성이 높은 Rust가 유리하다

도메인 (Domain)	추천 언어 (Recommendation)	이유 (Rationale)
 Cryptography / Security	 Rust	 성능 +24% 우위 및 메모리 안전성 필수 영역
 Math / Matrix Ops (ADAS)	 Rust	 루프 언롤링 최적화로 +57% 성능 우위
 Memory Management	 Rust	 대량 데이터 복사 시 +68% 성능 향상
 Control Logic / State Machine	 C Language	 복잡한 분기 처리 및 ITE 명령어 활용 (+36% 우위)
 Legacy / Hardware Drivers	 C Language	 기존 생태계 호환성 및 단순한 메모리 모델

세션 2

AUTOSAR CLASSIC

mobilgene에 RUST 통합



SAFETY



PERFORMANCE



INTEGRATION



FLEXIBILITY



AUTOSAR CLASSIC PLATFORM

APPLICATION LAYER

RUNTIME ENVIRONMENT

BASIC SOFTWARE

MICROCONTROLLER

FFI(Foreign Function Interface)를 통한 AUTOSAR Classic 통합

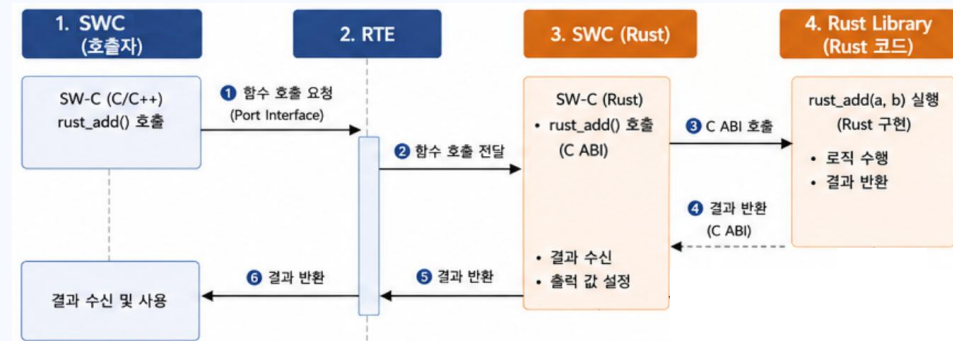
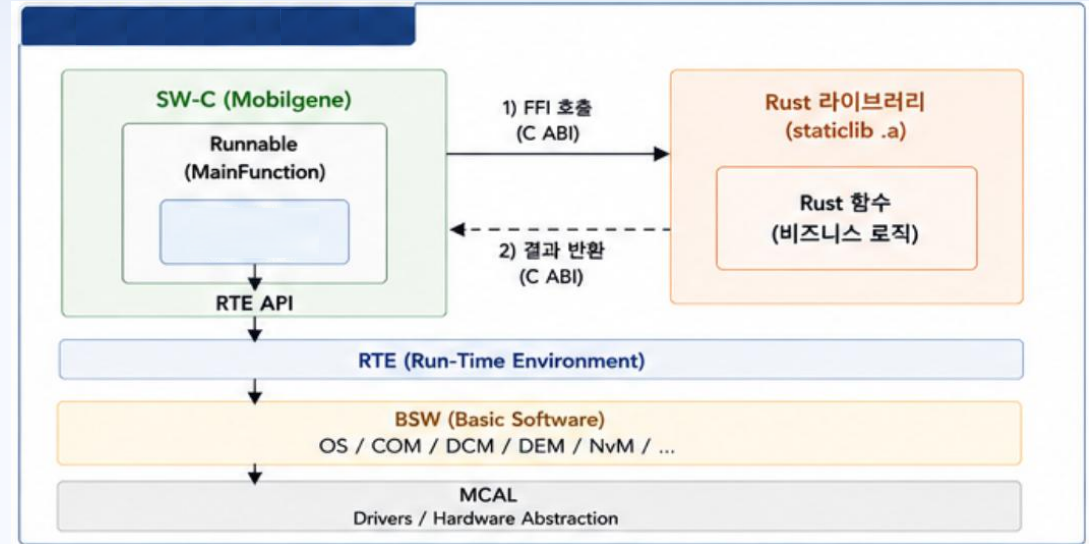
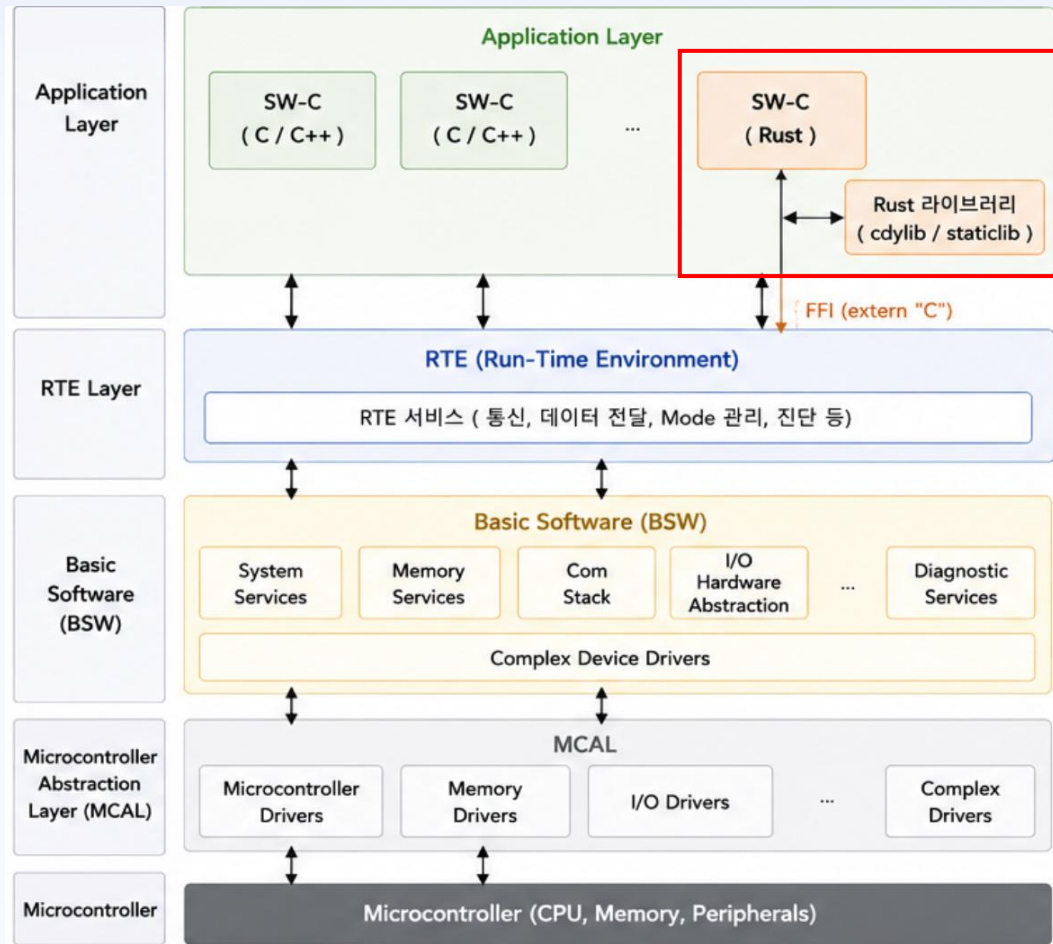
- 기존 배포받은 AUTOSAR platform(mobilgene)에서 기존 코드 수정없이 RUST를 통합
- FFI를 활용하여 C언어와 동일한 형태로 RUST 사용(중간 계층/런타임 없음, 함수 호출 오버헤드 매우 작음)
FFI(Foreign Function Interface)는 서로 다른 프로그래밍 언어 간에 함수(Function)를 호출할 수 있도록 연결해주는 인터페이스
- RTE·BSW·MCAL은 그대로 사용하고, Application Component에서 개별 Runnable만 Rust 호출



AUTOSAR Classic 기반의 차량용 소프트웨어 플랫폼

ECU 개발 시 필요한 BSW(Basic Software), RTE(Runtime Environment), OS, 통신 및 진단 스택 등을 제공하여 차량용 소프트웨어를 표준화된 방식으로 개발할 수 있도록 지원하는 개발 플랫폼

AUTOSAR SWC의 Runnable에서 Rust 함수 호출 구조 & 흐름



* **ABI(Application Binary Interface):**
다른 언어/컴파일러로 만든 코드가 서로 함수 호출할 수 있도록 하는 규약

* **Runnable:** RTE가 스케줄링하는 최소 실행 단위

Rust 소스에서 ELF까지 전체 진행 절차

 Cargo.toml

```
# Rust 빌드 옵션
[lib]
crate-type = ["staticlib"]      # 정적 라이브러리 산출
[build]
target = "thumbv7em-none-eabihf" # 타겟: ARM Cortex-M
```

2가지(`#[no_mangle]`, `extern "C"`) 어트리뷰트의 조합으로 FFI 함수 정의하고 C와 안전하게 연결

Rust (lib.rs)

```
#[no_mangle]
pub extern "C" fn rust_step(adc_sample: i32) -> i32 {
    // 핵심 로직 (예: 제어/필터/통신 처리 등)
    adc_sample + 1
}

-----

#[no_mangle]
pub extern "C" fn rust_add(a: i32, b: i32) -> i32 {
    a + b
}
```

어트리뷰트

역할

`#[no_mangle]`이름 망글링 비활성화 → C가 함수명으로
직접 호출

```
rust_step
↓
_ZN8rustlib9rust_step17h123456789
```

`extern "C"`C ABI (Application Binary Interface) 규약 적용
C 호출 규약 → 인자 전달, 스택, 반환값 등
C와 동일기존 C 기반자가 빌드 학습 없이
일반 외부 함수 호출처럼 Rust 모듈 사용 가능

C ↔ Rust 연동 흐름

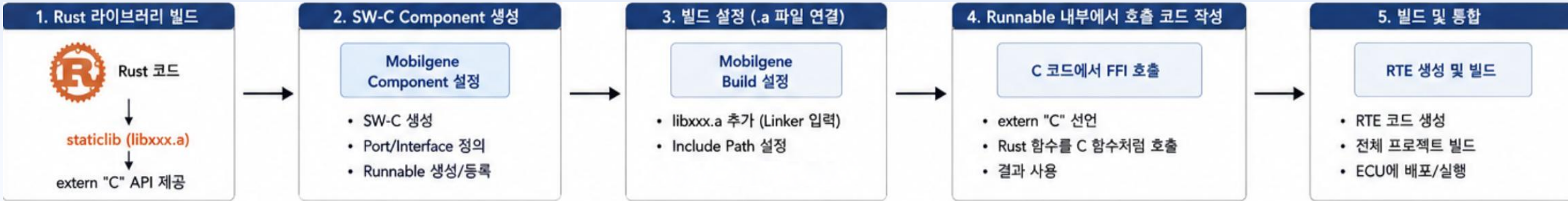
C 코드
(main.c 등)

함수 호출

```
rust_step(...)
rust_add(...)
```

Rust 라이브러리
(lib<name>.a / .so)핵심: `#[no_mangle]` + `extern "C"` 조합으로 Rust 함수를 C가 안전하고 직접 호출할 수 있게 연결

Rust 소스에서 ELF까지 전체 진행 절차 상세



1. Rust 코드 (staticlib 생성)

```

lib.rs
1  #[no_mangle]
2  pub extern "C" fn rust_init() {
3      // 애플리케이션 초기화(상태/버퍼/데이터 등)
4  }
5
6  #[no_mangle]
7  pub extern "C" fn rust_step(adc_sample: i32) -> i32 {
8      // 핵심 로직(예: 제어/필터/통신 처리 등)
9      adc_sample + 1
10 }
11
12 #[no_mangle]
13 pub extern "C" fn rust_add(a: i32, b: i32) -> i32 {
14     a + b
15 }
16
17
18
  
```

빌드 결과

libxxx.a
(Rust static library)

```

# Rust static library 생성
$ cargo build --release --target thumbv7em-none-eabihf
# 결과: target/thumbv7em-none-eabihf/release/libxxx.a
  
```

2. mobilgene: sw-c component 생성 및 runnable 등록

(1) sw-c component 생성

CSWC_RootComposition > MyRustComponent

Components and Ports

Component	Type
App_Lin_LIN1	App_Lin_LIN1 [ApplicationSwComponentType]
MyRustComponent	MyRustComponent [ARRoot/MyRustComponentType]
SWC_AppMode	SWC_AppMode [AppMode/ApplicationSwComponentType]

- CSWC_RootComposition에서 sw-c component 생성
- Component Name: "MyRustComponent"
- Type: MyRustComponentType (ApplicationSwComponentType 상속)

(2) runnable 등록

CSWC_RootComposition > MyRustComponent > Runnables

Runnables

- RustRunnable0

- sw-c "MyRustComponent"에 runnable 등록
- Runnable: "RustRunnable0"

4. Runnable 내부 호출 코드 (C 코드에서 FFI 호출)

```

RustRunnable.c
#include "Rte_MyRustComponentT.h"
#include <stdint.h>

/* Rust 함수 프로토타입 선언 */
extern void rust_init(void);
extern int32_t rust_step(int32_t adc_sample);
extern int32_t rust_add(int32_t a, int32_t b);

FUNC(void, MyRustComponentT_CODE) RustRunnable(void)
{
    int32_t adc_sample = Rte_Read_Adc_Value();
    int32_t out = 0;
    int32_t sum = 0;

    rust_init();
    out = rust_step(adc_sample);
    sum = rust_add(out, 10);

    Rte_Write_Result_Value(sum);
}
  
```

호출 방식

```

graph TD
    Runnable[Runnable C 코드] -- "FFI C ABI" --> RustLib[Rust Library .a]
    RustLib -- "결과 반환 C ABI" --> Result[결과 반환]
  
```

- extern "C" 로 선언된 Rust 함수를 C 함수처럼 호출
- 반환 값 사용 또는 RTE/BSW API와 연계 가능

3. 빌드 설정 (.a 파일 연결)

Build > Compile

LibraryFiles -- Compile [/AUTRON/SCons/...]

Value:

Filter:

Add Remove Up Down

Value: rustlib, sys, startup, arch, ansi

mobilgene R44(AUTOSAR Classic4.4)에서 Runnable 기반으로 Rust static library(.a)를 FFI 방식으로 연동 및 디버깅 검증: 26개 FFI 함수 정상

카테고리	검증 패턴
 기본 연산	정수 · 실수 · 비트 · 고정소수점
 메모리 접근	배열 · 구조체 · 메모리 복사
 제어 흐름	조건분기 · 상태머신 · 반복 · 함수호출
 통신 프로토콜	CAN 프레임 · CRC · 링버퍼 · 시그널 인코딩
 알고리즘	정렬 · 검색 · 필터 · 록업 · PID · 비트패킹
 하드웨어 IF	GPIO · 레지스터 · 인터럽트



측정의 신뢰성 확보

- 각 FFI 함수에 대해 **1,000회 이상** 반복 측정 수행 (초기 100회 warm-up 제외)
- 평균 · 표준편차 · 최솟값 · 최댓값 산출
- Welch's t-test 기반 통계적 유의성 검증 수행 ($p < 0.05$)

검증 환경

-  플랫폼: mobilgene
AUTOSAR Classic 4.4 (R44)
-  MCU: NXP S32K312
Cortex-M7 · 120MHz · 2MB/256KB
-  디버거: Lauterbach TRACE32,
Tasking BlueBox
-  결과: 26개 전 함수 정상 구동
-  검증방법: 디버거를 통해 구동 확인
오실로스코프로 각 기능 수행시간 측정

실제 검증 구성

