

Subject : Software Development

RUST를 활용한 모터 제어 SW 개발

2026. 05. 27

Presented by SEONENS



I. BLDC Motor

II. Rust & C Integration Process

III. Implementation

- Timer Scheduler
- CAN Communication
- Motor Controller

IV. Motor Control SW Structure

V. Rust Advantages

Motor Type Overview

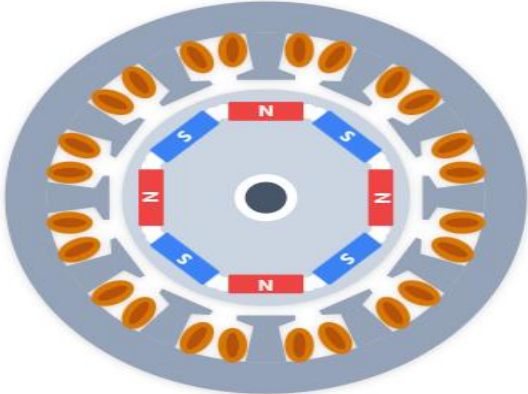
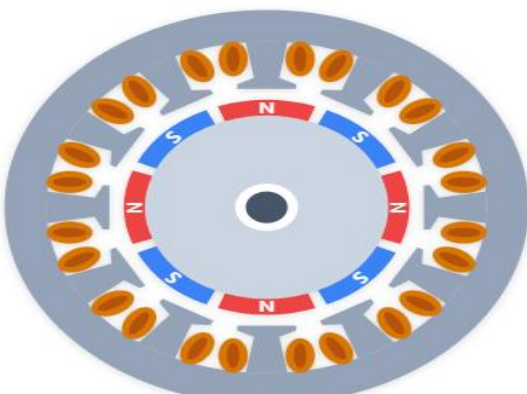
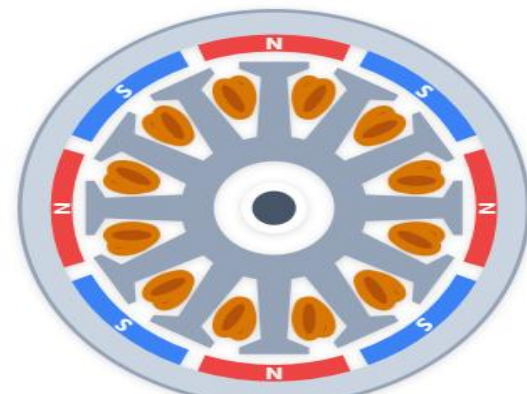
- 전류 공급 방식 및 정류(Commutation) 방식에 따라 DC, Step, BLDC 모터로 구분
- 차량용 액추에이터는 효율·수명·소음 측면에서 BLDC 모터가 표준 선택지로 자리잡음

구분	DC Motor	Step Motor	BLDC Motor
정류 방식	브러시 + 정류자 (기계식 접촉)	다상 권선 순차 여자 (개회로 펄스 제어)	전자식 정류(Inverter) (반도체 스위칭)
위치 센서	불필요	기본적으로 불필요 (Open Loop)	Hall Sensor / Encoder 또는 Sensorless(BEMF)
토크 특성	기동 토크 우수 저속 토크 양호	정지 시 유지 토크 우수 탈조 위험 존재	전 속도 영역 토크 균일 고효율
효율 / 수명	60~75% / 짧음 (브러시 마모)	70~80% / 김 (기계 접점 없음)	85~95% / 매우 김 (비접촉 정류)
주요 적용	와이퍼, 시트, 윈도우 등 저가형 액추에이터	프린터, 로봇 관절, 정밀 위치 제어	구동 모터, EPS, 펌프, WLC, 냉각팬 등

I. BLDC Motor

■ BLDC Motor Type

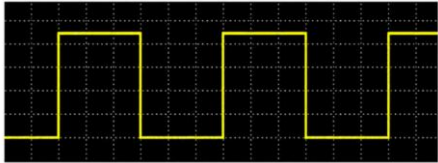
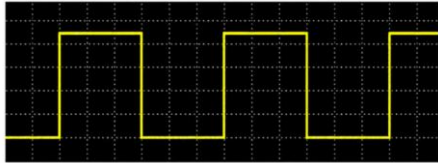
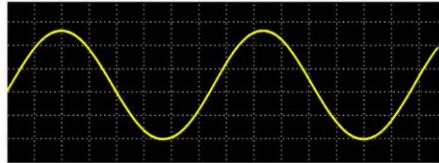
- 고정자 (Stator) ● 회전자 (Rotor)
- 자석 (Magnets) ● 코일 (Coils)

IPMSM	SPMSM	Outrunner BLDC
		
■ Interior Permanent Magnet Synchronous Motor (매입형 영구 자석 동기 전동기) ▶ 특징 - 회전자 철심 내부에 영구자석 매입, d-q축 인덕턴스 차이($L_d \neq L_q$) 큰 돌극형 구조 ▶ 장점 - 마그네트 + 릴럭턴스 토크 동시 활용 - 약자속 제어로 광범위 속도 영역 운전 - 자석 내부 고정 → 초고속 회전에 기계적 안전 ▶ 단점 - 회전자 구조 복잡 → 제조 비용 ↑ - MTPA 제어 등 비선형 알고리즘 필요 - LUT 기반 토크 맵 튜닝 공수 ↑	■ Surface Permanent Magnet Synchronous Motor (표면 부착형 영구 자석 동기 전동기) ▶ 특징 - 회전자 표면에 영구자석 부착, d-q축 인덕턴스 거의 동일($L_d \approx L_q$)한 비돌극형 구조 ▶ 장점 - 선형 토크 특성 → FOC 제어 구현 직관적 - 고속 응답성 우수, 토크 리플 작음 - 제조 공정 단순 → BLDC 대비 비용 합리적 ▶ 단점 - 초고속 영역에서 자석 박리(Detachment) 위험 - 약자속 제어 영역 IPMSM 대비 제한적 - 토크 밀도 IPMSM 대비 낮음	■ Outrunner Brushless DC Motor (외전형 BLDC, 내부 코일·외부 자석형) ▶ 특징 - 고정자(권선) 내부, 회전자(자석) 외부 링 형태 → 외 피 자체가 회전하는 역전형 구조 ▶ 장점 - 회전자 직경 큼 → 관성모멘트·토크 큼 - 저속 고폭 → 감속기 없이 직결 구동 - 외부 노출 회전자로 자연 공랭 효과 ▶ 단점 - 관성 커서 급가속/급정지(Dynamic Response) 불리 - 회전자 외부 노출 → 이물질·접촉 위험 - 외경 기준 토크 밀도 낮음

I. BLDC Motor

Motor Control Methods

- 모터 제어 방식은 제어 루프의 피드백 유무와 전류 제어 정밀도에 따라 구분
- 제어 방식에 따라 요구되는 센서 구성 및 연산량이 다름

제어 방식	Open Loop 6-Step	Closed Loop 6-Step	FOC
원리	사전 정의된 6개 스위칭 시퀀스($U \rightarrow V, U \rightarrow W, V \rightarrow W, V \rightarrow U, W \rightarrow U, W \rightarrow V$)를 일정 주기로 순차 인가하여 회전 자계 생성	Hall Sensor로 회전자 자속(Rotor Flux) 위치를 검출하고 전기각(Electrical Angle)에 동기화하여 인버터를 스위칭하는 사다리꼴 정류(Trapezoidal Commutation)	3상 전류를 Clarke→Park 변환으로 d-q 회전좌표계로 변환 후 자속분(I_d)-토크분(I_q) 전류를 독립 P 제어하고 SVPWM으로 인버터 구동하는 벡터 제어
위치 피드백	없음 (Sensorless Open Loop)	Hall Sensor (60° 단위 이상 위치 검출)	Resolver / Encoder 또는 Sensorless(BEMF, EKF 관측기)
전류 형태			
장점	알고리즘 단순, 센서 불필요로 BOM 절감, 8/16-bit MCU에서도 구동 가능	회전자 동기로 탈조 없음, 부하 변동에 견고, 가속 성능 양호	전 영역 최대 토크(MTPA), 저토크 리플·저소음, 약자속 제어 → 고속 운전
단점	토크 리플·소음 발생, 부하 급변 시 탈조 가능	6고조파 토크 리플, Hall 분해능(60°) 한계	연산량 ↑, 고분해능 센서 + 32-bit MCU 필요, PI 튜닝 난이도 ↑

I. BLDC Motor

■ Motor Control System Overview

■ Motor Specification

- Model Name DB28S01
- Type SPMSM
- Pole / Slot 4-Pole / 6-Slot
- Rated Voltage 15 [V] DC
- Current (No Load/Rated/Peak) 0.3/0.45/1.3 [A]
- Rated Power 4.2 [W] $P \approx \frac{\tau \times N}{9.55}$
- Speed (No Load/Rated) 9600/8000 [rpm]
- Torque (Rated/Peak) 0.005/0.015 [Nm]
- Outer Diameter Ø 28 [mm]

■ Motor Control Configuration

- Open-loop 6-Step Trapezoidal
(Timer 기반 Commutation)
- MCU / 모듈
AURIX TC36x / CCU6
- 상태 머신
Stop → Ready → Ramp → Run (Fault 상태 별도 처리)
- Commutation 출력
Step 0~5 자동 진행 → EN_GATE + U/V/W Out

6-Step Sequence				
Step	U	V	W	전류 경로
0	High	Low	Floating	U -> V
1	High	Floating	Low	U -> W
2	Floating	High	Low	V -> W
3	Low	High	Floating	V -> U
4	Low	Floating	High	W -> U
5	Floating	Low	High	W -> V

II. RUST & C Integration Process

■ C to Rust Conversion Process

※ ASW : Application Software Layer ※ SRV : Service Layer
 ※ RTE : Runtime Environment ※ CDD : Complex Device Driver
 ※ BSW : Basic Software Layer ※ ECAL : ECU Abstraction Layer
 ※ iLLD : Infineon Low Level Driver ※ MCAL : Microcontroller Abstraction Layer

- C에서 Rust로의 변환은 하드웨어 비의존적인 상위 계층(ASW)부터 하드웨어를 직접 접근할 수 있는 하위 계층(MCAL)까지 단계적으로 진행



II. RUST & C Integration Process

■ C to Rust Conversion Process

C Code	Rust Code
<pre> typedef enum { MOTOR_STATE_STOP = 0U, MOTOR_STATE_ALIGN = 1U, MOTOR_STATE_RAMP = 2U, MOTOR_STATE_RUN = 3U, MOTOR_STATE_FAULT = 4U } motor_state_t; static volatile uint8_t gu8_MotorState = MOTOR_STATE_STOP; void CddMotorHandler_MainFunction(void) { switch (gu8_MotorState) { case MOTOR_STATE_STOP: { CddMotorHandler_StateStop(); break; } case MOTOR_STATE_ALIGN: { CddMotorHandler_StateAlign(); break; } case MOTOR_STATE_RAMP: { CddMotorHandler_StateRamp(); break; } case MOTOR_STATE_RUN: { CddMotorHandler_StateRun(); break; } case MOTOR_STATE_FAULT: { CddMotorHandler_StateFault(); break; } default: { gu8_MotorState = MOTOR_STATE_STOP; break; } } } </pre>	<pre> #[repr(u8)] #[derive(Clone, Copy, PartialEq)] #[allow(non_camel_case_types)] pub enum motor_state_t { MOTOR_STATE_STOP = 0, MOTOR_STATE_ALIGN = 1, MOTOR_STATE_RAMP = 2, MOTOR_STATE_RUN = 3, MOTOR_STATE_FAULT = 4, } use motor_state_t::*; static GU8_MOTOR_STATE: AtomicU8 = AtomicU8::new(MOTOR_STATE_STOP as u8); pub fn CddMotorHandler_MainFunction() { let lu8_state = current_state(); match lu8_state { MOTOR_STATE_STOP => { state_stop() }, MOTOR_STATE_ALIGN => { state_align() }, MOTOR_STATE_RAMP => { state_ramp() }, MOTOR_STATE_RUN => { state_run() }, MOTOR_STATE_FAULT => { state_fault() }, } } </pre>

■ C의 typedef enum + switch / case / break vs. Rust의 #[repr(u8)] pub enum + match

- C : typedef enum {...} O; 로 enum 선언
switch (var) { case O: ... break; } Syntax format
- Rust : #[repr(u8)] + #[derive(...)] 속성과 함께 pub enum O {...} 로 선언
match var { O => expr, ... } Syntax format
모든 variant가 반드시 등장해야 컴파일 가능

II. RUST & C Integration Process

■ C to Rust Conversion Process

C Code	Rust Code
<pre>static volatile uint8_t gu8_Direction = MOTOR_DIR_CW; static volatile uint8_t gu8_CommStep = 0U; static void CddMotorHandler_NextCommStep(void) { if (gu8_Direction == MOTOR_DIR_CW) { gu8_CommStep++; if (gu8_CommStep >= MCALCCU6_COMM_STEPS) { gu8_CommStep = 0U; } } else { if (gu8_CommStep == 0U) { gu8_CommStep = MCALCCU6_COMM_STEPS - 1U; } else { gu8_CommStep--; } } }</pre>	<pre>static GU8_DIRECTION: AtomicU8 = AtomicU8::new(MOTOR_DIR_CW); static GU8_COMM_STEP: AtomicU8 = AtomicU8::new(0); fn next_comm_step() { let lu8_dir = GU8_DIRECTION.load(Ordering::Relaxed); let lu8_step = GU8_COMM_STEP.load(Ordering::Relaxed); let lu8_new_step: u8; if lu8_dir == MOTOR_DIR_CW { let lu8_s = lu8_step + 1; if lu8_s >= MCALCCU6_COMM_STEPS { lu8_new_step = 0; } else { lu8_new_step = lu8_s; } } else { if lu8_step == 0 { lu8_new_step = MCALCCU6_COMM_STEPS - 1; } else { lu8_new_step = lu8_step - 1; } } GU8_COMM_STEP.store(lu8_new_step, Ordering::Relaxed); }</pre>

- **C의 static volatile + ++ / = 직접 접근 vs. Rust의 static AtomicU8 + .load / .store 메서드**
 - C : static volatile uint8_t var 선언
var++ / var = X / if(var == Y) 형태로 직접 접근 (Load - Modify - Store)
 - Rust : static VAR: AtomicU8 = AtomicU8::new(init) 으로 선언
읽기는 VAR.load(Ordering::Relaxed), 쓰기는 VAR.store(val, Ordering::Relaxed) 메서드 호출
local 변수는 let (불변 기본) / let mut (가변 명시)로 구분

II. RUST & C Integration Process

■ C to Rust Conversion Process

※ #[no_mangle]: Rust 컴파일러가 함수 심볼명에 해시/타입 정보를 붙이는 mangling을 비활성화하여, C 링커가 원래 이름으로 함수에 접근 가능하게 함

C Code	Rust Code
<pre> #define STM0_BASE (0xF0001000U) #define STM_OFF_CMP0 (0x30U) #define STM_OFF_ISCR (0x40U) #define ISCR_CMP0IRR (1U << 0) static volatile uint32_t g_ticks = 0U; IFX_INTERRUPT(McalStm_SysTick_Isr, 0, IFX_INTPRIO_STM0_SR0) { volatile uint32_t* lp_cmp; uint32_t lu32_cur; uint32_t lu32_ticks; *(volatile uint32_t*)(STM0_BASE + STM_OFF_ISCR) = ISCR_CMP0IRR; lp_cmp = (volatile uint32_t*)(STM0_BASE + STM_OFF_CMP0); lu32_cur = *lp_cmp; lu32_ticks = g_ticks; *lp_cmp = lu32_cur + lu32_ticks; SrvSch_TimeTickMgr(); } </pre>	<pre> const STM0_BASE: u32 = 0xF000_1000; const STM_OFF_CMP0: u32 = 0x30; const STM_OFF_ISCR: u32 = 0x40; const ISCR_CMP0IRR: u32 = 1 << 0; static G_TICKS: AtomicU32 = AtomicU32::new(0); #[no_mangle] pub extern "C" fn mcal_stm_isr_logic() { unsafe { ptr::write_volatile((STM0_BASE + STM_OFF_ISCR) as *mut u32, ISCR_CMP0IRR); let lp_cmp = (STM0_BASE + STM_OFF_CMP0) as *mut u32; let lu32_cur = ptr::read_volatile(lp_cmp); let lu32_ticks = G_TICKS.load(Ordering::Relaxed); ptr::write_volatile(lp_cmp, lu32_cur.wrapping_add(lu32_ticks)); SrvSch_TimeTickMgr(); } } </pre>

- C의 #define + *(volatile *)ptr = val + cur + ticks vs. Rust의 const + ptr::write_volatile + .wrapping_add
 - C : #define X (val) (타입 없는 매크로), (volatile uint32_t*)addr 캐스트, *ptr = val 직접 대입, + 연산자 사용
 - Rust : const X: u32 = val (타입 있는 상수), addr as *mut u32 캐스트
ptr::write_volatile(p, val) 함수 호출, 산술 연산은 cur.wrapping_add(ticks) 메서드 호출
raw pointer 역참조 및 extern "C" 함수 호출은 unsafe { } 블록 안에서만 작성

II. RUST & C Integration Process

■ Cargo

- Rust의 빌드 시스템 & 패키지 매니저
- Cargo.toml(패키지 정보)를 분석해 프로젝트에 필요한 빌드 환경 자동구성

■ Cargo build Sequence

① Cargo.toml 분석



② 의존성 해결 및 다운로드



③ 빌드 스크립트(build.rs) 실행



④ rustc(Rust 컴파일러) 호출

- **Cargo.toml 분석**
 - 패키지의 이름, 버전 및 선언된 의존성(Dependencies) 목록 파악
- **의존성 해결 및 다운로드**
 - 선언된 라이브러리의 버전을 crates.io(Rust 공식 패키지 저장소)에서 다운로드
 - 이 과정에서 라이브러리들의 정확한 버전을 고정하는 Cargo.lock 생성 및 갱신
- **빌드 스크립트 실행**
 - 컴파일 전에 build.rs 파일이 있을 경우 먼저 실행
 - 주로 컴파일 시점에 필요한 코드/데이터를 자동으로 생성
- **rustc(Rust 컴파일러) 호출**
 - rustc를 호출하여 하위 크레이트부터 컴파일
- **빌드 명령어**
 - `cmake -G Ninja -DCMAKE_TOOLCHAIN_FILE=tricore.cmake -DCMAKE_BUILD_TYPE=Release -B build -S .`
 - `cmake --build build`

II. RUST & C Integration Process

■ RUST build file description (1)

- **Cargo.toml (패키지)**
 - 패키지 정보 정의 (패키지 이름, 라이브러리 타입, 소스 진입점 및 dependency)
 - 패키지마다 Cargo.toml 파일은 1개씩 존재

✓ Cargo.toml Package 구조:

```
[package]
name      = "rust-asw"
version   = "0.1.0"
edition   = "2021"

[lib]
name       = "rust_asw"
crate-type = ["rlib"]
path       = "lib.rs"

[dependencies]
rust-rte   = { path = "../RTE" }
```

- **[package]**
 - name: Cargo가 인식하는 패키지 이름
 - version: 패키지 버전 정보
 - edition: 사용할 RUST 언어 에디션 (2015/2018/2021/2024)
- **[lib]**
 - name: 생성될 라이브러리 파일명
 - crate-type: 라이브러리 타입 정의 (rlib: RUST 정적 라이브러리, staticlib: C 호환 정적 라이브러리, cdylib: C 호환 동적 라이브러리)
 - path: 진입점 경로 진입점 (src/lib.rs)
- **[dependencies]**
 - path: RTE 함수를 사용하기 위해 외부 크레이트 참조
 - C의 경우 외부 라이브러리 사용 시 #include 헤더 선언 및 실제 빌드를 위해 include 경로, 라이브러리 경로 추가 필요
 - RUST는 Cargo.toml [dependencies] 섹션에 라이브러리 추가 시 자동으로 해당 라이브러리까지 빌드

파일 트리

```
RUST_DEMO_SW_FILE
├── Cargo.toml (워크스페이스)
├── Cargo.lock
├── Cpu0_Main.c
├── ASW
│   ├── AdcMeasurement
│   │   └── adc_measurement.rs
│   ├── CanCom
│   │   └── can_com.rs
│   ├── LedBlink/LedDimming/
│   │   └── MotorControl/ (동일 패턴)
│   ├── Cargo.toml (패키지)
│   └── lib.rs
├── BSW
│   ├── CDD/CddMotorHandler
│   │   ├── Motor_handler.rs
│   │   └── Cargo.toml (패키지)
│   ├── ECAL
│   │   ├── CanIf/EcalIf/IoHwAb/
│   │   │   └── Cargo.toml (패키지)
│   │   └── lib.rs
│   ├── MCAL
│   │   ├── Adc
│   │   │   ├── bindings.rs
│   │   │   └── mcal_adc.rs
│   │   ├── CAN
│   │   │   ├── bindings.rs
│   │   │   └── mcal_can.rs
│   │   ├── Ccu6/Dio/Pwm/Stm/
│   │   │   └── (동일 패턴)
│   │   ├── Cargo.toml (패키지)
│   │   └── lib.rs
│   └── SRV
│       ├── lib.rs
│       ├── com.rs, ecum.rs, sch.rs
│       └── Cargo.toml (패키지)
├── RTE
│   ├── rte.rs
│   └── rte_cfg.rs
│   └── Cargo.toml (패키지)
└── bsw-combined
    ├── Cargo.toml (패키지)
    └── lib.rs
```

※ C : #include + Mailefile -I lib_path + -l lib_name

II. RUST & C Integration Process

■ RUST build file description (2)

■ Cargo.toml (워크스페이스)

- 프로젝트 내 여러 패키지를 하나의 작업 공간으로 묶어 통합 관리하는 설정 파일
- 이 파일을 통해 cargo build 한 번으로 전체 패키지 일괄 빌드

✓ Cargo.toml Workspace 및 Profile 구조

```
[workspace]
Members      = [ "ASW",
                  "BSW/CDD/CddMotorHandler",
                  "BSW/ECAL",
                  "BSW/MCAL",
                  "BSW/SRV",
                  "RTE",
                  "bsw-combined", ]
resolver     = "2"

[profile.release]
panic        = "abort"
opt-level    = "s"
lto          = true
codegen-units = 1

[profile.dev]
panic        = "abort"
```

■ [workspace]

- members: 워크스페이스에 포함할 패키지 경로 목록
- resolver: 의존성 resolver 버전

■ [profile.release] – release 빌드 시 설정

- panic: 패닉(런타임 오류) 발생 시 처리 방식 설정
(unwind - 스택을 올라가며 각 단계에서 사용한 메모리 정리 >> .elf 파일의 크기 ↑,
abort - 즉시 프로그램 중단 >> .elf 파일의 크기 ↓)
- opt-level: 컴파일러 최적화 수준 설정
(0~3 - 값이 클수록 컴파일 속도 ↓, 실행 속도 ↑,
s - 코드 크기 최적화)
- lto: 링크 타임 최적화 여부 설정
- codegen-units: 컴파일 시 병렬 코드 생성 단위 수 설정

■ [profile.dev] – develop 빌드 시 설정

파일 트리

```
RUST_DEMO_SW_FILE
├── Cargo.toml(워크스페이스)
├── Cargo.lock
├── Cpu0_Main.c
├── ASW
│   ├── AdcMeasurement
│   │   └── adc_measurement.rs
│   ├── CanCom
│   │   └── can_com.rs
│   ├── LedBlink/LedDimming/
│   │   └── MotorControl/(동일 패턴)
│   ├── Cargo.toml(패키지)
│   └── lib.rs
├── BSW
│   ├── CDD/CddMotorHandler
│   │   ├── Motor_handler.rs
│   │   └── Cargo.toml(패키지)
│   ├── ECAL
│   │   ├── CanIf/EcalIf/IoHwab/
│   │   └── Cargo.toml(패키지)
│   │   └── lib.rs
│   ├── MCAL
│   │   ├── Adc
│   │   │   ├── bindings.rs
│   │   │   └── mcal_adc.rs
│   │   ├── CAN
│   │   │   ├── bindings.rs
│   │   │   └── mcal_can.rs
│   │   ├── Ccu6/Dio/Pwm/Stm/
│   │   │   └── (동일 패턴)
│   │   ├── Cargo.toml(패키지)
│   │   └── lib.rs
│   └── SRV
│       ├── lib.rs
│       ├── com.rs, ecum.rs, sch.rs
│       └── Cargo.toml(패키지)
├── RTE
│   ├── rte.rs
│   ├── rte_cfg.rs
│   └── Cargo.toml(패키지)
└── bsw-combined
    ├── Cargo.toml(패키지)
    └── lib.rs
```

II. RUST & C Integration Process

■ RUST build file description (3)

- Cargo.lock (의존성 고정)
 - Dependency의 모든 라이브러리 버전을 기록한 스냅샷 파일
 - Cargo가 빌드 시 자동으로 생성 및 갱신
 - 다른 PC, 다른 시점에서도 **동일한 산출물 생성 (재현성 보장)**
 - 'It is not intended for manual editing' → 수동 편집 미허용

✓ Cargo.lock 구조

```
# This file is automatically @generated by Cargo.
# It is not intended for manual editing.
[[package]]
name      = "rust-mcal"
version   = "0.1.0"
dependencies = [
    "rust-cdd 0.1.0 (path+file:///.../CddMotorHandler)",
]
```

파일 트리

```
RUST_DEMO_SW_FILE
├── Cargo.toml(워크스페이스)
├── Cargo.lock
├── Cpu0_Main.c
├── ASW
│   ├── AdcMeasurement
│   │   └── adc_measurement.rs
│   ├── CanCom
│   │   └── can_com.rs
│   ├── LedBlink/LedDimming/
│   │   └── MotorControl/(동일 패턴)
│   ├── Cargo.toml(패키지)
│   └── lib.rs
├── BSW
│   ├── CDD/CddMotorHandler
│   │   ├── Motor_handler.rs
│   │   └── Cargo.toml(패키지)
│   ├── ECAL
│   │   ├── CanIf/EcalIf/IoHwab/
│   │   │   └── Cargo.toml(패키지)
│   │   └── lib.rs
│   ├── MCAL
│   │   ├── Adc
│   │   │   ├── bindings.rs
│   │   │   └── mcal_adc.rs
│   │   ├── CAN
│   │   │   ├── bindings.rs
│   │   │   └── mcal_can.rs
│   │   ├── Ccu6/Dio/Pwm/Stm/
│   │   │   └── (동일 패턴)
│   │   ├── Cargo.toml(패키지)
│   │   └── lib.rs
│   └── SRV
│       ├── lib.rs
│       ├── com.rs, ecum.rs, sch.rs
│       └── Cargo.toml(패키지)
├── RTE
│   ├── rte.rs
│   ├── rte_cfg.rs
│   └── Cargo.toml(패키지)
└── bsw-combined
    ├── Cargo.toml(패키지)
    └── lib.rs
```

II. RUST & C Integration Process

■ RUST build file description (4)

파일 트리

■ bindings.rs

- C로 작성된 정의들을 RUST에서 사용할 수 있도록 다시 선언해주는 파일

✓ bindings.rs 형식

```
#[repr(C)]
#[derive(Copy, Clone, Default)]
pub struct IfxCan_Message {
    pub bufferNumber: u8,
    _pad: [u8; 3],           // padding 명시
    pub messageId: u32,
    /* ... */
}

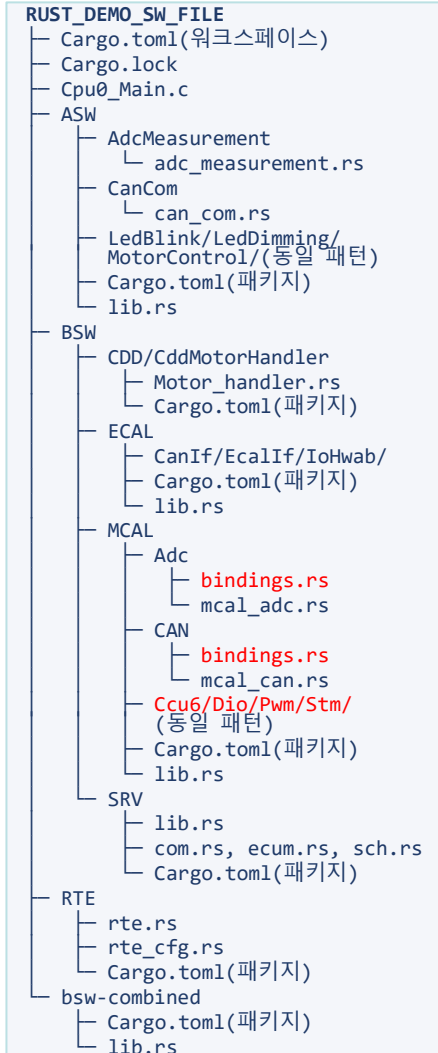
extern "C" {
    pub(super) fn IfxPort_setPinMode(
        port: *mut u8, pin_index: u8, mode: u32);
    pub static mut gu8_flag: u8;
}
```

■ **#[repr(C)]**

- RUST 컴파일러는 성능 최적화를 위해 구조체 멤버 순서를 재배열 가능
→ C와 같은 메모리 배치를 보장하기 위해
#[repr(C)] 속성을 붙여 재배열 방지 및 구조체 호환

■ **extern "C" 블록**

- RUST에서 C 함수 및 전역 변수를 부르기 위한 선언
- extern "C"로 호출되는 C 함수는 Rust 컴파일러가 안정성을 검증할 수 없으므로 unsafe 블록 안에서만 호출 가능
- 안정성을 검증 받지 못한 코드들은 unsafe 블록에 격리되어 코드 리뷰 시 **unsafe 블록** 위주로 검토하여 검토 효율 향상



II. RUST & C Integration Process

■ RUST build file description (5)

■ lib.rs

- 라이브러리의 진입점
- 하위 모듈을 pub mod로 포함
- crate 전체에 적용할 속성 선언

✓ lib.rs 구조

```
#![allow(non_snake_case)]
#![allow(dead_code)]

#[path = "Dio/mcal_dio.rs"]
pub mod mcal_dio;

#[path = "Can/mcal_can.rs"]
pub mod mcal_can;

#[path = "Adc/mcal_adc.rs"]
pub mod mcal_adc;
```

■ #![allow()]

- Rust 컴파일러가 dead code, naming rule, 미사용 import 등 자동으로 경고
- 휴먼 에러를 컴파일 단계에서 차단해 **코드 품질 유지**

■ #[path = "..."]

- 다음에 오는 모듈의 소스 파일 경로 설정
- C언어 .h, .c, 빌드 스크립트(.gpj/makefile) 에 모듈 정보가 분산되어 신규 모듈 추가 시 3파일 수정 필요
- Rust언어 lib.rs 한 곳에서 **pub mod 선언으로 추가 완료**

■ pub mod ...

- mod: 하위 모듈을 컴파일 대상으로 포함
- pub: 외부 crate에 공개 (pub이 없는 경우 crate 내부에서만 해당 모듈 호출 가능)

파일 트리

```
RUST_DEMO_SW_FILE
├─ Cargo.toml(워크스페이스)
├─ Cargo.lock
├─ Cpu0_Main.c
├─ ASW
│   ├── AdcMeasurement
│   │   └─ adc_measurement.rs
│   ├── CanCom
│   │   └─ can_com.rs
│   ├── LedBlink/LedDimming/
│   │   └─ MotorControl/(동일 패턴)
│   ├── Cargo.toml(패키지)
│   └─ lib.rs
├─ BSW
│   ├── CDD/CddMotorHandler
│   │   ├── Motor_handler.rs
│   │   └─ Cargo.toml(패키지)
│   └─ ECAL
│       ├── CanIf/EcalIf/IoHwab/
│       │   └─ Cargo.toml(패키지)
│       └─ lib.rs
├─ MCAL
│   ├── Adc
│   │   ├── bindings.rs
│   │   └─ mcal_adc.rs
│   ├── CAN
│   │   ├── bindings.rs
│   │   └─ mcal_can.rs
│   ├── Ccu6/Dio/Pwm/Stm/
│   │   └─ (동일 패턴)
│   ├── Cargo.toml(패키지)
│   └─ lib.rs
├─ SRV
│   ├── lib.rs
│   ├── com.rs, ecum.rs, sch.rs
│   └─ Cargo.toml(패키지)
├─ RTE
│   ├── rte.rs
│   ├── rte_cfg.rs
│   └─ Cargo.toml(패키지)
└─ bsw-combined
    ├── Cargo.toml(패키지)
    └─ lib.rs
```

II. RUST & C Integration Process

■ Build Process



※ Cmake: 프로젝트 구성 정보(CMakeLists.txt)를 읽어 Ninja 전용 빌드 파일(build.ninja) 생성
※ Ninja: 생성된 build.ninja 파일을 기반으로 소스 코드를 컴파일하여 최종 결과물(.elf) 생성

III. Implementation - Timer Scheduler

■ Mcal Stm

- STM0 Compare Match를 이용한 1ms 주기 시스템 타이머 드라이버
- 1ms마다 ISR이 실행되어 전체 시스템의 시간 기준(Tick)을 생성하여 스케줄러에 공급

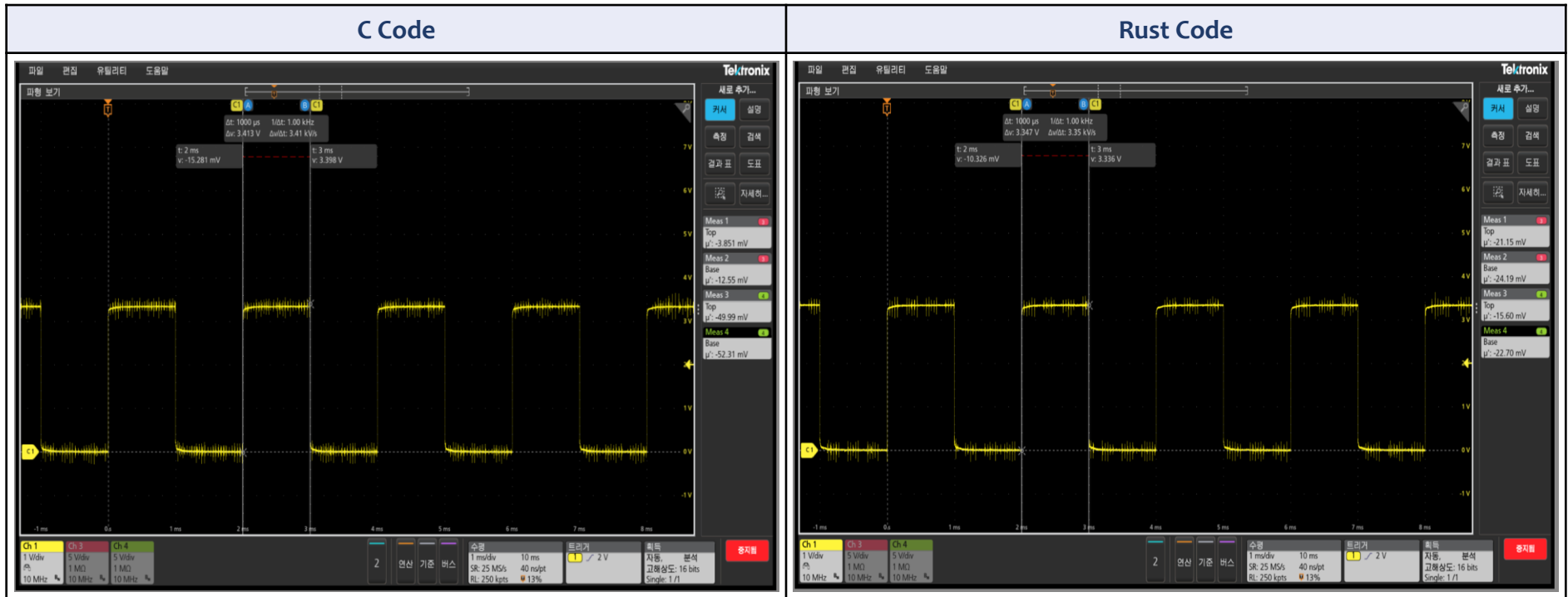
C Code	Rust Code
<pre>void IfxStm_clearCompareFlag(Ifx_STM *stm, IfxStm_Comparator comparator) { if (comparator == IfxStm_Comparator_0) { stm->ISCR.B.CMP0IRR = 1U; } else if ... } void IfxStm_increaseCompare(Ifx_STM *stm, IfxStm_Comparator comparator, uint32 ticks) { stm->CMP[comparator].B.CMPVAL = stm->CMP[comparator].B.CMPVAL + ticks; }</pre>	<pre>extern "C" { fn IfxStm_clearCompareFlag(stm: *mut u8, comparator: u32); fn IfxStm_increaseCompare(stm: *mut u8, comparator: u32, ticks: u32); } pub fn mc1_stm_isr_logic() { unsafe { IfxStm_clearCompareFlag(STM0_BASE, STM_COMPARATOR_0); let lu32_ticks: u32 = G_TICKS.load(Ordering::Relaxed); IfxStm_increaseCompare(STM0_BASE, STM_COMPARATOR_0, lu32_ticks); SrvSch_TimeTickMgr(); } }</pre>

- Rust에서 기존 C 라이브러리(iLLD) 함수 호출
 - extern "C" 선언, iLLD 함수를 RUST에서 호출 가능
 - Rust언어 G_TICKS, AtomicU32로 관리 → 데이터 보호

III. Implementation - Timer Scheduler

■ Mcal Stm

- C 및 Rust 기반 Scheduler 1ms 주기 Task Pin Toggle 동작 결과
 - 동일 동작 확인



III. Implementation - Timer Scheduler

■ Srv Sch

- 1ms Tick 기반의 태스크 스케줄러로, 15개의 주기별 태스크(1ms/5ms/10ms/20ms 등)를 관리
- McalStm의 1ms 주기 ISR이 Tick 플래그를 ON으로 설정하면 메인 loop에서 각 태스크의 주기·오프셋을 확인하여 실행 시점에 도달한 태스크의 함수 포인터 호출

C Code	Rust Code
<pre>void SrvSch_Scheduler(void) { ... for(1_u16TaskIndex=0; 1_u16TaskIndex<NUM_OF_TASK; 1_u16TaskIndex++) { if(gast_TaskInfo[1_u16TaskIndex].Activation == STD_ACTIVE) { if (gast_TaskInfo[1_u16TaskIndex].FuncPtr != STD_NULL_POINTER) { gast_TaskInfo[1_u16TaskIndex].FuncPtr(); } } } ... }</pre>	<pre>pub fn SrvSch_Scheduler() { ... for lu_idx in 0..NUM_OF_TASK { if TASK_INFO[lu_idx].activation == STD_ACTIVE { if let Some(lp_func) = TASK_INFO[lu_idx].func_ptr { lp_func(); } } } ... }</pre>

- **C의 NULL 포인터 vs Rust의 Option 타입**
 - C에서는 함수 포인터가 NULL인지 개발자가 직접 확인 필요, 누락 시 컴파일이 통과되어 런타임에 NULL 포인터를 호출할 가능성 잔존
 - RUST의 Option<fn()> 타입은 값이 없는 상태(None)을 타입 자체에 포함하여, if let Some(func) 구문으로 유효성 검증 없이는 호출할 수 없도록 컴파일 단계에서 강제

III. Implementation - CAN Communication

■ Mcal Can

- CAN-FD 노드 설정 및 TX / RX 프레임 송수신
- Bus-Off 복구 처리 및 RX 타임아웃 체크

C Code	Rust Code
<pre>void McalCan_SendMessage(IfxCan_Message *lpst_TxMsg, uint32_t *lpu32_TxData) { ... Ifx_CAN_TXMSG *txBufferElement = IfxCan_Node_getTxBufferElementAddress(...); txBufferElement->T0.B.ID = lpst_TxMsg->messageId; txBufferElement->T0.B.RTR = lpst_TxMsg->remoteTransmitRequest; ... node->TX.BAR.U = (1U << bufferId); } void McalCan_MainFunction(void) { McalCan_CheckBusOff(); for(uint8_t lu8_box = 0U; lu8_box < MCAL_NUM_RX_BOXES; lu8_box++) { if(gu16_RxTimeoutCounter[lu8_box] < MCAL_RX_TIMEOUT_TIME) { gu16_RxTimeoutCounter[lu8_box]++; } else { McalCan_init(); } } }</pre>	<pre>pub fn McalCan_SendMessage(lr_msg: &IfxCan_Message, lp_tx_data: *const u32) { ... let lu32_t0: u32 = ((lr_msg.messageId & STD_ID_MASK_11BIT) << RX_R0_STD_ID_SHIFT) ...; let lu32_buf_addr = CAN0_MRAM_BASE + MRAM_TX_BUFFER_OFFSET; unsafe { ptr::write_volatile(lu32_buf_addr as *mut u32, lu32_t0); node_reg_write(NODE_OFF_TXBAR, lu32 << 0); } } pub fn McalCan_MainFunction() { McalCan_CheckBusOff(); ... if lu16_cnt < MCAL_RX_TIMEOUT_TIME { GA_RX_TIMEOUT_COUNTER[lu_box].store(lu16_cnt + 1, Ordering::Relaxed); } else { McalCan_init(); } }</pre>

- C의 void* 인자 vs Rust의 &T reference
 - C는 void* 또는 raw pointer 인자가 어디서든 역참조가 가능하기 때문에 null / dangling / 타입 오류가 silent Undefined Behavior로 이어짐
 - Rust는 &T reference로 받으면 컴파일러가 null 불가능 + 유효 메모리 + 정확한 타입을 보장하므로 null 검사가 불필요
 - MCU 레지스터 접근과 같은 컴파일러가 검증하지 못한 코드만 unsafe { } 블록으로 한정

III. Implementation - CAN Communication

■ Ecal CanIf

- TX : 전송 프레임 헤더(MsgId/DLC/FrameMode) 조립 후 MCAL로 전달
- RX : 수신 데이터와 CAN 상태 확인 함수를 상위 계층에 전달

C Code	Rust Code
<pre> void EcalCanIf_Transmit(uint8_t lu8_SetIdx, const uint8_t *lpu8_TxData) { if(lu8_SetIdx < ECALCANIF_NUM_SETS) { IfxCan_Message lst_TxMsg = { .bufferNumber = 0, .messageId = 0x0, .messageIdLength = IfxCan_MessageIdLength_standard, .dataLengthCode = IfxCan_DataLengthCode_8, .frameMode = IfxCan_FrameMode_standard, ... }; lst_TxMsg.messageId = TX_MSG_ID[lu8_SetIdx]; lst_TxMsg.dataLengthCode = IFXCAN_DATALENGTHCODE_32; lst_TxMsg.frameMode = IfxCan_FrameMode_fdLongAndFast; McalCan_SendMessage(&lst_TxMsg, (uint32_t*)lpu8_TxData); } ... } </pre>	<pre> pub fn EcalCanIf_Transmit(lu8_set_index: u8, lp_tx_data: *const u8) { let lu_set_index = lu8_set_index as usize; if lu_set_index < ECALCANIF_NUM_SETS as usize { let mut lst_tx_msg = IfxCan_Message::default(); lst_tx_msg.messageId = TX_MSG_ID[lu_set_index]; lst_tx_msg.messageIdLength = IFXCAN_MESSAGEIDLENGTH_STANDARD; lst_tx_msg.dataLengthCode = IFXCAN_DATALENGTHCODE_32; lst_tx_msg.frameMode = IFXCAN_FRAMEMODE_FDLONGANDFAST; lst_tx_msg.bufferNumber = lu8_set_index; McalCan_SendMessage(&lst_tx_msg, lp_tx_data as *const u32); } ... } </pre>

- C의 **memset 0-init** vs Rust의 **#[derive(Default)]**
 - C는 **memset(&msg, 0, sizeof(msg))** 또는 멤버별 초기화가 필요하며 누락 시 미정의 값으로 동작
 - Rust는 **#[derive(Default)]**가 적용된 구조체 선언 시 컴파일러가 **자동으로 0-init 함수 합성 ::default()** 호출 시 모든 필드 0-init을 컴파일 타임에 보장

■ **ASW CanCom**

- | C Code | Rust Code |
|---|---|
| <pre>static uint8_t gu8_Rx_Buffer[SRVCOM_NUM_SETS][CANCOM_TXDATA_LEN]; /* gu8_Rx_Buffer[2][32]; */ void CanCom_MainFunction(void) { uint8_t lu8_Set; uint8_t lu8_Idx; for(lu8_Set = 0U; lu8_Set < SRVCOM_NUM_SETS; lu8_Set++) { if(Rte_IsRxUpdated(lu8_Set) == STD_ON) { for(lu8_Idx = 0U; lu8_Idx < CANCOM_TXDATA_LEN; lu8_Idx++) { gu8_Rx_Buffer[lu8_Set][lu8_Idx] = Rte_GetRxData(lu8_Set, lu8_Idx); } ... } } }</pre> | <pre>static GU8_RX_BUFFER: [[AtomicU8; CANCOM_TXDATA_LEN]; SRVCOM_NUM_SETS] = [ZERO_BUF; SRVCOM_NUM_SETS]; /* GU8_RX_BUFFER[[AtomicU8; 32]; 2] */ pub extern "C" fn CanCom_MainFunction() { for lu_set in 0..SRVCOM_NUM_SETS { if Rte_IsRxUpdated(lu_set as u8) == STD_ON { for lu_idx in 0..CANCOM_TXDATA_LEN { let lu8_TempData = Rte_GetRxData(lu_set as u8, lu_idx as u8); GU8_RX_BUFFER[lu_set][lu_idx].store(lu8_TempData, Ordering::Relaxed); } ... } } }</pre> |
| <pre>ctc W561: ["../ASW/CanCom.c" 102/22] access out of bounds 0 errors, 1 warnings ... Finished building target: AURIX_Demo.elf ... 15:42:07 Build Finished. 0 errors, 1 warnings. (took 5s.737ms)</pre> | <pre>error: this operation will panic at runtime --> ASW\CanCom\can_com.rs:121:17 ... 121 GU8_RX_BUFFER[3][lu_idx].store(lu8_TempData, Ordering::Relaxed); ^^^ index out of bounds: the length is 2 but the index is 3 ... error: could not compile 'rust-asw' (lib) due to 1 previous error ninja: build stopped: subcommand failed.</pre> |

- 

III. Implementation – Motor Control

■ Mcal Ccu6

- CCU60 모듈을 이용하여 모터의 3상을 PWM으로 출력 제어
- 6-Step 모터 제어

C Code	Rust Code
<pre>void McalCcu6_SetCommStep(uint8_t lu8_Step, uint8_t lu8_Duty) { ... switch (lu8_Step) { case 0U: /* U High, V Low, W floating */ MODULE_CCU60.MODCTR.B.T12MODEN = MODEN_UPWM_VRET; MODULE_CCU60.CC6SR[0].B.CCS = lu16_OnTick; MODULE_CCU60.CC6SR[1].B.CCS = gu16_T12Period +1U; MODULE_CCU60.CC6SR[2].B.CCS = break; } ... }</pre>	<pre>pub fn McalCcu6_SetCommStep(lu8_step: u8, lu8_duty: u8) { ... match lu8_step { 0 => { /* U High, V Low, W floating */ reg_modify(lp_ccu6, MODCTR_OFFSET, MODCTR_T12MODEN_MASK, MODEN_UPWM_VRET); reg_write(lp_ccu6, CC6SR0_OFFSET, lu16_on_tick as u32); ... } 1 => ... } unsafe fn reg_write(lp_base: *mut u8, lu_offset: usize, lu32_value: u32) { (lp_base.add(lu_offset) as *mut u32).write_volatile(lu32_value); }</pre>

- C의 'switch' vs. Rust의 'match'
 - C의 switch는 default 및 break를 생략해도 컴파일이 통과되어 잘못된 step 값에 대해 처리 없이 넘어갈 수 있음
 - RUST의 match는 모든 경우를 처리해야 컴파일 가능 (Exhaustiveness Check)

III. Implementation – Motor Control

■ Cdd MotorHandler

- 상태 머신(Stop → Ready → Ramp → Run)을 통해 모터 구동 시퀀스 관리
- 1ms 주기로 호출되는 MainFunction에서 현재 상태에 해당하는 핸들러 실행

C Code	Rust Code
<pre>static void CddMotorHandler_StateRun(void) { gu32_CommTimer++; if (gu32_CommTimer >= gu32_CommPeriod) { gu32_CommTimer = 0U; CddMotorHandler_NextCommStep(); } McalCcu6_SetCommStep(gu8_CommStep, gu8_Duty); }</pre> <u>C 전역변수 증가 asm</u> <pre>12659: movh.a %a2, 28672 12661: ld.w %d2, [%a2] → d2에 gu32_CommTimer 값 저장 12662: add %d2, 1 → d2 = d2 + 1 12665: st.w [%a2], %d2 → 메모리에 값 저장</pre>	<pre>fn state_run() { let lu32_comm_timer = GU32_COMM_TIMER.fetch_add(1, Ordering::Relaxed) + 1; let lu32_comm_period = GU32_COMM_PERIOD.load(Ordering::Relaxed); if lu32_comm_timer >= lu32_comm_period { GU32_COMM_TIMER.store(0, Ordering::Relaxed); next_comm_step(); } ... }</pre> <u>Atomic 전역변수 증가 asm</u> <pre>5927: movh.a %a15, 28672 5928: ld.w %d1, [%a15], lo:GU32_COMM_TIMER → d1에 GU32_COMM_TIMER 값 저장 5929: lea %a2, [%a15], lo:GU32_COMM_TIMER → a2에 GU32_COMM_TIMER 주소 저장 5930: add %d0, %d1, 1 (d0 = d1 + 1) 5931: mov %e2, %d1, %d0 → 레지스터 e2에 (예상 값=d1, 새 값=d0) 준비 5932: cmpswap.w [%a2], %e2 → [%a2]의 값이 d1이면 성공, 아니면 실패/ 결과값은 d2에 반환됨 5933: eq %d15, %d2, %d1 → d2(cmpswap 결과) == d1(예상 값)이면 d15=1(성공), d15=0(실패) 5935: jz.t %d15, 0, 5927 → d15가 0(=실패)이면, 5927로 돌아가서 재시도</pre>

- C의 전역 변수 공유 vs Rust의 Atomic
 - CPU는 메모리에서 직접 연산을 할 수 없어 읽기 → 수정 → 쓰기 3단계로 동작하며, 단계 사이에 ISR이 개입할 경우 데이터가 손실될 수 있음
 - C의 volatile은 원자성을 보장하지 않아 ISR 선점 시 불완전한 값이 관측될 수 있음
 - RUST의 Atomic은 CPU의 **cmpswap.w** 명령어를 활용하여, 쓰기 시점에 값이 변경되었는지 비교 후 변경되었으면 전체 과정을 재시도

III. Implementation – Motor Control

■ High U/V/W PWM & EN_GATE Output



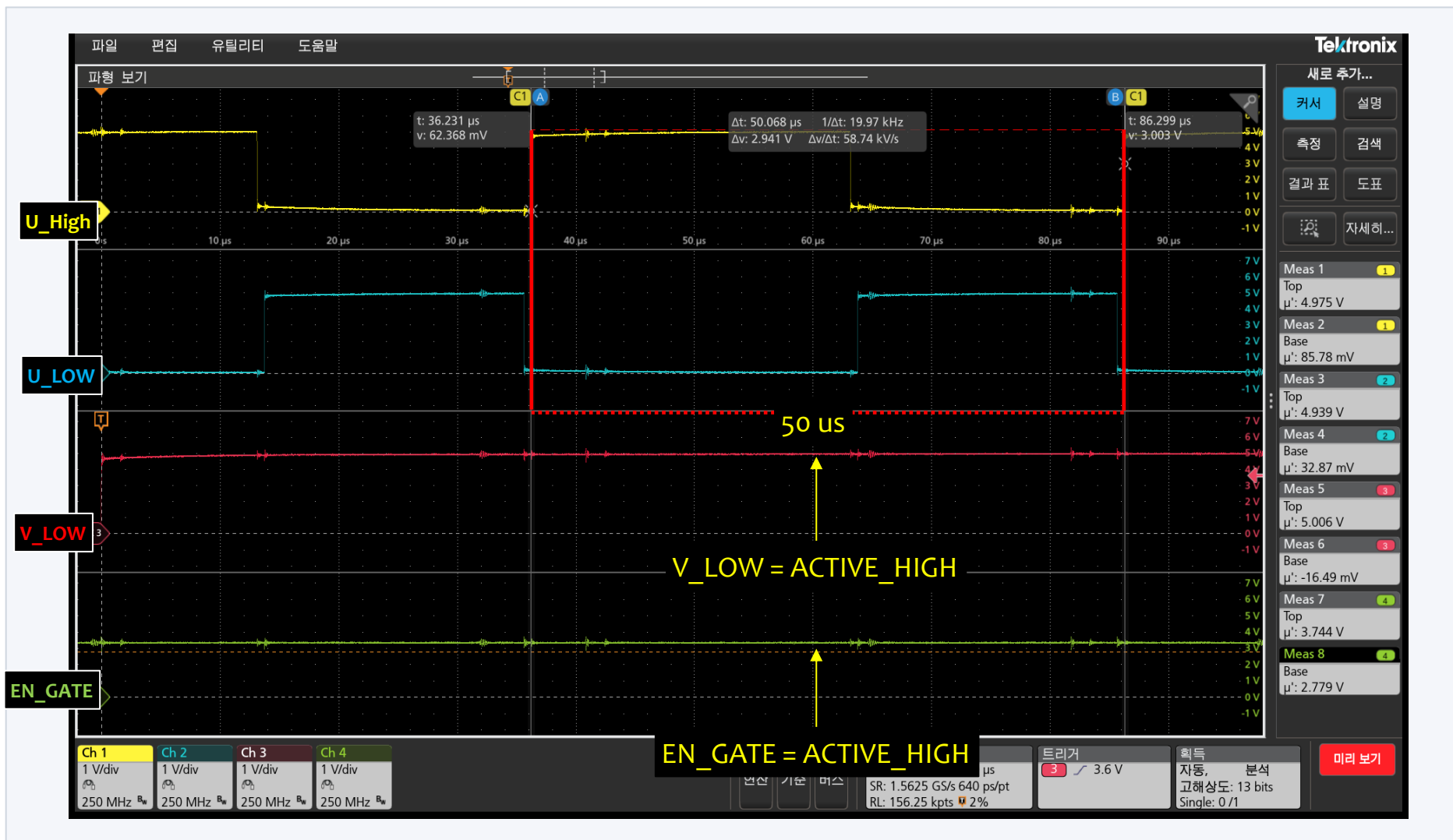
III. Implementation – Motor Control

■ Low U/V/W PWM Output



III. Implementation – Motor Control

■ Step-0 Output



IV. Motor Control Software Structure

■ Motor Control SW Table

※ GTM : Generic Timer Module
 ※ TOM : Timer Output Module
 ※ ATOM : Advanced Timer Output Module

※MCU 레지스터 접근이 필요한 MCAL 계층에 `unsafe {}` 블록을 집중시키고, 상위 계층(ECAL/SRV/RTE/ASW)은 안전한 Rust 코드로 개발 추천

Level1	Level2	Level3	Language	Description
ASW	AdcMeasurement	-	Rust	EVADC 센서로부터 읽은 12-bit Raw Data를 0~5V 사이의 물리적 전압 단위로 변환
	CanCom	-	Rust	2세트(RX 0x100/0x200, TX 0x101/0x201)의 CAN 메시지 송수신 처리
	LedBlink	-	Rust	8개의 LED 순차 점등 제어
	LedDimming	-	Rust	PWM 듀티 사이클 조절로 8개의 LED 밝기 제어
	MotorControl	-	Rust	버튼 입력을 받아 RTE를 통해 CDD MotorHandler에 모터 제어 명령 전달
RTE	Rte	-	Rust	ASW Init/Task 함수를 Scheduler에 등록 및 실행, ASW↔BSW간 I/O 추상화 래퍼
	RteCfg	-	Rust	RTE에서 사용할 ASW Init 함수 및 주기별 Task 함수 매핑 테이블 정의
BSW	SRV	ComM	Rust	EcalCanIf를 통해 CAN 메시지 송수신 및 Bus-Off 상태 관리
		EcuM	Rust	전체 BSW 모듈 초기화
		Sch	Rust	STM ISR로부터 1ms 플래그를 받아 TASK의 주기 도달 여부 판정 및 RteSch_TaskXXX 함수 호출
	CDD	MotorHandler	Rust	BLDC 모터 제어, LED 및 S2o2 버튼 핀 초기화
	ECAL	CanIf	Rust	TX 프레임 헤더(MsgId/DLC/FrameMode) 조립 및 MCAL 호출, RX/Status 액세스 래퍼 제공
		EcalIf	Rust	ADC, DIO, PWM, CAN 등 ECAL 하위 모듈의 초기화 함수 일괄 호출
		IoHwAb	Rust	ADC 읽기, DIO 토글, PWM 듀티 설정 등 런타임 I/O 하드웨어 추상화 계층
	MCAL	Adc	Rust	EVADC 모듈의 그룹/채널 설정 및 변환 수행
		Can	C / Rust	CAN-FD 노드 설정, TX/RX 및 Bus-Off 복구 처리
		Ccu6	Rust	CCU6o 모듈로 3상 PWM 출력 제어
		Dio	Rust	디지털 I/O 핀의 입출력 방향/모드 설정 및 상태 read/write를 PORT 레지스터 직접 제어로 수행
		Pwm	Rust	GTM TOM, ATOM 채널을 통한 PWM 신호 생성 및 듀티 제어
		Stm	C / Rust	STMo Compare Match로 1ms 주기 인터럽트 생성하여 SrvSch_TimeTickMgr 호출
		iLLD	C	AURIX peripheral 레지스터 접근 API 라이브러리 (Infineon 공식 제공)
Cpuo_Main.c	-	-	C	Cpu Core_o 진입점, HW 초기화 후 Rust 스케줄러 메인 루프 호출

V. RUST Advantages

■ Rust Advantages

※ GPIO : General Purpose Input Output ※ RMW : Read-Modify-Write
※ FFI : Foreign Function Interface ※ RAI : Resource Acquisition Is Initialization
※ SDK : Software Development Kit



강력한 정적 분석

컴파일 단계에서 하드웨어 자원의 소유권을 검증하여 핀 중복 설정 및 잘못된 레지스터 접근을 차단

- Peripheral Safety : GPIO 중복 할당을 컴파일 타임에 감지
- MCU Abstraction : 잘못된 레지스터 접근을 사전 방지
- Array Bounds Check : 배열 인덱스 OOB 접근을 컴파일/런타임에 자동 검출
- Pointer Safety : Null · Dangling 포인터 역참조를 컴파일 단계에서 차단



유연한 메모리 관리

Heap 사용이 선택적이며, 동적 할당 없이도 안전한 메모리 관리가 가능하여 저사양 MCU에 최적화

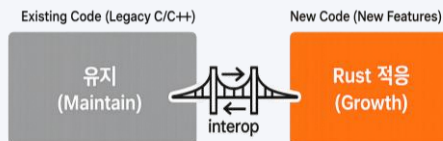
- `#![no_std]` : 표준 라이브러리 없이 베어메탈 환경 지원
- Static Allocation : 메모리 단편화 없이 예측 가능한 실시간성 보장
- Borrow Checker : Garbage Collector 없이 컴파일 타임에 메모리 안전성 보장
- RAI : 스코프 종료 시 Drop trait이 자동 호출되어 자원 해제



안전한 동시성 제어

데이터 레이스를 컴파일러가 차단하여 인터럽트(ISR)와 메인 루프 간 안전한 데이터 공유를 보장

- Interrupt Safety : ISR/Task 컨텍스트 간 공유 자원에 대한 안전한 접근 제어
- Data Race Free : Atomic 타입과 Memory Ordering으로 race condition 방지
- Send/Sync Trait : Thread/ISR 컨텍스트 간 데이터 이동 · 공유 안전성 검증
- Lock-Free RMW : `fetch_add/compare_exchange`로 원자적 RMW 처리



뛰어난 상호 운용성

기존 C 언어 코드베이스나 제조사 SDK와 원활히 통합되며 점진적인 마이그레이션 지원

- Zero-cost FFI : 오버헤드 없이 기존 C 라이브러리 호출
- C Bindings : C 헤더 정의를 `bindings.rs`에 재선언하여 Rust에서 호출
- `#[repr(C)]` : C ABI 호환 메모리 레이아웃으로 struct 직접 공유
- Symbol Preservation : `extern "C" + #[no_mangle]`로 C 링커가 원래 이름 참조

고맙습니다



세온이앤에스
Seon ENS, Inc.

E n g i n e e r i n g a n d S o l u t i o n s

www.seonens.com