2주차 보고서:
RTOS 기반 PMIC 모니터링 및 차량 진단 제어기 개발

최용진

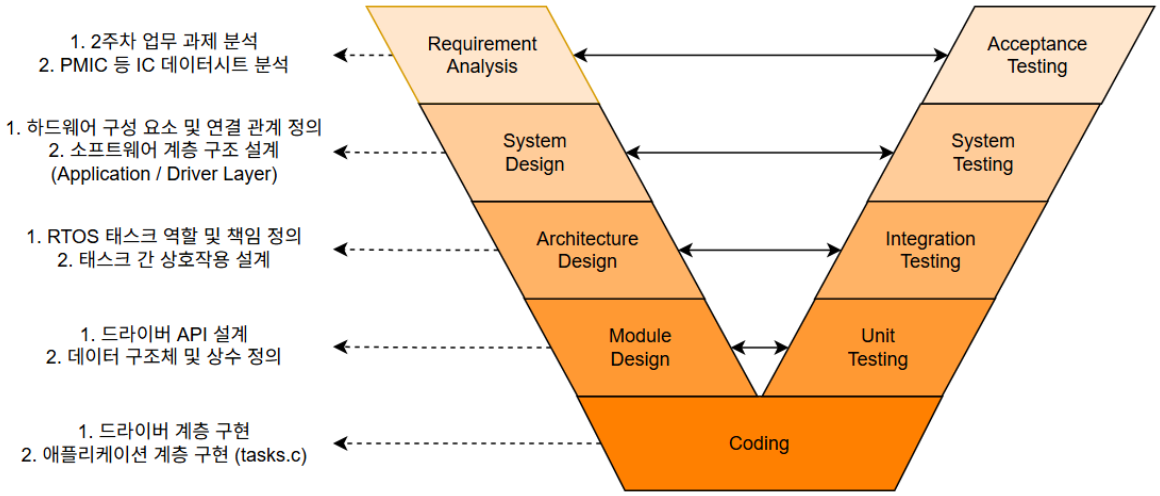
2025.08.01

목차

1. 2주차 업무 개요

1.1. 프로젝트 목표

본 프로젝트는 자동차 소프트웨어 개발 표준인 **V-모델 좌측**에 따라 요구사항 분석부터 시스템 설계, 상세 설계 및 구현 단계를 체계적으로 수행한다. V-모델을 적용한 2주차 개발 계획을 github docs 폴더에 게시해놓았다.



STM32F413ZHTx MCU를 기반으로 **RTOS** 환경을 구축하고, 여러 태스크를 동기화하여 **전력 관리 반도체(PMIC)**의 상태를 실시간으로 모니터링한다.

PMIC에서 감지된 Fault 정보와 ADC를 통해 측정한 전압 이상 상태를 **DTC(고장 진단 코드)**로 정의하여 **EEPROM**에 저장하고, **CAN** 통신을 통해 외부 진단기의 요청에 따라 해당 DTC 정보를 조회하거나 삭제하는 차량 진단 제어기를 개발한다.

1.2. 주요 개발 내용:

1. **RTOS 기반 멀티태스킹 환경 구축:**
   - 1ms 주기 태스크: I2C, SPI, CAN 통신 처리
   - 5ms 주기 태스크: UART, ADC 처리
   - Mutex를 이용한 태스크 간 공유 자원 동기화
2. **주변 장치 드라이버 개발:**
   - I2C (DMA 방식): PMIC(MP5475GU) 제어
   - SPI (DMA 방식): EEPROM(25LC256) 데이터 읽기/쓰기

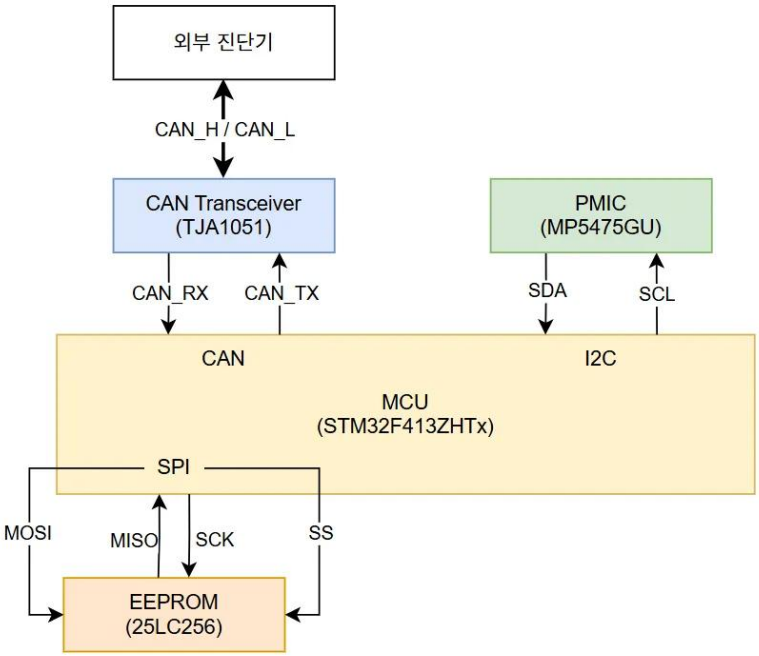
- CAN (Interrupt 방식): 외부 진단기 통신
- UART (Polling 방식): 시스템 상태 메시지 출력
- ADC: PMIC 특정 채널(BUCK D) 전압 모니터링

3. **핵심 진단 로직 구현:**
   - PMIC의 UV(저전압), OC(과전류) Fault 레지스터를 주기적으로 확인하여 DTC 생성
   - ADC로 PMIC 전압을 모니터링하고, 설정된 임계값 초과 시 DTC 생성
   - 생성된 DTC를 EEPROM에 저장 및 관리
   - CAN 메시지 수신 시, 요청에 따라 EEPROM의 DTC 정보 조회 및 삭제 기능 수행
4. **코드 관리 및 문서 저장:**
   - Git을 사용한 버전 관리 및 GitHub 업로드

2. 시스템 설계

2.1. 하드웨어 구성도

다음은 MCU와 PMIC, EEPROM, CAN 트랜시버 간의 연결 관계를 나타내는 블록 다이어그램이다.



- **MCU ↔ PMIC (I2C 통신):**
  - SDA와 SCL 라인을 통해 연결된다.
  - MCU는 I2C 마스터로서 PMIC의 내부 레지스터를 읽어와 전압/전류 Fault 상태를 확인한다.
- **MCU ↔ EEPROM (SPI 통신):**
  - MOSI, MISO, SCK, CS 라인을 통해 연결된다.
  - MCU는 SPI 마스터로서 PMIC 또는 ADC에서 감지된 결함 정보를 DTC로 변환하여 EEPROM에 저장하거나 읽어온다.
- **MCU ↔ CAN 트랜시버 ↔ 외부진단기 (CAN 통신):**
  - MCU의 CAN 컨트롤러 핀인 CAN_TX와 CAN_RX를 통해 트랜시버와 연결된다.
  - CAN 트랜시버는 MCU의 논리 신호를 물리적인 CAN 버스 신호(CAN_H, CAN_L)로 변환하여 외부 진단기 등 다른 CAN 노드와 통신을 중계한다.

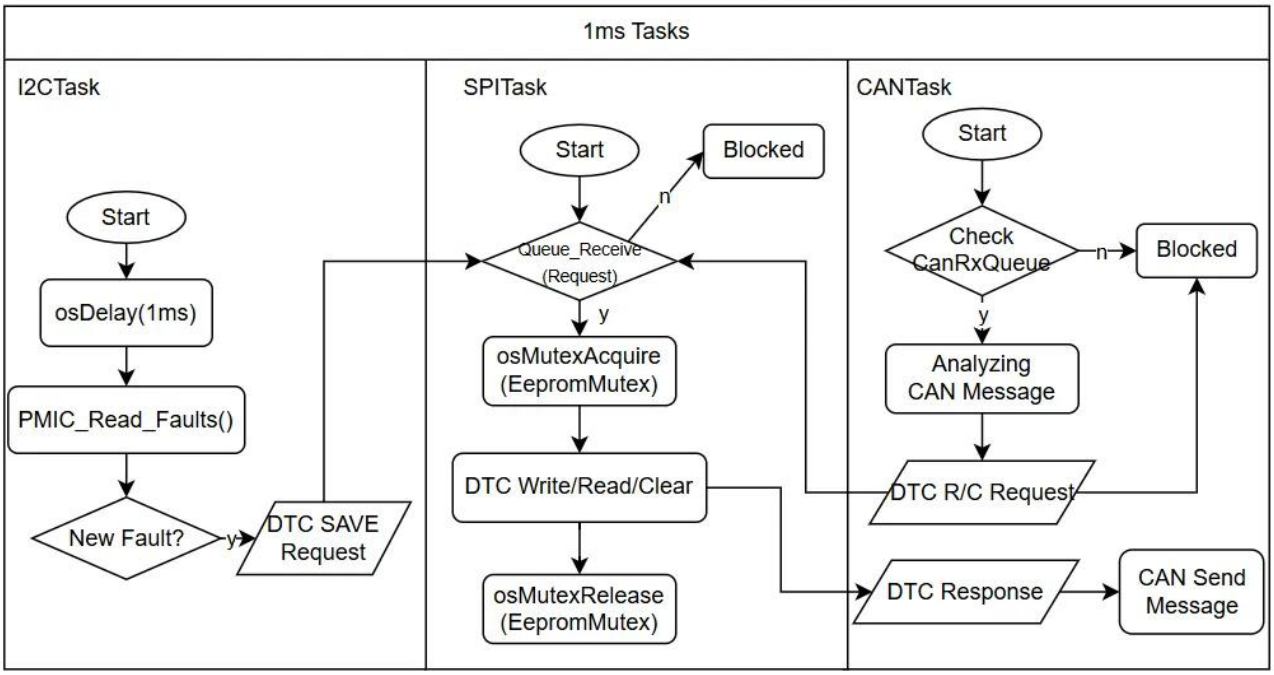
2.2. 소프트웨어 아키텍처

본 프로젝트의 소프트웨어는 Application Layer과 Driver Layer의 2계층 구조로 설계하여 코드의 모듈성과 재사용성을 높인다.

1. Driver Layer
   - 각 IC를 직접 제어하는 역할
   - 칩 하나당 .c 와 .h 파일 한 쌍으로 구성
   - HAL 함수들을 감싸서 명확한 기능의 함수 제공
   - I2C 드라이버: pmic_mp5475gu.c
   - SPI 드라이버: eeprom_25lc256.c
   - CAN 드라이버: yj_can.c
   - ADC 드라이버: yj_adc.c
   - UART 드라이버: yj_uart.c
2. Application Layer
   - 시스템의 전체적인 동작 시나리오 구현
   - tasks.c에 I2CTask, CANTask와 같은 RTOS 태스크들이 구성됨

2.2.1. RTOS 태스크 설계

- 1ms 주기 태스크 (I2C, SPI, CAN)의 역할 및 동작 흐름



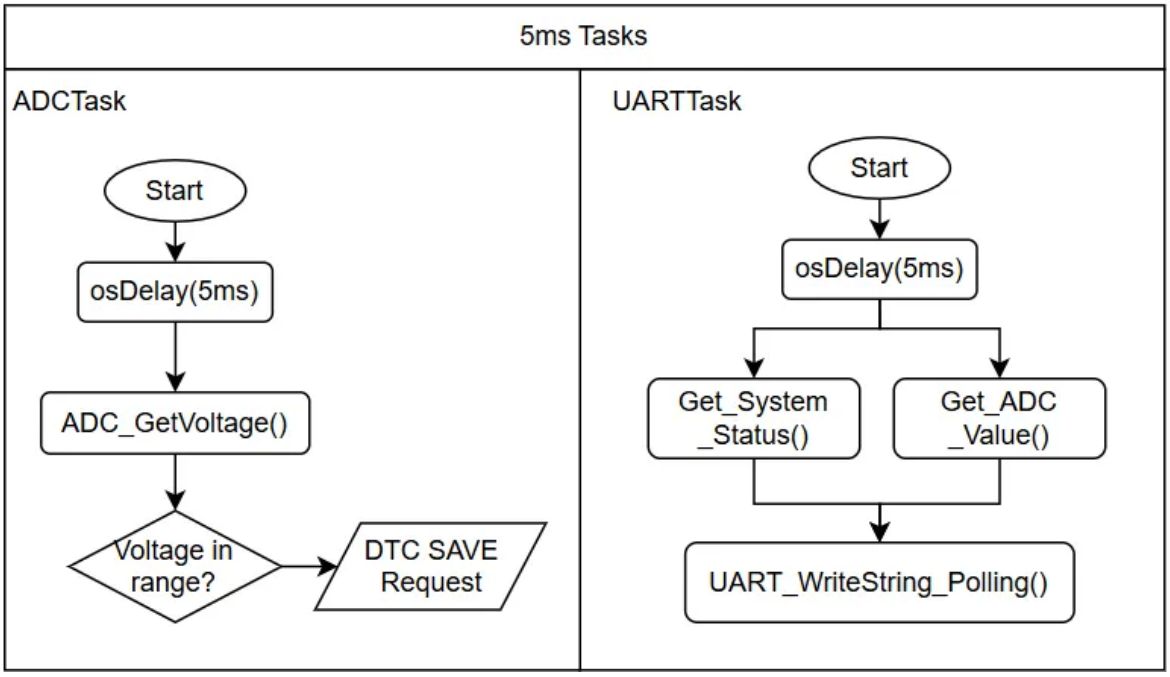
- I2CTask
  - **역할:** 주기적으로 PMIC(MP5475GU)의 상태를 감시한다.
  - **주요 동작:** 1ms마다 깨어나 드라이버 계층의 PMIC_Read_Faults() 함수를 호출하여 PMIC 내부 Fault 레지스터를 확인한다.
  - **결과 처리:** 이전에 없던 새로운 Fault가 감지되면, 해당 정보를 DTC_RequestQueue에 담아 전송하여 SPITask에게 "새로운 고장 발생"을 알린다.
- SPITask
  - **역할:** EEPROM(25LC256)에 대해 DTC의 모든 읽기/쓰기 동작 전담한다.
  - **주요 동작:** 평소에는 Blocked 상태로 대기, DTC_RequestQueue에 메시지가 도착하면 깨어난다.
  - **동작 1 (DTC 저장):** I2CTask로부터 새로운 Fault 이벤트를 수신하면, 뮤텍스를 점유하여 EEPROM 접근 권한을 확보한 뒤, EEPROM_Write_DTC()를 호출하여 DTC를 저장한다.
  - **동작 2 (DTC 처리):** CANTask로부터 DTC 조회/삭제 요청을 받으면, 뮤텍스 점유 후 EEPROM_Read_DTCs() 또는 EEPROM_Write_DTCs() 함수를 호출하여 요청을 처리하고, 그 결과를 DTC_ResponseQueue를 통해 CANTask에 회신한다.

- CANTask
  - **역할:** 외부 진단기와의 모든 CAN 통신을 처리하는 유일한 창구이다.
  - **주요 동작:** CAN 메시지가 수신되면 CAN RX ISR이 CanRxQueue로 데이터를 넣어주고, CANTask는 이 큐에 데이터가 들어오기를 기다린다 → **CANTask에서 데이터를 넣지 않는다.**
  - **결과 처리:** 수신된 CAN 메시지를 분석하여 진단기의 요청(DTC 조회/삭제)이 무엇인지 파악한다.

    해당 요청을 DTC_RequestQueue를 통해 SPITask에 전달하고, DTC_ResponseQueue로부터 처리 결과가 올 때까지 기다린다.

    결과 수신 후, 진단기에 보낼 응답 CAN 메시지를 만들어 전송한다.

- **5ms 주기 태스크 (UART, ADC)의 역할 및 동작 흐름**



- ADCTask
  - **역할:** PMIC의 특정 아날로그 채널 전압을 주기적으로 측정하여, 시스템의 전압 안정성을 감시한다. I2CTask가 디지털 Fault 레지스터를 확인하는 것과 달리, ADCTask는 실제 아날로그 전압 레벨을 직접 확인한다.
  - **주요 동작:** 5ms마다 깨어나 드라이버 계층의 ADC_GetVoltage() 함수를 호출하여 PMIC 채널의 전압 값을 읽어온다.

- **결과 처리:** 측정된 전압이 사전에 정의된 정상 범위를 벗어날 경우(예: 과전압 또는 저전압), 이에 해당하는 DTC를 생성한다.

  생성된 DTC 정보는 1ms 태스크 그룹에서 사용하는 것과 동일한 DTC_RequestQueue로 전송하여 SPITask가 EEPROM에 저장하도록 요청한다.

- UARTTask
  - **역할:** 시스템의 현재 동작 상태나 주요 데이터(ADC 측정값 등)를 PC 시리얼 터미널과 같은 외부 장치로 전송하여, 개발 및 디버깅 과정을 돕는다.
  - **주요 동작:** 5ms마다 깨어나, 다른 태스크들이 갱신한 시스템의 전반적인 상태와 가장 최근에 측정된 전압 값 등 시스템의 주요 정보를 수집한다.
  - **결과 처리:** 수집된 정보들을 사람이 읽을 수 있는 문자열 포맷팅한다. 드라이버 계층의 UART_WriteString_Polling() 함수를 호출하여 해당 문자열을 UART로 송출한다.

2.2.2. 통신 드라이버 설계

- **I2C & SPI DMA 방식 구현 방안**

  DMA(Direct Memory Access)를 사용하는 가장 큰 이유는 CPU를 통신 데이터 전송 작업으로부터 해방시켜 다른 중요한 일을 할 수 있도록 하는 것이다.

  RTOS 환경에서는 이를 극대화하여, DMA가 일하는 동안 현재 태스크를 Blocked 상태로 만들고 다른 태스크에게 CPU를 양보하는 방식으로 시스템 효율을 높인다.

  설계 목표:

  - HAL_I2C_Master_Transmit_DMA(), HAL_SPI_TransmitReceive_DMA()와 같은 **Non-Blocking DMA 함수**를 사용하여 통신을 시작한다.
  - RTOS의 **Semaphore**를 사용하여 DMA 전송 완료 IRQ를 받을 때까지 현재 태스크를 **Blocked 상태**로 전환한다.

- **CAN Interrupt 방식 구현 방안**

  DMA 방식과 마찬가지로, 핵심은 **ISR의 역할을 최소화**하고, RTOS의 기능을 활용하여 안정적이고 효율적으로 데이터를 처리하는 것이다.

  CAN 통신에서는 '완료' 신호만 보내는 세마포어보다, Event와 수신된 데이터 자체를 담을 수 있는 **Message Queue**를 사용하는 것이 훨씬 빠르고 안전하다.

  설계 목표:

  - CAN 메시지 수신 인터럽트가 발생했을 때 실행되는 **ISR의 코드를 최대한 짧고**

빠르게 유지하여 시스템 전체의 반응성을 저해하지 않는다.

- 수신된 CAN 메시지 데이터는 Message Queue를 통해 ISR 컨텍스트에서 CANTask 컨텍스트로 안전하게 전달한다.
- CANTask는 메시지 큐에 새로운 데이터가 수신될 때까지 **Blocked 상태**로 대기하여 CPU 자원 낭비를 방지한다.

- **UART Polling 방식 구현 방안**

  다른 통신 방식들과 달리 UART는 디버깅 메시지 출력이라는 비교적 중요도가 낮은 역할을 담당하므로, 가장 구현이 간단한 Polling 방식을 사용하는 것이 합리적이다.

  설계 목표:

  - 디버깅 메시지 출력과 같이 실시간성이 중요하지 않은 기능에 대해 가장 간단한 **Blocking 함수**를 사용하여 구현 복잡도를 낮춘다.
  - UARTTask는 낮은 우선순위의 주기적인 태스크이므로, 데이터 전송 중에 잠시 Blocking되더라도 시스템의 핵심 기능에 영향을 주지 않는다.

3. 핵심 기능 구현

3.1. PMIC (MP5475GU) 상태 감지

3.1.1. 전압/전류 Fault 감지 관련 레지스터 분석

1. 입력 전압 (V_{IN}) 보호

- **저전압 차단 (UVLO):** 입력 전압이 너무 낮으면 PMIC의 동작을 막아 시스템의 불안정한 시작과 오작동을 방지한다.

  UVLO(System 범주 0x43[7])를 제어해 하강/상승 임계값을 설정한다.

  하강/상승 임계값 사이는 hysteresis 구간으로, **전압이 상승 중일 때** 상승 임계값 도달까지 회로는 **비활성화** 상태를 유지하며, **전압이 하강 중일 때** 하강 임계값 도달까지 회로는 **활성화** 상태를 유지한다.

  UVLO 제어 비트는 MTP(Multiple-Time Programmable) 메모리를 통해 설정이 가능하므로 일반적인 I2C 통신으로 비트를 설정할 수 없다.

43 (WR)	UVLO	MTP_ PROGRAM	SHUTDOWN_ DELAY_EN	OFF_TIME	SFRST_NUM

7	WR	UVLO	1'b0	V_{CC} < 2V, V_{DRV} < 2V	Sets the V_{IN} UVLO threshold. 0: 3.7V rising UVLO threshold, and 3.48V falling UVLO threshold (default) 1: 2.85V rising UVLO threshold, and 2.6V falling UVLO threshold Note that the UVLO control bit only takes effect in the MTP.

- **과전압 보호 (OVP)**: 입력 전압이 16.6V를 초과하면, PMIC는 과전압 보호 모드로 진입한다.

  VBULK_OV (System 범주 0x09[3])를 통해 OVP 진입 전에 부하를 미리 줄이거나 사용자에게 알림을 보내는 등의 조치를 취할 수 있다.

  입력 전압이 14.5V를 초과하는 경우 플래그가 설정된다.

  14.2V ~ 14.5V 사이일 경우, 상태는 이전 전압 변화 방향에 따라 유지된다.

09 (WC)	RESERVED	1.8VLDO_ FAULT	1.1VLDO_ FAULT	VR_ FAULT	VBULK_ OV	VDRV_OV	PMIC_ HIGH_ TEMP_ WARNING	PMIC_ HIGH_ TEMP_ SHUTDOWN

3	WC	VBULK_OV	1'b0	Indicates the V_{INx} OV status. 0: V_{INx} is below 14.2V 1: V_{INx} exceeds 14.5V

2. 출력 전압(V_{OUT}) 보호

- **저전압 보호 (UVP)**: 출력 전압이 설정값의 약 20% 이하로 떨어지면 BUCKx_UV (System 범주 0x07[7:4])가 1로 설정된다.

  이 자체만으로 큰 조치를 취하지 않으며, 주로 **과전류(OCP)와 동시에 발생할 때** 강력한 보호 조치인 '히컵 모드'가 활성화된다.

  정상 출력 전압은 각 BUCK 컨버터 레지스터의 VOUT_SELECTx, V_REFx_HIGH, V_REFx_LOW 필드에서 설정할 수 있다.

07 (RC)	BUCKA_ UV	BUCKB_ UV	BUCKC_ UV	BUCKD_ UV	BUCKA_ OV	BUCKB_ OV	BUCKC_ OV	BUCKD_ OV
07 (RC)	BUCKA_ UV	BUCKB_ UV	BUCKC_ UV	BUCKD_ UV	BUCKA_ OV	BUCKB_ OV	BUCKC_ OV	BUCKD_ OV

- **과전압 보호 (OVP)**: 출력 전압이 설정값의 122%를 초과하면, Low-side MOSFET을 강제로 켜서 과도한 전압을 그라운드로 방전시킨다.
  - VOUT_OVP_ENx (BUCKx 범주 0x12, 0x1A, 0x22, 0x2A [3]): 각 BUCK 컨버터의 OVP 기능을 켠다.

12 (WR)	VOUT_ LIMIT_ ENA	MODEA	CURRENT_LIMITA	VOUT_ OVP_ENA	PHASE_DELAY_ SELECTA	VOUT_ DIS_ENA

3	WR	VOUT_OVP_ENx	1	V_{CC} < 2V, V_{DRV} < 2V	Enables each buck's output OVP. 0: Disabled 1: Enabled

  - BUCKx_OV (System 범주 0x07[3:0]): 각 BUCK 컨버터의 OV 상태를 나타낸다.

07 (RC)	BUCKA_ UV	BUCKB_ UV	BUCKC_ UV	BUCKD_ UV	BUCKA_ OV	BUCKB_ OV	BUCKC_ OV	BUCKD_ OV

3:0	RC	BUCKx_OV	4'b0000	Indicates buck x's (where x = A, B, C, or D) output OV status. 0: No OV condition on buck x 1: OV condition on buck x (above 20% of nominal output window)

3. 과전류 보호 (OCP):

- **Valley Current Limit:** 스위칭 주기마다 인덕터 전류의 최저점(valley)을 감시한다. 만약 이 전류가 설정된 임계값 이하로 떨어지지 않으면, 다음 스위칭 사이클을 건너뛰는 방식으로 전류를 제한한다.
- **Hiccup Mode**: 지속적인 과전류 또는 단락 상태에서 PMIC와 시스템을 보호한다. 앞서 언급했듯이 OCP와 UVP가 동시에 감지될 때 활성화된다. PMIC는 출력을 완전히 차단했다가, 정해진 시간 후 다시 소프트 스타트를 시도한다. 오류가 지속되면 이 과정이 hiccup되어, 평균 전력 소모를 줄여 열로 인한 파손을 방지한다.

- CURRENT_LIMITx (BUCKx 범주 0x12, 0x1A, 0x22, 0x2A [5:4]): 각 BUCK 컨버터의 Valley 전류 제한 값을 설정한다.

3:0	RC	BUCKx_OV	4'b0000	Indicates buck x's (where x = A, B, C, or D) output OV status. 0: No OV condition on buck x 1: OV condition on buck x (above 20% of nominal output window)

5:4	WR	CURRENT_LIMITx	11	V_{CC} < 2V, V_{DRV} < 2V	Sets the current limit of each buck, with 15% accuracy. 00: 4A valley current limit for 3A output current (I_{OUT}) applications 01: 5A valley current limit for 4A I_{OUT} applications 10: 6A valley current limit for 5A I_{OUT} applications 11: 7A valley current limit for 6A I_{OUT} applications

- BUCKx_OC, BUCKx_OC_WARNING (System 범주 0x08[7:4][3:0]): 각각 OC Fault가 발생하여 OCP 기능이 작동되었음을 알리고 **Fault 발생 전에 위험을 미리 알려주는** 신호를 나타낸다.

08 (RC)	BUCKA_ OC	BUCKB_ OC	BUCKC_ OC	BUCKD_ OC	BUCKA_ OC_ WARNING	BUCKB_ OC_ WARNING	BUCKC_ OC_ WARNING	BUCKD_ OC_ WARNING

7:4	RC	BUCKx_OC	4'b0000	Indicates buck x's (where x = A, B, C, or D) OC status. 0: No buck x OC condition 1: Buck x over-current protection (OCP) has occurred

3:0	RC	BUCKx_OC_ WARNING	4'b0000	Indicates buck x's (where x = A, B, C, or D) high current consumption warning status (the output current exceeds 85% of the threshold). 0: No high current consumption warning 1: High current consumption warning

3.1.2. 데이터 구조 정의

pmic_mp5475gu.h 파일에서는 MP5475 데이터시트를 기반으로 I2C Slave Address를 매크로로 정의하고, 시스템 상태 및 Fault 레지스터를 enum 형태로 설계하였다.

또한, **Union과 비트필드**를 활용하여 레지스터 데이터를 효율적으로 파싱할 수 있는 구조를 구현하였다.

3.1.3. I2C 드라이버 및 I2CTask 구현

1. **I2C 드라이버** (pmic_mp5475gu.c/.h)
    - **초기화:** main.c에서 PMIC_Init() 함수를 호출하여 DMA 통신 완료 신호를 받을 i2c_dma_semaphore를 생성한다.
    - **읽기 요청:** I2CTask는 PMIC_Read_Faults() 함수를 호출한다. 이 함수는 HAL_I2C_Mem_Read_DMA()를 통해 Non-Blocking 방식으로 PMIC 레지스터 읽기를 시작한다.
    - **Blocked:** PMIC_Read_Faults() 함수는 osSemaphoreAcquire()를 호출하여 I2CTask를 Blocked 상태로 전환시키고 DMA 완료 신호를 기다린다.
    - **완료 신호:** DMA 전송이 완료되면 stm32f4xx_it.c에 있는 HAL_I2C_MemRxCpltCallback() 콜백 함수가 실행되어, osSemaphoreRelease()를 통해 Blocked 된 I2CTask를 깨운다.
2. **I2CTask** (tasks.c)
    - **주기적 실행:** StartI2CTask() 함수는 osDelay(1)을 통해 1ms 주기로 실행된다.
    - **데이터 수집:** 매 주기마다 pmic_mp5475gu.c에 구현된 PMIC_Read_Faults()를 호출하여 PMIC의 Fault 레지스터 값을 읽어온다.
    - **상태 비교:** static 변수에 저장된 이전 Fault 상태와 현재 읽어온 Fault 상태를 비교한다.
    - **이벤트 생성:** 비교 결과, 이전에 없던 **새로운 Fault**가 감지되면(0 → 1로 변경), dtc_manager.h에 정의된 DTC 코드를 DTC_Message_t에 담아 DTC_RequestQueue로 전송한다.
    - **에러 처리:** PMIC_Read_Faults() 함수가 통신 실패(HAL_TIMEOUT 등)를 반환하면, 통신 두절에 해당하는 시스템 DTC를 전송한다.

3.2. DTC 관리 (EEPROM 연동)

3.2.1. DTC 정의 (dtc_manager.h)

•ECU는 2바이트(uint16_t) 16진수 값으로 DTC를 저장 및 전송한다.

•진단기는 이 숫자 값을 받아 "C0035"와 같은 5자리 코드로 변환하여 표시한다.

•아래 목록은 https://www.obd-codes.com/trouble_codes/ 와 같은 공개 자료를 참조하여

•PMIC의 각 Buck 출력이 브레이크 관련 부품에 전원을 공급한다고 가정하고,

•UV, OC 등의 상태를 가장 유사한 의미의 OBD-II 표준 DTC와 매핑하였다.

전원 블록	감시 항목	DTC Code	설명
BUCK A (좌측 전륜 휠 스피드 센서 전원)	UV	C1221	센서 입력 신호 없음
	OC	C1232	센서 회로 단선/합선
BUCK B (우측 전륜 휠 스피드 센서 전원)	UV	C1222	센서 입력 신호 없음
	OC	C1233	센서 회로 단선/합선
BUCK C (ABS 펌프 모터 전원)	UV	C1242	펌프 모터 회로 단선
	OC	C1217	펌프 모터 접지 합선
BUCK D (솔레노이드 밸브 릴레이 전원)	UV	C0577	솔레노이드 회로 전압 낮음
	OC	C0121	밸브 릴레이 회로 오작동
ADC (ECU 메인 입력 전원 감시)	임계값 이하	C1236	시스템 공급 전압 낮음
	임계값 초과	C1237	시스템 공급 전압 높음
기타 시스템 오류	PMIC 과열 등	U0121	ABS 모듈과 통신 두절

3.2.2. SPI 드라이버 및 SPITask 구현

1. SPI 드라이버 (eeprom_25lc256.c / .h)
   - **초기화**: main.c에서 EEPROM_Init() 함수를 호출하여 DMA 통신 완료 신호를 기다릴 spi_dma_semaphore를 생성한다.
   - EEPROM_Write_DTC(): SPITask로부터 DTC 저장 요청을 받으면, HAL_SPI_Transmit_DMA() 함수를 호출하여 DMA를 이용한 비동기 쓰기 작업을 시작한다.
   - EEPROM_Read_DTCs(): SPITask로부터 DTC 읽기 요청을 받으면, HAL_SPI_TransmitReceive_DMA() 함수를 호출하여 DMA를 이용한 비동기 읽기 작업을 시작한다.
   - **비동기 대기**: 위 두 함수는 DMA 작업을 시작시킨 직후, osSemaphoreAcquire(spi_dma_semaphore, ...)를 호출하여 SPITask를 즉시 **Blocked 상태**로 만든다.
   - **완료 처리**: DMA 전송이 완료되면 stm32f4xx_it.c에 있는 HAL_I2C_MemRxCpltCallback() 콜백 함수가 실행되어, osSemaphoreRelease()를 통해 Blocked 된 I2CTask를 깨운다.
2. SPITask (tasks.c)
   - **대기 상태**: StartSPITask 함수는 osMessageQueueGet(DTC_RequestQueueHandle, ...)을 호출하며, I2CTask나 CANTask로부터 요청이 올 때까지 Blocked **상태**로 대기한다.
   - **요청 처리**: 메시지를 받으면 깨어나, osMutexAcquire(EepromMutexHandle, ...)를 통해 EEPROM에 대한 점유권을 얻는다. 그 후 메시지 타입에 따라 다음 작업을 수행한다.
     - SAVE_DTC_REQUEST: EEPROM_Write_DTC()를 호출하여 새로운 DTC를 저장한다.
     - READ_ALL_DTCS_REQUEST: EEPROM_Read_DTCs()를 호출하여 저장된 모든 DTC를 읽고, 그 결과를 DTC_ResponseQueueHandle을 통해 CANTask에게 응답한다.
     - CLEAR_ALL_DTCS_REQUEST: EEPROM_Write_DTC()를 호출하여 DTC 저장 영역을 모두 삭제한다.
   - **정리**: 모든 작업이 끝나면 osMutexRelease(EepromMutexHandle)를 호출하여 점유권을 반납하고, 다시 다음 요청을 기다리며 Blocked 상태로 돌아간다.

3.3. CAN 진단 통신

3.3.1. CAN 메시지 프로토콜 정의

본 프로젝트의 진단 통신은 자동차 표준인 UDS 프로토콜을 기반으로 설계되었다. 이를 통해 DTC를 읽고 삭제하기 위한 CAN 메시지 ID와 데이터 형식을 can_uds_protocol.h에서 정의하였다.

- **기반 표준**: UDS (Unified Diagnostic Services, **ISO 14229**)
- **요청 ID**: 진단기는 모든 ECU가 수신하는 표준 기능 주소 0x7DF로 요청 메시지를 전송한다.
- **응답 ID**: ECU는 자신의 고유한 물리 주소인 0x7E8로 응답 메시지를 전송한다.
- **제약**: 현재 구현된 DTC 읽기 응답 로직은 **단일 CAN 메시지의 8바이트 데이터 크기 제약**으로 인해, 한 번의 응답에 **최대 2개의 DTC** 정보만 포함하여 전송한다. 저장된 모든 DTC를 전송하기 위해서는 8바이트 이상의 데이터를 여러 메시지로 나누어 보내는 전송 프로토콜(Transport Protocol, ISO 15765-2)의 추가 구현이 필요하다.

3.3.2. CAN 응답 로직 / CANTask

1. **CAN 드라이버** (yj_can.c)
   - CAN_Init(): CAN 통신에 필요한 하드웨어를 시작시키고, 메시지가 도착하면 알려달라고 **수신 인터럽트를 활성화**한다.
   - CAN_SendMessage(): CANTask로부터 전달받은 ID와 데이터를 실제 CAN 버스에 전송하는 역할을 한다.
2. **CAN 수신 인터럽트** (stm32f4xx_it.c)
   - 외부 진단기로부터 CAN 메시지가 도착하면, ISR인 HAL_CAN_RxFifo0MsgPendingCallback() 함수가 실행되어, 수신된 메시지를 CanQueue에 빠르게 넣고 즉시 종료된다.
3. **CANTask** (tasks.c)
   - StartCANTask()
     - 태스크가 시작되면 먼저 수신 인터럽트를 활성화한다. ISR이 CanQueue에 메시지를 넣어줄 때까지 Blocked 된다.
     - 메시지가 도착하면 메시지를 보낸 주소(StdId)가 우리가 약속한 진단기 주소(CAN_ID_DIAG_REQUEST)가 맞는지 확인한다.
     - 유효한 요청인 경우, Process_CAN_Response로 메시지를 전달한다.

- Process_CAN_Response

1. **DTC 읽기 요청(SID_READ_DTC_INFO) 처리:**
    a. SPITask에게 "EEPROM에서 모든 DTC를 읽어달라"고 요청하고, 응답을 받을 때까지 잠시 대기한다.
    b. SPITask로부터 받은 DTC 목록을 UDS 프로토콜 형식에 맞는 CAN 메시지로 CAN_SendMessage()를 통해 진단기로 전송한다.
2. **DTC 삭제 요청(SID_CLEAR_DIAG_INFO) 처리:**
    a. SPITask에게 "EEPROM의 모든 DTC를 삭제해달라"고 요청한다.
    b. 삭제 작업을 기다리지 않고, "요청이 정상적으로 접수되었다"는 의미의 긍정 응답 메시지를 즉시 만들어 진단기로 전송한다.

3.4. ADC 전압 모니터링

3.4.1. 전압 레벨 스펙 정의

ADCTask는 PMIC의 특정 전원 출력 채널(BUCK D)을 12-bit 해상도의 ADC로 주기적으로 측정하여 시스템의 전압 안정성을 감시한다.

1. **기준 전압**
    - **BUCK D**의 기본 출력 전압은 **1.8V**이다.
2. **정상 동작 범위**
    - MP5475GU 데이터시트에 따르면 기준 전압의 20% 범위를 정상 동작 범위로 정의한다.
    - **정상 범위: 1.44V ~ 2.16V**
    - ADC 측정 전압이 정상 범위를 벗어날 경우, 시스템 전원에 이상이 발생한 것으로 판단하고 DTC를 생성한다.
3. **DTC 매핑**
    - **저전압 감지 시:** DTC_C1236_VOLTAGE_LOW (Low System Supply Voltage) 생성
    - **과전압 감지 시:** DTC_C1237_VOLTAGE_HIGH (High System Supply Voltage) 생성

이러한 기준을 통해 ADCTask는 PMIC의 디지털 Fault 레지스터만으로는 감지하기 어려운 미세한 전압 변동이나 이상 상태를 감지하여 시스템의 안정성을 더욱 높일 수 있다.

3.4.2. ADC 측정 및 DTC 생성 로직 구현

1. **ADC 드라이버** (yj_adc.c / .h)

   ADC_GetVoltage()

   1. HAL_ADC_Start(): ADC 변환을 시작한다.
   2. HAL_ADC_PollForConversion(): 변환이 완료될 때까지 잠시 대기한다(Blocking).
   3. HAL_ADC_GetValue(): 변환된 12비트 디지털 값(0~4095)을 읽어온다.
   4. **값 변환**: 읽어온 디지털 값을 MCU의 ADC 기준 전압(VREF, 3.3V)을 이용하여 실제 전압(float)으로 계산한 뒤, 이 값을 반한다.

2. **ADCTask** (tasks.c)

   StartADCTask()

   1. osDelay(5)를 통해 **5ms**마다 한 번씩 동작한다.
   2. 매 주기마다 yj_adc.c의 ADC_GetVoltage() 함수를 호출하여 현재 BUCK D의 전압 값을 얻어온다.
   3. static 변수(g_is_voltage_low, g_is_voltage_high)를 이용해 이전 루프에서의 전압 상태를 기억한다.
   4. 현재 전압이 임계값을 벗어나고 **이전 상태는 정상이었을 경우에만**, 즉 **고장이 처음 발생한 순간에만** DTC_RequestMessage_t를 SPITask가 기다리는 DTC_RequestQueueHandle로 전송한다.
   5. 전압이 다시 정상 범위로 돌아오면, static 상태 변수를 리셋하여 다음 고장 발생을 감지할 수 있도록 준비한다.

3.5. UART 시스템 상태 모니터링

1. **UART 드라이버** (yj_uart.c / .h)

   UART_Printf()

   1. UARTTask로부터 출력할 문자열 형식(예: "Voltage: %.2fV")과 데이터(예: 1.81)를 전달받는다.
   2. vsnprintf 함수를 이용해 이 둘을 조합하여 "Voltage: 1.81V"와 같은 최종 출력 문자열을 만든다.
   3. HAL_UART_Transmit 함수를 호출하여 완성된 문자열을 **폴링(Blocking) 방식**으로 전송한다. 전송이 완료될 때까지 CPU가 잠시 기다린다.

2. **UART 태스크** (tasks.c)

   StartUARTTask()

   1. osDelay(5)를 통해 **5ms에 한 번씩** 동작한다.
   2. ADCTask가 측정한 최신 BUCK D 전압 값(g_latest_adc_voltage)과 상태 (g_is_voltage_low, g_is_voltage_high)를 **공유 변수**를 통해 읽어온다.
   3. **상황별 상태 출력**:
      - 만약 전압이 낮으면, "Low System Supply Voltage" 메시지를,
      - 만약 전압이 높으면, “HIGH System Supply Voltage" 메시지를,
      - 아무 이상이 없으면, "System Status: OK" 라는 정상 상태 메시지를 만든다.
   4. 위에서 만든 최종 메시지를 UART_Printf() 함수에 전달하여 PC 터미널로 출력을 요청한다.

4. 결론

4.1. 프로젝트 결과 요약

본 프로젝트는 **RTOS 기반의 차량용 진단 제어기 펌웨어**를 성공적으로 설계하고, 그 핵심 로직을 C언어로 구현하는 것을 목표로 하였다. 최종적으로 다음과 같은 개발 목표를 달성하며 프로젝트를 완수하였다.

- **계층적 소프트웨어 아키텍처 설계:** 애플리케이션 계층(tasks.c)과 하드웨어 제어를 담당하는 드라이버 계층을 명확히 분리하여, 코드의 모듈성, 재사용성, 유지보수성을 위한 소프트웨어 구조를 완성하였다.
- **RTOS 기반 멀티태스킹 구현:** I2CTask, SPITask, CANTask 등 기능별로 분리된 5개의 태스크를 구현하고, 메시지 큐, 세마포어, 뮤텍스와 같은 RTOS 객체를 활용하여 이들이 **안전하고 효율적으로 협력**하도록 설계하였다. 이를 통해 실시간 시스템의 핵심인 동시성 문제를 해결하였다.
- **실시간 Fault 감지 및 DTC 관리 시스템 구현:** PMIC의 UV/OC Fault와 ADC의 전압 이상을 실시간으로 감지하고, **이전에 없던 새로운 Fault**만 선별하여 표준 DTC로 변환하는 로직을 구현하였다. 생성된 DTC는 SPITask를 통해 EEPROM에 안정적으로 저장 및 관리된다.
- **표준 진단 프로토콜 구현:** 자동차 진단 표준인 **UDS(ISO 14229)** 프로토콜을 기반으로 외

부 진단기와의 통신 규약을 정의하고, CANTask를 통해 DTC 읽기 및 삭제 요청을 성공적으로 처리하는 로직을 구현하였다.

- **효율적인 드라이버 구현:** I2C/SPI 통신에는 **DMA와 세마포어**를, CAN 수신에는 **인터럽트와 메시지 큐**를 적용하여, 통신이 이루어지는 동안 CPU가 다른 작업을 처리할 수 있는 비동기/Non-Blocking 방식의 고효율 드라이버를 설계하였다.

4.2. 개선 및 보완 사항

본 프로젝트는 성공적으로 핵심 기능을 구현하였으나, 실제 제품으로 발전시키기 위해 다음과 같은 개선 및 보완 사항을 제안할 수 있다.

- **CAN 전송 프로토콜(ISO 15765-2) 구현:** 현재 DTC 읽기 응답은 단일 CAN 프레임(8바이트)의 제약으로 최대 2개의 DTC만 전송할 수 있다. 8바이트가 넘는 긴 데이터를 여러 프레임으로 나누어 보내는 **전송 프로토콜(Transport Protocol)**을 구현하면, 저장된 모든 DTC 목록(16개 이상)을 한 번에 전송하는 완전한 진단 기능을 구현할 수 있다.
- **동적 Fault 복구 로직 추가:** 현재는 Fault 발생 시 DTC를 기록하는 데 중점을 두고 있다. 향후 특정 전원 채널(예: Buck A)에서 OC Fault가 발생했을 때, 해당 채널을 비활성화했다가 일정 시간 후 **자동으로 다시 활성화하여 시스템을 복구**하려는 시도와 같은 능동적인 에러 처리 로직을 추가할 수 있다.
- **실제 하드웨어 기반의 V-모델 검증 수행:**

현재까지의 개발은 **V-모델**의 좌측(설계 및 구현)에 해당한다. 프로젝트의 완성도를 높이기 위해 NUCLEO 보드, PMIC, EEPROM, CAN 트랜시버 등의 실제 하드웨어를 구성하고, 이를 바탕으로 V-모델 우측의 **통합 및 시스템 테스트**를 수행할 수 있다.

이는 소프트웨어 로직이 실제 하드웨어의 전기적 특성 및 타이밍과 맞물려 안정적으로 동작하는지를 검증하는 필수적인 단계로, 펌웨어의 신뢰성을 최종적으로 확보하는 과정이다.

5. 부록

5.1. 소스코드 및 문서 저장 (GitHub 링크)

전체 소스코드와 결과 보고서, 참고 자료(사용 IC 데이터시트)들을 저장하였다.

https://github.com/YJCHOI15/ecu-dtc-manager