

6 조 상태 기반 차량 제어 시스템 결과보고서

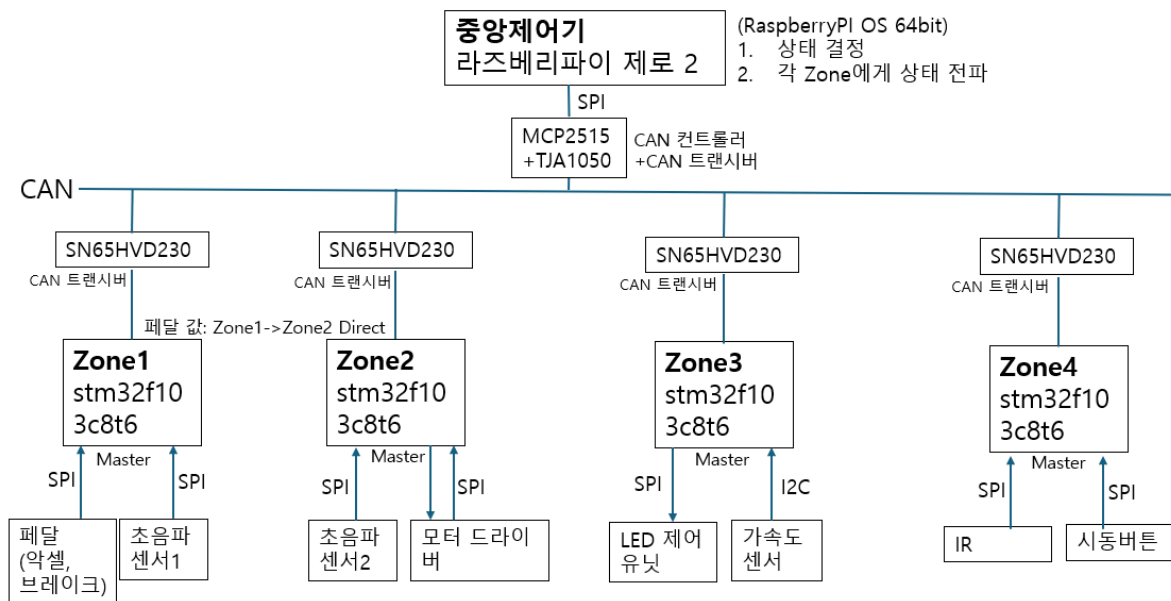
목차

1. 개요
2. 구성
3. 통신 구성
4. STATE 설계 (목표 기능)
5. CAN 상세 설계
6. 상세 설계
7. 결과 분석 - 별도 첨부
8. 고찰
9. 참고 문헌

1. 개요

중앙 제어기와 4 개의 Zone 제어기, 총 5 개의 Node 를 갖는 CAN 버스를 통해 상태를 전파하는 시스템이다. 개발 환경은 SSH, STM32CubeIDE 이다.

2. 구성



이름	보드 종류	기능
중앙 제어기	Raspberry Pi Zero 2 W	- 상태 결정
Zone1	STM32F103C8T6 호환보드	- 페달 전송 - 초음파 전송
Zone2	STM32F103C8T6 호환보드	- 초음파 전송 - 모터 드라이버 전송
Zone3	STM32F103C8T6 호환보드	- LED 전송 - 가속도 전송
Zone4	STM32F103C8T6 호환보드	- IR 전송 - 버튼 전송

보드 결정 주요 요인은 가격으로, 수업에서 배운 STM32 제품을 쓰되 여러 보드를 써야 하기에 가능한 저렴하지만 적당한 스펙과 TIM 기능이나 통신 인터페이스가 적절히 탑재되어 있는 것을 별도로 선택해 사용했다.

Zone	통신	산하 모듈	기능
Zone1	SPI1	페달 (STM32F103C8T6)	페달로서 2 개의 가변 저항 측정 및 데이터 전송
	SPI2	초음파 (STM32F103C8T6)	HC-SR04 초음파 센서 구동 및 데이터 전송
Zone2	SPI1	초음파 (STM32F103C8T6)	HC-SR04 초음파 센서 구동 및 데이터 전송
	SPI2	모터 드라이버 (STM32F103C8T6)	TB6612FNG 모터 드라이버 구동
Zone3	SPI1	LED 제어 유닛 (STM32F103C8T6)	상태에 따른 LED 제어
	I2C1	가속도	ADXL345 가속도 센서 측정 및 전송
Zone4	SPI1	IR	탐승 감지
	SPI2	BTN	시동 버튼 감지

3. 통신 구성

3.1 I2C

400kHz Fast Mode 로 I2C 를 사용했다. 가속도 센서가 I2C 만을 지원했기에 SPI 대신 I2C 통신을 구축했다. 풀업 저항은 0->1 상승 시간이 300ns 이내, 최소 저항값 표준을 지키기 위해 안전하게 4.7kohm 을 사용했다.

3.2 SPI

STM32F103C8T6 는 2 개의 SPI 인터페이스를 지원하여 SPI1, SPI2 를 나누어 동시 접근이 없도록 했다. HCLK = 64MHz, APB2 = 64MHz, APB1 = 36MHz 기준 SPI1 은 APB2 를 사용하기에 Prescaler = 64, SPH2 는 APB1 클럭 버스를 공유하기에 Prescaler = 32 로 설정해 1000KBits/s 로 보레이트를 통일했다. 모드는 Full-Duplex 를 채택했으나, Half-Duplex 로도 구동 가능하다. 기본적으로 Zone 제어가 산하 모듈 관리자라는 개념을 살리기 위해 SPI Master 로 구동하게 했다. 따라서, 산하 모듈이 데이터를 제공할 때에는 별도의 DRDY 핀으로 신호를 보내 데이터가 준비됨을 알려 마스터가 읽어가도록 했다. 마스터 측에서 CS 는 코드 상에서 명확한 구분을 위해 Software NSS 를 채택했다.

3.3 CAN

HCLK = 64MHz, APB1 = 36MHz 기준 Prescaler = 4, Time Quanta in Bit Segment 1 = 12 Times, Segment 2 = 3 Times 로 세팅해 보레이트를 500KBits/s 로 보레이트를 설정했다. 모든 노드에서 동일한 보레이트를 가져야 한다. 해당 설정으로 샘플링 포인트는 81.25%로 적당한 값이 되도록 했다.

Zone 제어기 고장시 아예 버스에서 분리하는 기능을 위해, 굳이 필요없는 자동 버스 OFF 관리 등의 기능은 전부 Disable 했다.

버스의 반사파를 상쇄하기 위해 120ohm 저항을 별도로 버스 양단에 삽입했다.

4. STATE 설계 (목표 기능)

전체 시스템이 상태를 기반으로 동작하도록 FSMD 를 구현했다. 복잡하게 if 문이 중첩되고 반복되지 않기에 스파게티 코드를 방지하고, 코드 흐름이 명확해 문제 발생 시 디버깅이 용이하고 예측 가능성이 좋다. 아래 STATE 와, 상태에 따른 모듈의 동작을 표로 첨부한다.

상태 변화는 OFF->READY->DRIVE->(EVENT) EMERGENCY ->(EVENT) READY 의 Sequence 를 갖는다. CENTRAL_DOWN, BROKEN_SELF, MOTOR_ERR 등의 기타 상태는 EVENT 로, Sequence 와 무관하게 발생할 수 있다. DRIVE 상태에서 초음파 측정 값이 임계값(실험을 위해 편리하게 3cm 선정)보다 작거나, 가속도 값이 갑자기 너무 크게 나타난다면(실험을 위해 편리하게 3g 선정) 충격 위험이나 충격을 받았다고 판정해 EMERGENCY 상태로 전환된다. EMERGENCY 상태에서는 Zone2 가 모터 드라이버에게 Duty = 0 명령을 내려 모터를 정지하도록 한다. 가속도 센서값이 1G 를 2 초간 유지한다면 정지 즉 충격 완료로 판정해 다시 READY 상태로 전환한다.

CENTRAL_DOWN 상태에서, 운행 중 중앙제어기가 망가져도 모터 제어는 계속 되어야 하기에 단순히 HEART BEAT 를 끄고 Zone1-Zone2 의 페달 데이터 전송은 유지하도록 해 페달 기능이 유지되도록 했다.

BROKEN_SELF 는, Zone 내부에서만 사용되는 상태로 스스로의 CAN 기능이 망가졌음을 나타낸다. 중앙 제어기측에서는 Heartbeat 가 들어오지 않는 Zone 을 DEAD 처리한다. 특히, Zone2 의 경우 CAN 기능이 고장나면 모터를 정지하는 명령을 모터드라이버에 전송하도록 했다.

STATE	설명	노드별 동작
OFF	탑승 전	중앙: Heartbeat 전송 및 Zone4 신호 대기 Zone1: Do Nothing Zone2: Do Nothing Zone3: Do Nothing Zone4: IR 을 통한 탑승 감지
READY	탑승 감지 후, 시동 버튼 대기	중앙: Heartbeat 전송 및 Zone4 신호 대기 Zone1: Heartbeat 전송 Zone2: Heartbeat 전송 Zone3: Heartbeat 전송, 환영등 LED 명령 Zone4: Heartbeat 전송, 시동 버튼 감지
DRIVE	페달 조작 가능 및 초음파, 가속도 데이터 취득 시작	중앙: Heartbeat 전송 Zone1: Heartbeat 전송, 페달 감지 및 초음파 감지 Zone2: Heartbeat 전송, 초음파 감지 및 모터 드라이버 명령 Zone3: Heartbeat 전송, 가속도 감지 및 LED OFF 명령

		Zone4: Hearbeat 전송
EMERGENCY	충돌 위험 상태, 모터 정지 명령 - 초음파 임계값 이하 혹은 가속도 값 임계값 이상.	중앙: Hearbeat 전송 Zone1: Hearbeat 전송 Zone2: Hearbeat 전송, 모터 드라이버 정지 명령 Zone3: Hearbeat 전송, 가속도 감지 및 비상등 LED 명령 Zone4: Hearbeat 전송
CENTRAL_DOWN	중앙 제어기 고장	중앙: Do Nothing Zone1: 페달 감지 Zone2: 모터 드라이버 전송 Zone3: 중앙 제어기 고장 LED 명령 Zone4: Do Nothing
BROKEN_SELF	노드 내부 고장	중앙: EMERGENCY 상태 전파 Zone: 앞서 서술함.
MOTOR_ERR	모터가 포함된 Zone2 가 고장난 특히 위험한 상황	중앙: Hearbeat 전송 Zone1: Hearbeat 전송 Zone2: 자신의 CAN STOP, 모터 드라이버 정지 명령 Zone3: Hearbeat 전송, 비상등 LED 명령 Zone4: Hearbeat 전송

5. CAN 상세 설계

5.1 CAN 제어 레지스터 셋팅.

3.3 CAN 설계와 같다.

5.2 CAN Data Packet

<pre>typedef union { struct { uint16_t data; uint8_t reserved[4]; }; </pre>	<p>data 필드는 다른 data 를 위한 필드이다. 실제 데이터가 탑재된다.</p> <p>reserved 는 단순 8 바이트를 맞추기 위해 추가했다. 패킷 일관성을 위함이다.</p> <p>counter 필드는 CAN 에러 감지를 위해, 첨가했다. 수신측에서의 카운터와 맞지 않으면, 최신 데이터가</p>
---	---

<pre>uint8_t counter; uint8_t checksum; } __attribute__((packed)) data16; uint8_t bytes[8]; } CAN_Payload_t;</pre>	<p>아님을 알 수 있으므로 송신측이 멈춤을 검증할 수 있다.</p> <p>checksum 필드는 데이터 변조 여부를 확인하기 위함이다. SPI 패킷에서는 crc 를 썼기에, CAN 에는 다른 방법을 써보고자 간단하게 checksum 을 사용했다.</p> <p>Union 을 사용해 1 바이트 단위로 접근이 가능해 CAN 통신에서 다루기 쉽도록 했다.</p> <p>컴파일러의 패딩 여부를 확인할 수 없어 강제로 패딩 없이 8 바이트를 맞추도록 세팅했다.</p>
--	---

5.3 CAN ID 설계

상태 전파를 가장 중요하게 여겼기에, 젤 낮은 ID 그룹을 부여했다. 이때, 중앙 제어기의 Heartbeat 는 별도로 삽입하기보단 상태 전파를 100Hz 로 하는 것으로 대체했다. 다음으로, 안전을 위한 센서 값인 초음파와 가속도 데이터의 ID 를 0x010 번째 그룹으로 설정했다. 페달 데이터는, 안전 센서보다 중요도가 낮다고 생각했기에 0x040 을 부여했다. 그리고 Zone4 의 IR, BTN 은 안전과 무관하기에 가장 높은 0x140 번째 ID 를 할당했다. 각 Zone 의 Heartbeat 는, 안전과 관련된 ID 보다 높게 0x070 번째로 설정해 우선 순위를 조정했다. 마지막으로, 모터 고장 상태 관련해서는 0x020 번째를 부여했다.

Zone1 - 초음파 : 0x010, 페달 : 0x040

Zone2 - 초음파 : 0x011, 모터상태 : 0x110, 모터 SPI 고장 : 0x020, 고장시 모터상태: 0x021

Zone3 - 가속도 : 0x012

Zone4 - IR : 0x142, BTN : 0x144

Central - 모터 고장 상태 전파 : 0x001, Emergency 상태 전파 : 0x002, 기타 상태 전파 0x004=Heartbeat

Zone HearBeat & Status - Zone1:0x071, Zone2:0x072, Zone3:0x073, Zone4:0x074

5.4 CAN 네트워크 부하

노드	주기가 있는 데이터 전송량
중앙 제어기	Hearbeat 100Hz
Zone1	초음파 10Hz, 페달 10Hz, Hearbeat 100Hz
Zone2	초음파 10Hz, 모터 드라이버 PWM 100Hz, Hearbeat 100Hz
Zone3	가속도 100Hz, Hearbeat 100Hz
Zone4	Hearbeat 100Hz

8 바이트 데이터 기준, 임의의 비트 스퍼핑을 고려해 $108+17(\text{비트스퍼핑}) = 125$ 비트로 한 번의 전송시의 비트양을 임의로 산정했다. 1Hz 전송은 1msg/s 이고, 1msg 는 125bit 라고 볼 수 있다. 따라서 1Hz 당 125bits/s 이다. 그러므로 주기적으로 CAN 버스에 올라가는 데이터의 총량은 $[100]+[10+10+100] + [10+100+100] + [100+100] + [100] = 730\text{msg/s} = 91250\text{bit/s}$ 이다. $91250/500000 [*100\%] = 18.25\%$ 로, 매우 낮은 안정적인 버스 점유율이다. 따라서, CAN 통신의 지연은 매우 낮을 것으로 예상할 수 있다. 쉽게 확인할 수 있는 SAE J1939 의 권장인 50%, CiA 권장인 30~40%보다도 낮으므로 매우 안정적인 점유율로 볼 수 있다.

5.5 CAN Filter

리스트 필터로 설정했다. 각 노드별 수신 ID 리스트는 아래 표에서 확인할 수 있다.

중앙 제어기	0x040 제외 모든 ID
Zone1	0x001, 0x002, 0x004
Zone2	0x002, 0x004, 0x040
Zone3	0x001, 0x002, 0x004
Zone4	0x001, 0x002, 0x004

페달-모터의 보다 빠른 반응을 위해, Zone1 의 페달 데이터(ID=0x040)은 중앙 제어기를 거치지 않고 바로 Zone2 로 전달된다.

6. 상세 설계

중앙 제어기, Zone 제어기, Zone 산하 제어기 모듈의 코드는, 보고서에 신기에 양이 많기에 별도의 파일로 제출한다. 주석을 꼼꼼히 달았기에, 코드 리뷰를 대체한다.

(이 점, 교수님께 별도로 허락 받았습니다.)

6.1 중앙제어기

리눅스 환경 실습을 위해, 리눅스를 기반으로 하는 RaspberryPi OS 를 사용할 목적으로 RaspberryPi Zero 2 W 를 사용했다. 신호 반응성은 떨어지나, RTOS 보다 다양한 기능을 제공하는 커널을 사용할 수 있어 기능 확장성이 좋다.

Raspberry Pi Zero 2 W 는 기본적으로 CAN 통신을 위한 컨트롤러나 트랜시버가 탑재되어 있지 않다. 따라서, SPI 를 활성화해 MCP2515 와 TJA1050 에 연결해 커널이 이를 인식해 네트워크 장치로 만들어주도록 했다. STM32F1 등의 MCU 에서는 CAN 을 시리얼 포트 비슷하게 처리하지만, 리눅스는 CAN 을 와이파이 같은 네트워크 인터페이스로 취급한다. 네트워크 소켓 API 기반인 SocketCAN 커널 드라이버를 사용해 CAN 데이터를 주고받도록 했다.

중앙 제어기는 Zone1~4로부터 제공받는 데이터를 처리해 시스템의 전체 상태를 결정하고, CAN 버스로 전파함과 함께 시리얼 출력을 통해 현 상태를 안내해주는 역할을 한다.

6.2 Zone1 제어기

중앙 제어기와 Zone2와의 CAN 통신, Zone 1 산하 모듈인 초음파 담당 STM32F103C8T6, 페달 담당 STM32F103C8T6와의 SPI 통신을 관할한다. I2C 통신을 사용하면 버스가 깔끔하나 보다 빠르게 통신하기 위해, 그리고 산하 모듈이 2개밖에 되지 않기에 SPI 를 채택했다.

FreeRTOS(CMSIS_V2) 환경을 기반으로 구성해, CAN 과 SPI 통신 관리를 멀티스레딩 환경으로 마치 병렬처리 하듯 구현했다. 각 통신당 하나의 Task 가 그 통신을 관할하게 하여, 통신 자원을 공유하지 않도록 해 concurrency 문제를 원천적으로 배척했다.

동작하는 태스크(스레드)는 spi1Task, spi2Task, canTask 이다. 각각 SPI1, SPI2, CAN 을 담당한다. 다만, 하위 센서의 값 전송과 중앙 제어기의 상태 변경 등의 우선 순위를 명확히 할 수 없어 동일한 우선 순위인 High 로 통일했다. 만일 canTask 의 우선 순위를 낮추더라도, spi1Task 와 spi2Task 는 DMA SPI 를 다루므로 점유 시간이 짧고, 이벤트 Driven 인 만큼 우선 순위를 좀 더 높게 뒀도 무관하다. 이 프로젝트에서처럼

우선 순위를 동일하게 설정해도 지터가 없거나 매우 짧게 나타나 정확한 측정이 불가할 정도로 세 태스크 전부 주기를 지키며 스케줄링이 잘 이루어짐을 LED 토글 코드를 삽입해 로직 아날라이저로 확인해 실시간성을 확인했다. FreeRTOS-CMSIS_V2 는 기본적으로 우선순위 선점형 라운드 로빈을 사용해, 우선순위 동등하다면 모두 공평한 시간만큼 CPU 를 할당받는다.

spi1Task 는 산하 모듈 중 초음파 모듈에서 초음파 측정을 완료하면, DRDY_SONIC 을 SET 한다. 이를 EXTI 로 받아 EXTI ISR 에서 spi1Task 에 FLAG_SONIC_READ 플래그를 osThreadFlagsSet 으로 전달한다. spiTask 는 해당 플래그를 대기하며 블락되어있다, 플래그가 들어오면 SPI1 으로 SPI 통신을 시작한다. 이때, SPI 는 해당 프로젝트에서 DMA 를 활용해 CPU 가 CAN 통신에 더 관여하도록 했다. DMA 에 SPI 요청을 넣으면, spi1Task 는 FLAG_SONIC_DMA_COMPELETE 를 기다리며 osThreadFlagsWait 으로 블락 상태에 들어간다. SPI 전송이 완료되면, 인터럽트 설정을 해놓았기에 TxRxCpltCallback ISR 이 실행된다. 해당 ISR 에서 spi1Task 에 FLAG_SONIC_DMA_COMPLETE 플래그를 보내 태스크를 깨워 SPI 통신을 마무리하고 CS(NSS)를 다시 HIGH 로 올려 통신 종료를 슬레이브에게 알린다. 전송받은 패킷에서 data 를 추출해 g_distance, 즉 초음파 거리로 저장한다. 이때, Cortex-M3 는 32 비트 아키텍처로 바이트, 하프워드, 워드 단위의 단순 대입은 Atomic 하게 처리되므로 별도로 보호하지 않아도 일관성 문제로부터 자유로울 수 있다. 즉, 16 비트인 g_distance 는 데이터가 찢어지는 등의 문제가 발생하지 않는다. SPI 패킷은 해당 프로젝트에서 모두 동일하며, 아래 첨부한다. pragma pack(push, 1)을 통해 패딩을 넣을거면 1 바이트 기준으로 넣으라고 컴파일러에게 알린다. 따라서, 패딩없이 8 바이트가 딱찬다. 또한, 동일한 구조체를 master-slave 가 공유하기에, 엔디언 문제에서 벗어나 rx_buffer.pkt.data 와 같이 편리하게 패킷에 접근할 수 있다.

spi2Task 는 spi1Task 와 동일한 구조이나, SPI2 를 다루고 페달(악셀, 브레이크)와 통신한다는 점만 다르다.

canTask 는 CAN 송신을 담당한다. 앞서 4. STATE 설계 및 5. CAN 상세 설계에서와 같이 동작하도록, switch(global_state) - case 문법을 기반으로 설계했다. STATE 에 따라 CAN 버스에 송신할 데이터를 결정한다. Hearbeat 는 OFF 와 CENTRAL_DOWN, BROKEN_SELF 상태를 제외한 모든 상태에서 송신하기에 Fall-Trough 를 의도적으로 사용해 코드 길이를 줄이고 시각적으로 편하게 했다. 또한, 중앙 제어기의 Heartbeat 감시는 가장 앞에 배치해, 중앙 제어기 다운시 switch 문에 앞서 state_global 을 수정할 수 있게 했다. 중앙 제어기에서 보내는 데이터는 RxFifo0MsgPendingCallback ISR 에서 처리한다. 체크섬, 카운터를 확인하고 오류가 있으면 건너뛴다. 정상적인 패킷이 들어왔다면, switch-case 문을 사용해 ID 를 구분해 대응되는 동작을 하도록

했다. canTask 에서 Hearbeat 를 감시할 수 있도록 해당 ISR 에서 중앙제어기가 보낸 데이터를 CAN 으로 수신한 시점을 기록해둔다. canTask 에서 해당 시점으로부터 200ms 간 중앙제어기로부터 수신하지 않았을 때, 중앙 제어기가 죽었다고 판단하고 state_global 을 CENTRAL_DOWN 으로 설정한다. 해당 태스크는 주기가 있는 태스크로, osDelayUntil 을 사용해 코드실행 딜레이 없이 정확하게 10ms 주기를 지키도록 했다.

추가적으로, STM32F103C8T6 의 경우 CAN 컨트롤러는 탑재되어 있어 TX 와 RX 신호는 만들 수 있으나, CAN 트랜시버는 없기에 SN65HVD230 모듈을 별도로 연결해 실제 차동 신호를 만들어주도록 했다.

산하 모듈 중 초음파 담당 STM32F103C8T6 는, 100ms 를 주기로 트리거 신호를 발사해 초음파 모듈이 초음파로 거리를 측정하게 한다. 에코는 TIM - PWM Input(combined channel) 기능으로 타이머 캡처 기능을 사용해 처리했다. 초음파를 발사하고 수집하는 시간이 필요하므로 안정적으로 10Hz 로 느리게 동작하게 했다. 전방 장애물 감지는 0.1 초 간격으로도 충분하다고 판단했기에, 10Hz 면 충분하다고 판단했다.

산하 모듈 중 페달 담당 STM32F103C8T6 는 Non-os 로 단일 메인 루프로 구동하며, 100ms 를 주기로 ADC 측정으로 가변 저항을 읽어온다. ADC1 - IN0 는 악셀 가변저항 페달을, IN1 은 브레이크 가변저항 페달을 읽는다. 사람의 반응속도를 고려했을 때, 200ms 의 지연정도는 페달 조작 시 사람 입장에서 쾌적하다고 느낄 수 있다. 따라서 100ms 면 매우 충분하다고 판단했다. 모든 SPI 통신에서 사용하는 패킷을 아래 첨부한다.

<pre>#pragma pack(push,1) typedef struct{ uint8_t header; int16_t data; uint8_t status; uint8_t seq; uint8_t reserved; uint8_t crc; uint8_t tail; }SPI_Packet_t; #pragma pack(pop)</pre>	<p>header 는 어떤 데이터인지 나타내주는 헤더이다.</p> <p>data 는 데이터를 담는 필드이다.</p> <p>status 는 정상적으로 측정한 데이터인지 나타내준다.</p> <p>seq 는 앞서 CAN 패킷과 같이 CNT 의 역할을 한다.</p> <p>reserved 는 8 바이트를 맞추기 위함이다.</p> <p>crc 는 데이터 변조 여부를 확인하기 위한 필드이다. tail 은 데이터의 끝을 알려준다.</p>
--	--

<pre>typedef union{ SPI_Packet_t pkt; uint8_t bytes[8]; }SPI_Buffer_t;</pre>	<p>CAN 과 마찬가지로, 유니온을 사용해 바이트 단위로도 접근이 용이하도록 했다.</p>
---	---

6.3 Zone2 제어기

Zone1 과 마찬가지로 FreeRTOS 기반이며 spi1Task, spi2Task, canTask 로 구성했다.

Zone1 의 canTask 에서, CAN 으로 송신한 데이터를 받으면 RxFifo0MsgPendingCallback 에서 이를 구분해 g_pedal 로 업데이트하고, spi2Task 에게 FLAG_MOTOR_SEND 플래그를 보내 osThreadFlagsWait 으로 블락중이던 spi2Task 를 깨운다. 이후 spi2Task 에서는 산하 모터 드라이버 담당 STM32F103C8T6 에게 SPI 로 데이터를 전송한다. 이 점을 제외하고는 기타 동작 원리는 Zone1 과 같다.

산하 제어기 중 초음파 센서 담당 STM32F103C8T6 는 Zone1 에서의 것과 같다.

산하 제어기 중 모터 드라이버 담당 STM32F103C8T6 는 Non-os 에서 구동하며, TB6612FNG 모터 드라이버 모듈을 제어한다. TIM 의 PWM 기능을 사용해 드라이버에게 PWM 신호를 인가한다. 드라이버의 VM(모터 인가 전압)은 12V 를 어댑터에서 그대로 출력해줬다. Zone2 제어기의 spi2Task 에서 SPI DMA 로 보낸 데이터를 받으면 콜백 ISR 를 실행시켜 데이터를 받았음을 spi_rx_flag 플래그를 SET 해 main 루프에서 처리하게 한다. 메인 루프에서 폴링 방식으로 해당 플래그를 체크하며, 수신한 데이터가 있는 경우에 패킷에서 데이터를 뜯어 이를 모터 드라이버의 PWM 듀티로 인가한다. 이후 곧바로 다음 데이터를 수신하기 위해 HAL_SPI_TransmitReceive_DMA 를 실행해준다. Multi-Duplex 모드이므로 해당 HAL 드라이버를 사용해도 된다. 오히려 더미 데이터를 송신하기에 클락에 유리하다.

6.4 Zone3 제어기

FreeRTOS 기반이며, 3 가지의 Task 를 갖는다. canTask, ledTask, accelTask 이다. 이번 제어기는 산하 모듈로 I2C 통신을 사용하는 가속도 센서(IMU 중 가속도 센서만 사용)와 통신해야 하므로, I2C 통신을 담당하는 accelTask 를 구동한다.

canTask 는 앞선 Zone 제어기들과 동일한 구조로 실행된다. 마찬가지로 CAN RX Callback 에서 반영해준 global_state 를 기반으로 switch 문을 실행해 목표 기능을 수행한다.

ledTask 는 앞선 Zone 제어기들의 spiTask 와 같다. global_state 를 switch 로 처리해 산하 LED 담당 STM32F103C8T6 에게 명령을 SPI 로 전송한다. 해당 태스크는, LED 를 제어하기 위한 태스크로 굳이 빠르거나 실시간성이 중요하지 않다. 따라서 매우 느린 주기인 300ms 로 폴링해 동작한다.

accelTask 는 I2C 통신을 담당한다. 가속도 센서에는 별도로 MCU 를 부여하지 않고, 곧바로 데이터를 받아왔다. 해당 센서 모듈 자체가 하나의 MCU 라고 간주했다. 별도의 드라이버를 작성해 100Hz 샘플 레이트, $\pm 2g$, Full-해상도 등의 초기화를 진행하고 초기의 오프셋을 제거하기 위해 캘리브레이션을 정지상태에서 진행한다. 그리고 Read_Accel()를 작성해 I2C-DMA 로 데이터를 받아오도록 했고, Accel_ProcessData()를 작성해 받아온 데이터를 보정(오프셋 보정 & 스케일 보정)하도록 했다. 스케일 보정 값은 데이터시트를 참고해 0.004g/LSB 로 적용했다. accelTask 에서 위의 함수들을 사용해 가속도 센서값을 받아온다. 100Hz 를 주기로 루프가 구동되며, DRIVE 와 EMERGENCY 상태라면 Read_Accel()을 실행 후 osThreadFlagsWait 으로 FLAG_I2C_DMA_COMPLETE 를 기다린다. 만약 수신이 완료되면 I2C_MemRxCpltCallback 에서 osThreadFlagsSet 으로 플래그를 set 해줘 태스크를 깨운다. 이후 Accel_ProcessData()를 실행해 데이터를 보정한다. 보정이 완료된 데이터 ax, ay, az 는 전역변수로 업데이트해준다. 이때, float 을 CAN 으로 보내면 기존에 사용하던 패킷을 수정해야한다. 8 바이트에 맞춰놓은 패킷 구조를 유지하고 싶어, 소수점 2 자리수 까지만 반영해도 충분하다 판단 후 (int16_t)(ax*100.0f)로 데이터를 가공한다. 수신 측인 중앙 제어기에서는 다시 100 을 나눈후 float 으로 형변환해 사용한다. 이때 중요한 점은 ax, ay, az 를 int16_t 로 사용했을 때 총합 48 비트로 일관성에 문제가 생길 수 있다. 잠깐의 대입이기에 매우 빨리 진행되므로, 단순히 critical section 으로 지정해 이 문제를 방지했다. (taskEnter_CRITICAL & taskEXIT_CRITICAL).

이때, ledTask 는 상대적으로 매우 느린 주기이며 매우 빠르게 작업 완료하는 Task 이므로 starvation 을 고려해 높은 스케줄링 우선순위를 고려했다. 다만, CAN 에서 LED 로 인해 상태 업데이트 늦어지면 안되므로 canTask 보다는 한단계 낮은 우선순위를 부여했다. accelTask 의 경우는 100Hz 로 LED 에 비해 매우 빠른 주기이므로, LED Task 보다 우선순위가 높다면 can 과 accel task 에 밀려 ledTask 가 starvation 에 빠질 수 있으므로 accelTask 는 한단계 낮은 우선순위를 부여했다. ledTask 가 매우 빠르게 작업이 완료되고 주기가 느리기 때문에 accelTask 에 지연이 발생하지는 않는 것으로 관찰했다.

6.5 Zone4 제어기

Zone4 는 오직 OFF 상태에서 IR 감지, READY 상태에서 버튼 감지, 그리고 HEART-BEAT 기능만을 수행하여 개발 시간을 단축하고자 Non-os 를 채택했으며 산하 모듈을 두지 않았다. 메인 함수의 루프에서 heartbeat 송신 및 감지를 모두 처리하고, switch 문을 사용해 state 에 따른 동작을 수행한다. 앞선 제어기들과 마찬가지로 RxFifo0MsgPendingCallback 에서 state 를 업데이트한다.

6.5 전원 구성

12V-3A 정격 출력 스펙의 어댑터를 LM2596S 를 사용한 DC-DC 벅 컨버터를 사용해 3.3V 로 Step Down 해 MCU 에게 전원을 공급했다. Raspberry Pi Zero 2W 는 노트북에 연결해 전원을 별도로 공급했다. 스타 그라운드 처리해 모든 GND 를 한 지점으로 모아 전압의 기준점을 맞추었다.

초음파 담당 STM32F103C8T6 등에서 전원 문제로 정상 동작하지 않음을 디버깅을 통해 관찰했다. 따라서, 각 모듈을 위한 전원 공급 라인에 10uF 전해 커패시터로 전압을 안정화하고, 104pF 세라믹 커패시터로 순간적 전류 공급을 담당하도록 바이패스 커패시터들을 구성해 각 보드 전원 공급을 안정화했다.

7. 결과 분석

코드와 결과의 양이 매우 많아 보고서 분량 제한 상 부득이하게 별도의 파일로 첨부드립니다. 양해 부탁드립니다.

8. 고찰

해당 프로젝트는 상태를 기반으로 전체 시스템을 운용하는 임베디드 시스템을 구현한 프로젝트이다. 이를 위해 중앙 제어기와 4 개의 Zone 제어기를 배치해, 중앙 제어기가 관리하는 상태에 따라 시스템이 동작하도록 시스템 전체를 구성했다.

특히, 수업에서 배운 SPI 통신 인터페이스를 활용해 한 Zone 과 그 산하 모듈 간의 통신 시스템을 구현했다. 수업에서 배우지 않은 CAN 통신을 학습해 적용하기도 하고, 여러 모듈을 시스템에 집적하는 경험을 했다. 단순히 충돌 대응 및 페달로 모터를 조작하는 기능을 넘어, 노드들의 고장에 대비하는 Fail-safe, 그리고 초음파 센서를 2 개 배치에 Fault-Tolerance 기능을 고려해 보다 높은 완성도의 시스템을 구현했다.

이전 실습 혹은 이전 학기들에서의 프로젝트와 다르게 이번 프로젝트는 기획과 설계에 많은 시간을 할애했다. State 설계, Zone 내부 Task 구동 방식 설계, CAN 통신 및 SPI 통신 설계에 많은 기한을 사용했다. 따라서, 이미 잘 짜여진 설계도 위에서 코드 작업을 진행해 매우 수월하고 빠르게 작업을 마무리할 수 있었다. 또한, 중앙 제어기, Zone 제어기, Zone 산하 제어기가 분리되어 있기에 각각을 작업한 후 합치는 방식으로 작업해 팀원과 작업 분할에 유리했다.

시스템 전체가 FSMD 기반으로 동작하고, Zone 내부에서도 기능별 Task 가 구분되어 있고 디버깅 과정도 매우 원활했다. 예를 들어, 중앙 제어기의 로그에서 Zone1의 초음파 데이터를 취득하지 못하는 문제에 대해 빠르게 Zone1의 CAN 통신 -> SPI 통신 순으로 동작을 관찰했다. 통신 과정에서는 문제를 발견할 수 없어 산하의 초음파 제어기 STM32F103C8T6 파트를 관찰했고, 또 그 내부에서 main의 루프가 정상 실행함을 확인해 결국 초음파 센서 문제로 특정지을 수 있었다. 다른 센서로 교체해도 문제가 동일해, 전원 문제를 의심했고 멀티미터로 확인한 결과 초음파 센서에 5V보다 낮은 전압이 인가되고 있음을 확인했다. 따라서 바이패스 커패시터를 달아 보드에 전원이 확실히 공급되도록 해 문제를 해결했다.

또한, Zone2가 OFF 상태에서 READY가 되어도 의도와 다르게 Heartbeat가 나가지 않고 CAN 통신이 먹통이 되는 것을 관찰했다. 이에 task들의 우선 순위는 동일하니 starvation 문제는 제외시켰다. 따라서 Heart beat를 송신하는 canTask를 문제 원인으로 삼았고 로직을 관찰한 결과 fall-through 설계에 실수가 있어 Heart beat 송신이 되지 않는 것을 확인할 수 있었다.

이외의 많은 문제들에 대해서도 이처럼 모듈, 그리고 기능별로 잘 분리가 되어있도록 설계해냈기에 디버깅이 매우 빠르고 쉽게 진행되었다. 보다 복잡한 시스템을 설계할 때, 표준에 따르거나 모듈화를 철저히 하는 것의 중요성을 배웠다.

결과적으로, 목표한 기능이 STATE가 전환되가며 동작함을 확인했기에, 프로젝트의 목표를 달성했다고 판단한다.

아쉬운 점은, Zone Architecture에서 Zone 제어기들 간의 통신은 배선 용이성과 무게 감소로 인한 연비 향상 이점을 위해 CAN 통신을 주로 사용한다고 배웠다. 따라서 CAN2.0을 실습해보고자 CAN 통신으로 중앙 제어기와 Zone들 간의 통신을 구현했다. 하지만, Zonal Architecture에서는 Zone 제어기와 중앙 제어기 간의 통신은 Ethernet으로 구현하나, Ethernet 모듈을 별도로 구매하기에 부담스러워 포기한 점은 아쉬운 점이다.

또한 비용과 기한 문제로 카메라나 라이다 센서를 넣지 못했다. 이후 해당 임베디드 시스템의 완성도를 높이고 대량의 데이터를 시스템에서 처리해보는 경험을 위해

별도로 카메라 모듈을 Raspberry Pi zero 2W 에 추가 연결해 EMERGENCY 조건으로 추가해보고자 한다.

9. 참고 문헌

[1] STM32F1 시리즈 HAL & LL DRIVER

chrome-

extension://efaidnbmnnnibpcajpcgclclefindmkaj/https://www.st.com/resource/en/user_manual/um1850-description-of-stm32f1-hal-and-lowlayer-drivers-stmicroelectronics.pdf

[2] STM32F103C8T6 Datasheet

chrome-

extension://efaidnbmnnnibpcajpcgclclefindmkaj/https://www.st.com/resource/en/datasheet/stm32f103c8.pdf

[3] 그 외 STM32 문서

https://www.st.com/en/microcontrollers-microprocessors/stm32f103/documentation.html

[4] HC-SR04 Datasheet

chrome-

extension://efaidnbmnnnibpcajpcgclclefindmkaj/https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf

[5] ADXL345 Datasheet

https://www.alldatasheet.co.kr/datasheet-pdf/view/254714/AD/ADXL345.html

[6] TB6612FNG Datasheet

chrome-

extension://efaidnbmnnnibpcajpcgclclefindmkaj/https://cdn.sparkfun.com/datasheets/Robotics/TB6612FNG.pdf