리눅스 기초 텔레그램 봇 만들기

후방 감지 센서

김가람 단국대학교 산업보안학과

Dankook univ.

과목명 : 리눅스기초
담당교수 : 이승광
학과 : 산업보안학과
학번 : 32230324
이름: 김가람
제출일 : 2025/06/18

목 차

Ⅰ. 서론

본 보고서는 라즈베리파이에서 파이썬을 활용해 시스템을 설계하고 텔레그램을 통해 원격으로 명령을 주고받을 수 있는 봇을 바탕으로 후방 감지 시스템을 구현하는 것을 목표로 한다. 이 시스템은 초음파 센서를 통해 뒷쪽의 거리 정보를 실시간으로 측정하고 물체가 일정 거리 이상 가까워졌을 때 LED 경고등을 깜박이거나 텔레그램을 통해 경고 메시지를 전송하는 기능을 수행한다. 또한 사진 캡처 방식을 통해 사용자가 후방 상황을 간편하게 확인할 수 있도록 구성하였다. 이에 따라 본 보고서에서는 시스템의 전체적인 하드웨어 및 소프트웨어 설계 과정을 설명하고 텔레그램 봇 명령 처리, 초음파 센서 제어, LED 경고 시스템, 사진 전송 등 주요 구현 요소들을 분석하고자 한다.

Ⅱ. 본론

1. 시스템 설계

1) 하드웨어 구성

- 라즈베리파이 4 Model B

시스템의 중심이 되는 소형 컴퓨터로 리눅스 기반의 운영체제를 설치하여 파이썬 프로그램을 실행하며 GPIO 핀을 통해 외부 센서 및 장치와 연동이 가능하다.

- 저항 (220Ω, 560Ω)

LED마다 1개씩 직렬로 연결하여 과전류로 인한 손상을 방지하였다. 이를 통해 LED에 흐르는 전류를 제한함으로써 안정적인 동작을 가능하게 한다. 본 시스템에서는 220Ω 저항을 각각 1개씩 연결하고자 하였으나 수량이 부족해 1개는 560Ω으로 대체하였다.

- 브레드보드 및 점퍼 케이블

센서와 부품을 라즈베리파이 GPIO 핀에 연결하기 위한 도구들로 납땜 없이 빠르고 유연하게 회로를 구성할 수 있게 해준다.

- 초음파 센서 (HC-SR04)

후방 감지를 위한 핵심 센서로 TRIG 핀을 통해 초음파 신호를 보내고 ECHO 핀으로 반사되어 돌아오는 신호를 받아 물체까지의 거리를 계산한다. 거리 측정 결과는 LED 제어와 텔레그램 메시지 처리에 활용된다.

· VCC : 5V 전원
· GND : 접지
· TRIG : GPIO20 핀 (출력용)

· ECHO : GPIO16 핀 (입력용)

- USB 카메라

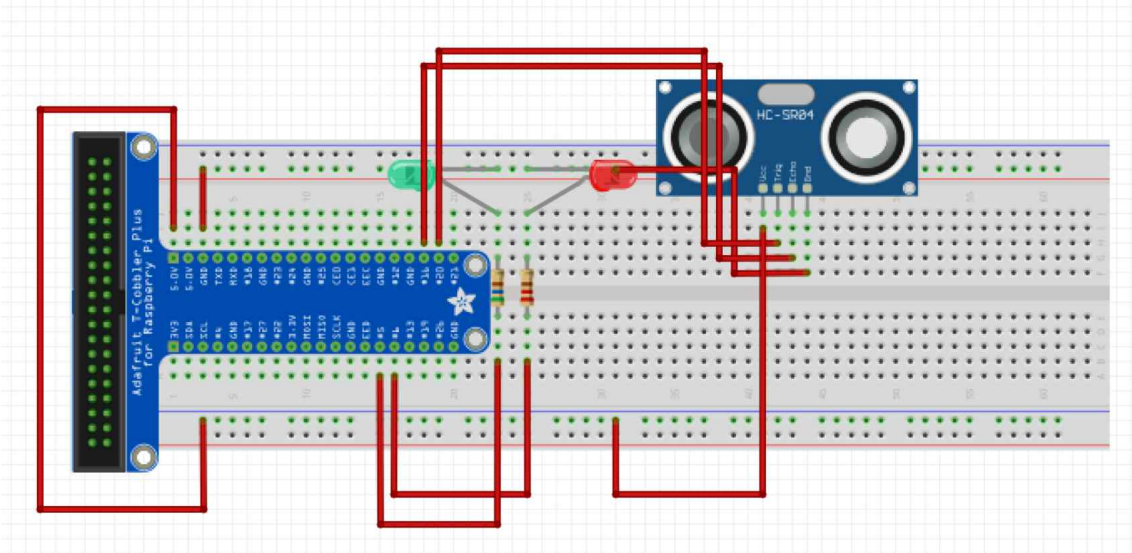
실시간 후방 정보를 얻기 위해 사용하며, 라즈베리파이에 연결하여 1.5초 간격으로 사진을 촬영하고 이를 텔레그램 봇을 통해 사용자에게 전송한다.

- LED 2개

사용자에게 시각적으로 거리 경고를 하기 위해 사용한다. 거리 값에 따라 깜빡이는 속도가 조절되며, 물체와 가까워질수록 더 빠르게 깜빡이도록 설정하였다.

· LED 1 (긴다리) : GPIO6 핀 (출력용)
· LED 2 (긴다리) : GPIO5 핀 (출력용)

- 회로도



2) 소프트웨어 구성

본 시스템의 소프트웨어는 Python3을 기반으로 구현되었으며, 라즈베리파이의 GPIO 핀을 직접 제어하고 카메라 모듈을 활용하기 위해 리눅스 환경에서 구동되도록 설계하였다. Telegram Bot API와 OpenCV, RPi.GPIO, asyncio 등의 라이브러리를 활용하여 텔레그램 기반의 원격 후방 감지 시스템을 구현하였다.

- 필수 라이브러리 구성

```
import cv2
import logging
import os
import asyncio
import RPi.GPIO as GPIO
import time
from telegram import Update
from telegram.ext import ApplicationBuilder, CommandHandler, ContextTypes
```

· **OpenCV (cv2) :** 카메라 제어 및 영상 처리

· **telegram, telegram.ext :** 텔레그램 봇 API 연동

· **RPi.GPIO :** 라즈베리파이 GPIO 핀 제어

· **asyncio :** 멀티태스킹 및 비동기 프로그래밍

· **logging :** 시스템 로그 관리 및 디버깅

· **os :** 파일 시스템 조작

· **time :** 시간 측정 및 타이밍 제어

- 시스템 초기 설정

· 텔레그램 봇 토큰 설정

BotFather로부터 발급받은 고유 토큰을 입력한다. (보안상 토큰은 가렸다.)

```
# 텔레그램 토큰
TOKEN = '                                                    '
```

· 로그 출력 포맷 설정

로그가 출력될 때 시간, 모듈 이름, 로그 레벨, 메시지 형식을 설정한다. 이때, 로그 레벨 WARNING 이상만 출력되게 하여 불필요한 로그를 최소화한다.

```
# 로그 설정
logging.basicConfig(format = '%(asctime)s - %(name)s - %(levelname)s - %(message)s', level=logging.WARNING)
```

· GPIO 핀 매핑 설정

라즈베리파이의 GPIO 핀을 사용하기 위해 먼저 BCM 방식으로 핀을 설정하고 실행 시 나타날 수 있는 불필요한 경고 메시지를 제거한다. 이후, 초음파 센서와 LED를 제어하기 위해 각각의 핀을 입출력 모드로 초기화한다. LED는 GPIO 6번과 5번 핀에 연결하여 출력 모드로 설정하고, 초음파 센서의 TRIG(송신)은 GPIO 20번 핀, ECHO(수신)은 GPIO 16번 핀에 연결하여 각각 출력과 입력으로 설정한다.

```
# 핀 설정
LED_PIN_1 = 6
LED_PIN_2 = 5
TRIG = 20
ECHO = 16

GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(LED_PIN_1, GPIO.OUT)
GPIO.setup(LED_PIN_2, GPIO.OUT)
GPIO.setup(TRIG, GPIO.OUT)
GPIO.setup(ECHO, GPIO.IN)
```

· 전역 변수 설정

streaming_task와 led_task는 각각 스트리밍 작업(실시간 사진 전송)과 LED 제어 작업을 나타내는 비동기 작업 객체이다. 이 두 작업은 각각 stream()과 blink_led() 함수가 독립적으로 실행되도록 만들어준다. 이를 통해 텔레그램 봇이 여러 기능을 동시에 실행하고 관리할 수 있게 된다.
is_streaming 변수는 후방 감지 시스템이 현재 실행 중인지 아닌지를 나타내며, /backward 명령어를 통해 후방 감지가 시작되면 True로, /stop 명령어를 통해 후방 감지가 중단되면 False로 변경된다.
이를 통해 시스템의 중복 실행을 방지하고, 동작 여부를 쉽게 판단할 수 있다.

```
# 전역 변수
streaming_task = None
is_streaming = False
led_task = None
```

2 . 텔레그램 봇 구현

1) 봇 명령 처리 시스템

봇의 핵심 명령어는 /start, /backward, /stop이다. /start는 봇 초기화와 사용 가능한 명령어를 안내하는 명령어, /backward는 후방 감지 시스템을 시작하는 명령어, /stop은 후방 감지 시스템을 중지하는 명령어이다. 각 명령어는 telegram.ext 모듈의 CommandHandler를 통해 처리되며 비동시 함수로 구현되어 실시간 처리가 가능하다.

· /start

사용자가 봇과의 대화를 시작할 때 입력하는 기본 명령어로, 사용 가능한 기능들을 안내하는 역할을 한다. 해당 명령어가 입력되면 reply_text() 메서드를 통해 사용자에게 후방 감지 시스템을 시작하거나 종료할 수 있는 두 개의 명령어(/backward, /stop)를 안내한다.

```python
# /start 명령어
async def start(update: Update, context: ContextTypes.DEFAULT_TYPE):
    await update.message.reply_text("<명령어>\n/backward : 후방 감지 시작\n/stop : 후방 감지 중지")
```

· /backward

후방 감지 시스템을 시작하는 명령어로, 초음파 센서를 통해 거리를 측정하고 LED 경고 시스템과 실시간 사진 전송을 동시에 실행한다. 먼저, 이미 동작 중인 경우를 확인하여 중복 실행을 방지하고, 사용자에게 후방 감지 시작을 알리는 메시지를 전송한다. 이후 asyncio.creat_task()를 사용하여 두 가지 비동기 작업을 동시에 실행한다. 이때, stream()은 라즈베리파이 카메라로 촬영한 사진을 주기적으로 텔레그램에 전송함으로써 실시간으로 후방 상태를 확인할 수 있게 해주고, bling_led는 감지된 거리값에 따라 LED의 깜빡이는 속도를 조절하여 사용자에게 물체와의 거리가 가까움을 경고해준다.

```python
# /backward 명령어 → 영상 스트리밍 시작 (단, 텔레그램은 실시간 영상 서비스를 지원하지 않으므로 1.5초 간격으로 사진 전송)
async def backward(update: Update, context: ContextTypes.DEFAULT_TYPE):
    global streaming_task, led_task
    if streaming_task is not None and not streaming_task.done():
        await update.message.reply_text("이미 작동 중입니다.")
        return

    await update.message.reply_text("후방 감지를 시작합니다. /stop 으로 종료하세요.")
    print("후방 감지 시작")
    chat_id = update.effective_chat.id
    is_streaming = True

    # 스트리밍과 LED 깜빡임 동시에
    streaming_task = asyncio.create_task(stream(context, chat_id))
    led_task = asyncio.create_task(blink_led())
```

· /stop

감지 시스템을 안전하게 종료하는 역할을 해준다. is_streaming 플래그를 False로 변경하여 루프를 종료시키고, 사용자에게 중단 사실을 알리는 메시지를 전송한다. 그 후, streaming_task와 led_task로 실행 중이던 두 비동기 작업을 cancel() 메서드를 통해 중지시키고, asyncio.CacelledError 예외를 처리하여 프로그램의 비정상적인 종료를 방지한다.

```python
# /stop 명령어 → 영상 전송 종료
async def stop(update: Update, context: ContextTypes.DEFAULT_TYPE):
    global is_streaming, streaming_task, led_task
    if not is_streaming:
        await update.message.reply_text("후방 감지 실행 중이 아닙니다.")
        return

    is_streaming = False
    await update.message.reply_text("후방 감지를 중단했습니다.")
    print("텔레그램 봇 작동 중지")
    print("후방 감지 중지")
```

```
# stop 후 예외처리
try:
    if streaming_task:
        streaming_task.cancel()
        await streaming_task
except asyncio.CancelledError:
    pass

try:
    if led_task:
        led_task.cancel()
        await led_task
except asyncio.CancelledError:
    pass

streaming_task = None
led_task = None
```

2) 메시지 처리 및 응답

· 봇 명령어 등록

텔레그램 봇이 사용자의 명령어를 인식하고 해당 기능을 수행하게 하려면 각 명령어와 그것을 처리할 함수를 봇 애플리케이션에 등록해야 한다. ApplicationBuilder를 사용해 텔레그램 봇 애플리케이션을 생성하고, 앞서 정의한 봇 토큰을 통해 해당 봇에 접속할 수 있게 한다. add_handler()는 /start를 입력하면 start() 함수가 호출되고, /backward를 입력하면 backward()가 호출되는 것처럼 사용자가 입력하는 명령어와 그 명령어를 처리할 함수를 연결해준다.

```
# main 함수
def main():
    app = ApplicationBuilder().token(TOKEN).build()
    app.add_handler(CommandHandler("start", start))
    app.add_handler(CommandHandler("backward", backward))
    app.add_handler(CommandHandler("stop", stop))
```

· 응답 메시지

명령 실행 응답

/start 명령을 실행 했을 경우에는 사용 가능한 명령어 목록을 제공해주고, /backward 명령을 실행 했을 때는 후방 감지를 시작한다는 메시지, /stop 명령을 실행 했을 때는 후방 감지를 중단한다는 메시지를 사용자에게 전송한다.

```
    await update.message.reply_text("<명령어>\n/backward : 후방 감지 시작\n/stop : 후방 감지 중지")
```

```
    await update.message.reply_text("후방 감지를 시작합니다. /stop 으로 종료하세요.")
```

```
    await update.message.reply_text("후방 감지를 중단했습니다.")
```

상태 기반 응답

사용자가 /backward 명령을 입력했을 때 후방 감지가 이미 작동 중인 상황이라면 이미 작동이라는 메시지를 출력하여 중복 실행을 방지하고, /stop 명령을 입력했을 때 후방 감지가 작동 중이지 않은 상황이라면 실행 중이 아니라는 메시지를 출력하여 오작동 가능성을 줄인다.

```
        await update.message.reply_text("이미 작동 중입니다.")
```

```
        await update.message.reply_text("후방 감지 실행 중이 아닙니다.")
```

· 채팅 세션 관리

텔레그램 봇은 여러 사용자가 동시에 명령어를 입력하고 상호작용할 수 있는 환경을 제공하기 위해 각 사용자의 채팅 세션을 식별하고 관리한다. 이를 위해 update.effective_chat.id를 사용하여 명령어를 입력한 개별 사용자의 고유 채팅 ID를 추출하고 해당 ID를 기준으로 응답 메시지나 사진 전송 등을 수행한다. 예를 들어, 한 사용자가 /backward 명령어를 통해 후방 감지 기능을 실행하고 있을 때 다른 사용자가 같은 명령을 입력하더라도 별도의 채팅 ID를 통해 각자의 요청이 구분된다.

```
    chat_id = update.effective_chat.id
```

3. 후방 감지 시스템 구현

1) 초음파 센서 거리 측정

· TRIG

먼저, TRIG 핀에 매우 짧은 시간(10μs) 동안 HIGH(True) 신호를 보내 초음파를 발사한다. 이때, TRIG 핀에 0.05초 (50ms) 동안 LOW(False) 신호를 보내는 이유는 거리 측정을 시작하기 전에 초음파 센서를 재설정하거나 잔류 전압을 제거하여 센서가 이전 상태로 남아있는 것을 방지하고 새로 시작할 수 있게 하기 위해서이다.

```
# 거리 측정 함수
def measure_distance():
    GPIO.output(TRIG, False)
    time.sleep(0.05)
    GPIO.output(TRIG, True)
    time.sleep(0.00001)
    GPIO.output(TRIG, False)
```

· ECHO

초음파가 장애물에 부딪혀 반사되어 돌아오면 ECHO 핀에서 신호가 HIGH로 바뀌는데, 이 신호가 HIGH 상태로 유지되는 시간, 즉 왕복 시간을 측정한다. 이때, 센서 오작동, 너무 먼 거리, 장애물 없음 등으로 인해 ECHO 핀이 HIGH 상태로 변하지 않는 경우, while문 내부가 아예 실행되지 않기 때문에

pulse_end가 정의되지 않았다는 오류가 발생할 수 있다. 따라서 이를 방지하기 위해 while문 밖에 pulse_end의 기본값으로 현재 시간을 미리 넣어둔다. 또 ECHO 신호를 너무 오래 기다리는 것을 방지하기 위해 timeout 설정(40ms)을 추가하였다.

```
    pulse_start = time.time()
    timeout = pulse_start + 0.04

    while GPIO.input(ECHO) == 0 and time.time() < timeout:
        pulse_start = time.time()

    pulse_end = time.time()

    while GPIO.input(ECHO) == 1 and time.time() < timeout:
        pulse_end = time.time()
```

· 거리 계산

초음파가 왕복하는 데 걸린 시간(pulse_duration = pulse_end - pulse_start)을 측정하고, 이를 초당 343m 속도로 이동하는 음속을 기준으로 하고, 왕복 거리이므로 2로 나누고 cm를 기준으로 하므로 100을 곱해야 한다. (distance = pulse_duration * 343 * 100 / 2 -> distance = pulse_duration * 17150) 이후, 계산한 값을 소수점 둘째 자리까지 반올림하여 반환해준다.

```
    pulse_duration = pulse_end - pulse_start
    distance = pulse_duration * 17150
    return round(distance, 2)
```

2) LED 경고 시스템

· LED 깜빡임 간격 계산 함수

LED의 깜빡임 간격은 초음파 센서로 측정된 거리에 따라 조절된다. 0~10cm 구간에서는 0.1초 간격으로 매우 빠른 깜빡임을 보이며, 0~50cm 구간에서는 거리에 비례하여 간격이 증가하고, 50cm 이상 구간에서는 1초 간격으로 느린 깜빡임을 보인다. 특히, 0~50cm 구간에서는 거리를 50으로 나누어 0~1 범위의 비율 값을 생성한 후 min()을 사용해 계산된 값이 1.0을 초과하지 않도록 제한하고(50cm 이상의 거리에선 항상 1.0초), max()를 사용해 최종 결과가 0.1초 미만이 되지 않도록 하였다(10cm 미만의 거리에선 항상 0.1초).

```
def get_blink_interval(distance):
    if distance < 10:
        return 0.1
    elif distance > 50:
        return 1.0
    return max(0.1, min(1.0, distance / 50))
```

· LED 깜빡임 함수

사용자가 후방 상황에 대해서 좀 더 효과적으로 인식할 수 있게 하기 위해 2개의 LED를 사용하였다. is_streaming이라는 전역 변수가 True인 동안 반복 실행되며, 내부에서 거리 측정 함수(measure_distance)를 호출하여 후방 물체와의 현재 거리를 dist라는 변수에 실시간으로 가져온다. 이후 해당 거리 값을 바탕으로 LED 깜빡임 간격을 계산 해주는 함수(get_blink_interval())를 호출하여 interval이라는 변수로 결과를 받아온다. 이후 LED 1과 LED 2는 번갈아 켜진다. 이때, 먼저 LED 1을 먼저 켜고 LED 2를 끄며, 계산된 시간(interval)만큼 대기 후 LED 1을 끄고 LED 2를 켠 뒤, 동일한 간격으로 다시 대기하는 과정을 반복함으로써 LED 2개가 교대로 깜빡이게 하며, 후방의 물체와 가까워질수록 깜빡임 속도가 빨라지는 방식으로 사용자에게 경고해준다. 또한 후방 감지 기능이 중단될 경우, is_streaming의 값이 False로 바뀌면서 루프가 종료되고 LED 1과 LED 2 모두 OFF 상태로 설정되어 모든 LED가 꺼지게 된다.

```python
# LED 깜빡임 함수
async def blink_led():
    global is_streaming
    while is_streaming:
        dist = measure_distance()
        interval = get_blink_interval(dist)

        # LED1 ON, LED2 OFF
        GPIO.output(LED_PIN_1, GPIO.HIGH)
        GPIO.output(LED_PIN_2, GPIO.LOW)
        await asyncio.sleep(interval)

        # LED1 OFF, LED2 ON
        GPIO.output(LED_PIN_1, GPIO.LOW)
        GPIO.output(LED_PIN_2, GPIO.HIGH)
        await asyncio.sleep(interval)

    # 꺼질 때 둘 다 OFF
    GPIO.output(LED_PIN_1, GPIO.LOW)
    GPIO.output(LED_PIN_2, GPIO.LOW)
```

3) 실시간 사진 전송

이 기능은 사용자가 /backward 명령어를 입력하면 활성화되며(is_streaming의 값이 True로 설정되며), 라즈베리파이에 연결된 카메라 모듈을 통해 약 1.5초 간격으로 후방 사진을 촬영하여 사용자의 채팅방에 업로드한다. 해당 간격(1.5초)은 텔레그램 플랫폼의 특성상 실시간 영상 스트리밍을 직접 지원하지 않기 때문에 연속적인 사진 전송을 통해 실시간 스트리밍을 구현하기 위해 설정되었다.

이를 구현하기 위한 stream() 함수는 is_streaming이 True인 동안 루프가 유지되며, cv2.VideoCapture를 통해 카메라 프레임을 획득하고 이미지 저장 디렉토리에 임시 파일로 저장한다.

저장된 이미지를 context.bot.set_photo()를 통해 사용자에게 전송하고, 1.5초 동안 대기한 후 이를 반복된다. 이때 카메라가 정상적으로 작동하지 않으면 카메라를 열 수 없다는 오류 메시지를 출력하고 잠수를 종료한다. 한편, 거리 측정 결과에 따라(10cm 이내일 경우) 경고 메시지를 전송하는 기능 또한 이 함수 내부에 포함되어 있다. 이는 거리 측정 함수인 measure_distance()에는 사용자의 채팅 ID (chat_id)를 전달받을 방법이 없지만 stream() 함수는 촬영된 이미지를 채팅방에 전송하는 과정에서 chat_id를 인자로 직접 받아오기 때문에 사용자별 메시지 전송이 가능하다. 따라서 경고 메시지 기능을 stream() 함수에 포함시키는 것이 기능 구현 측면에서 더 편리한 선택이라 생각했다. 또 이전 메시지 전송 시점으로부터 5초가 경과한 경우에만 새로운 경고 메시지를 전송하도록 제한을 두었는데, 이는 거리 조건만으로 메시지를 전송할 경우, 너무 짧은 간격으로 전송되어 텔레그램 서버에 과부하를 줄 수 있다고 생각했기 때문이다.

```python
# 사진 캡처 함수
async def stream(context, chat_id):
    global is_streaming
    is_streaming = True
    last_warning_time = 0

    try:
        # 카메라 열기
        camera = cv2.VideoCapture(0, cv2.CAP_V4L)
        camera.set(cv2.CAP_PROP_BUFFERSIZE, 1)
        if not camera.isOpened():
            await context.bot.send_message(chat_id=chat_id, text="카메라를 열 수 없습니다.")
            is_streaming = False
            return

        while is_streaming:

            # 거리 측정
            dist = measure_distance()

            # 경고 메시지 전송 (5초 간격)
            current_time = time.time()
            if dist < 10 and (current_time - last_warning_time > 5):
                await context.bot.send_message(
                    chat_id=chat_id,
                    text=f"⚠️ 너무 가까워요! 물체가 {dist}cm에 있어요!"
                )
                last_warning_time = current_time

            # 사진 전송
            ret, frame = camera.read()
            if not ret:
                break
            path = "/tmp/stream.jpg"
            cv2.imwrite(path, frame)
            with open(path, "rb") as photo:
                await context.bot.send_photo(chat_id=chat_id, photo=photo)
            os.remove(path)
            await asyncio.sleep(1.5)

    finally:
        camera.release()
        is_streaming = False
```

4. 시스템 안정성 및 최적화

1) 비동기 프로그래밍 및 멀티태스킹

비동기 프로그래밍은 여러 작업을 동시에 처리할 수 있도록 해주며 각 작업의 I/O 작업을 기다리는 동안 다른 작업이 중단 없이 실행될 수 있도록 한다. 본 시스템에서는 asyncio 모듈을 활용하여 비동기 프로그래밍 기반의 멀티태스킹 구조로 설계하였다.

특히, 실시간 사진 촬영 및 전송을 담당하는 stream() 함수와 거리 측정 값을 바탕으로 LED 깜빡임을 제어하는 blink_led() 함수를 async def로 정의하였다. 이들은 await를 통해 실행을 일시 중단했다가 다시 이어가면서 여러 작업이 교차 실행되어 멀티태스킹 효과를 낸다. 이때, await는 해당 작업이 완료될 때까지 기다리되 기다리는 동안 다른 작업이 실행될 수 있도록 CPU를 양보하는데, 본 시스템에서는 이를 활용해 LED 깜빡임 간격이나 사진 전송 간격을 조절하거나 텔레그램 봇의 명령어 응답과 메시지 전송, 사진 전송 중에도 사용자의 명령에 즉각 반응할 수 있도록 해주었다.

또한, 사용자가 /backward 명령어를 입력하면 아래와 같이 asyncio.create_task()를 활용해 두 작업을 별도의 비동기 태스크로 병렬 실행한다. 이는 사진 전송과 LED 제어가 동시에 끊김없이 수행되도록 하며, 사용자가 /stop 명령어를 입력하면 cancel()로 취소 신호를 보내고 await로 실제 종료까지 기다려 태스크가 안전하게 종료되도록 한다.

· LED 깜빡임 간격 조절

```
# LED1 ON, LED2 OFF
GPIO.output(LED_PIN_1, GPIO.HIGH)
GPIO.output(LED_PIN_2, GPIO.LOW)
await asyncio.sleep(interval)

# LED1 OFF, LED2 ON
GPIO.output(LED_PIN_1, GPIO.LOW)
GPIO.output(LED_PIN_2, GPIO.HIGH)
await asyncio.sleep(interval)
```

· 사진 전송 간격 조절

```
await asyncio.sleep(1.5)
```

· 텔레그램 봇의 명령어 응답, 메시지 전송, 사진 전송

```
await update.message.reply_text("<명령어>\n/backward : 후방 감지 시작\n/stop : 후방 감지 중지")
await context.bot.send_message(chat_id=chat_id, text="카메라를 열 수 없습니다.")
await context.bot.send_message(
    chat_id=chat_id,
    text=f"⚠ 너무 가까워요! 물체가 {dist}cm에 있어요!"
)
```

```
await update.message.reply_text("이미 작동 중입니다.")
await update.message.reply_text("후방 감지를 시작합니다. /stop 으로 종료하세요.")
await update.message.reply_text("후방 감지 실행 중이 아닙니다.")
await update.message.reply_text("후방 감지를 중단했습니다.")
await update.message.reply_text("<명령어>\n/backward : 후방 감지 시작\n/stop : 후방 감지 중지")
await context.bot.send_photo(chat_id=chat_id, photo=photo)
```

· /backward 입력 시

```
# 스트리밍과 LED 깜빡임 동시에
streaming_task = asyncio.create_task(stream(context, chat_id))
led_task = asyncio.create_task(blink_led())
```

· /stop 입력 시

```
if streaming_task:
    streaming_task.cancel()
    await streaming_task
```

```
if led_task:
    led_task.cancel()
    await led_task
```

2) 예외 처리 및 리소스 관리

시스템이 실시간으로 하드웨어를 제어하고 사용자와 상호작용하는 구조이기 때문에 예외 상황에서도 안정적으로 동작하고 자원을 적절히 관리하는 것이 중요하다. 본 시스템에서는 이를 위해 각 기능별로 예외를 처리하고, 시스템 종료 시점에 GPIO와 카메라 자원을 정리하고 있다.

· stream()

try...finally 구조를 통해 도중에 예외가 발생하거나 작업이 취소되더라도 반드시 카메라 자원을 해제하도록 하였다.

```
try:
    # 카메라 열기
    camera = cv2.VideoCapture(0, cv2.CAP_V4L)
    camera.set(cv2.CAP_PROP_BUFFERSIZE, 1)
    if not camera.isOpened():
        await context.bot.send_message(chat_id=chat_id, text="카메라를 열 수 없습니다.")
        is_streaming = False
        return

    while is_streaming:
```

(생략)

```
finally:
    camera.release()
    is_streaming = False
```

· /stop 입력 시

사용자가 /stop 명령어를 입력할 경우에는 각각의 비동기 태스크를 cancel()로 취소한 후 await하여 실제로 종료될 때까지 기다리는데, 비동기 함수가 강제로 취소될 때, 즉 asyncio.create_task()로 실행 후 task.cancel()을 호출하면 자동으로 발생하는 예외인 asyncio.CancelledError를 안전하게 처리하여 프로그램이 멈추는 것을 방지하였다. 이후, 전역 변수(streaming_task, led_task)들을 재설정하여 다음 명령 실행 시 충돌을 방지하였다.

```
    # stop 후 예외처리
    try:
        if streaming_task:
            streaming_task.cancel()
            await streaming_task
    except asyncio.CancelledError:
        pass

    try:
        if led_task:
            led_task.cancel()
            await led_task
    except asyncio.CancelledError:
        pass

    streaming_task = None
    led_task = None
```

· main()

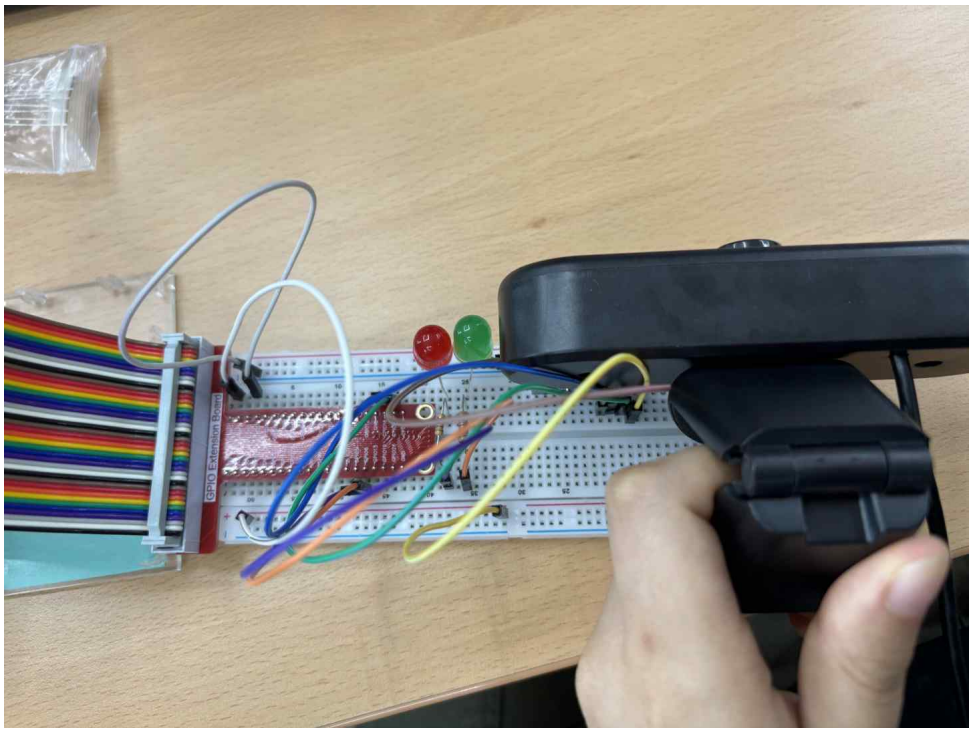
main() 함수에서는 finally 블록에서 전체 종료 시점에 GPIO.cleanup()을 호출하여 LED 핀과 초음파 센서 핀 등의 출력을 초기화함으로써 이후 시스템이 재시작되더라도 충돌이 발생하지 않도록 해준다.

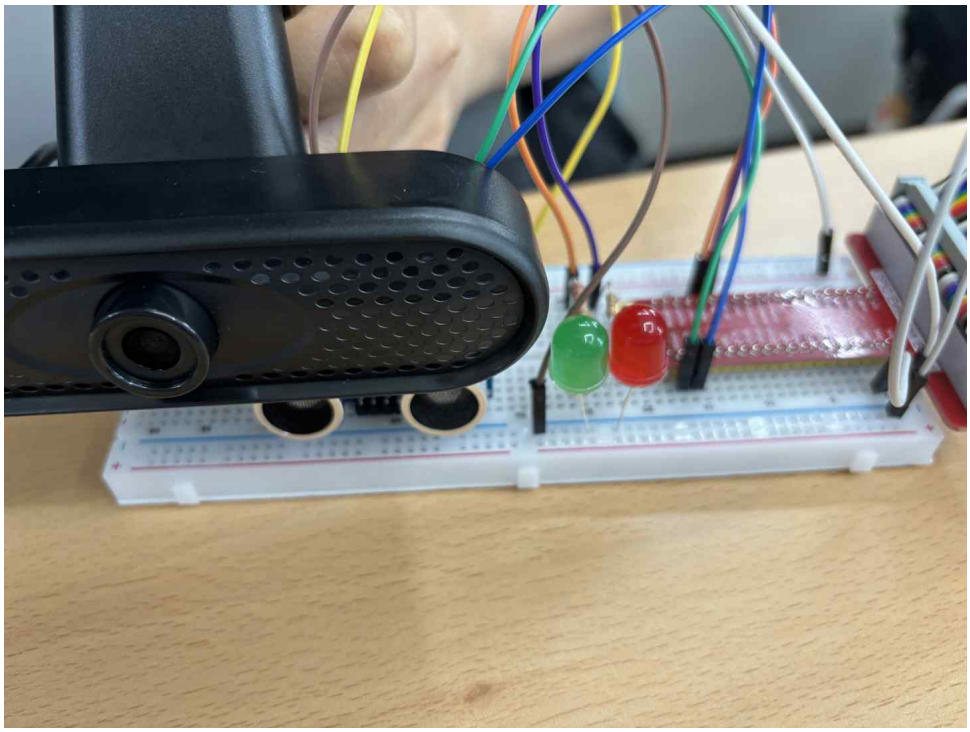
```
# main 함수
def main():
    app = ApplicationBuilder().token(TOKEN).build()
    app.add_handler(CommandHandler("start", start))
    app.add_handler(CommandHandler("backward", backward))
    app.add_handler(CommandHandler("stop", stop))

    try:
        app.run_polling()
        print("텔레그램 봇 작동 중...")

    finally:
        GPIO.cleanup()  # 여기서 한 번만 cleanup

if __name__ == '__main__':
    main()
```

5. 시스템 동작 확인

1) 최종 구현 결과 및 동작 테스트

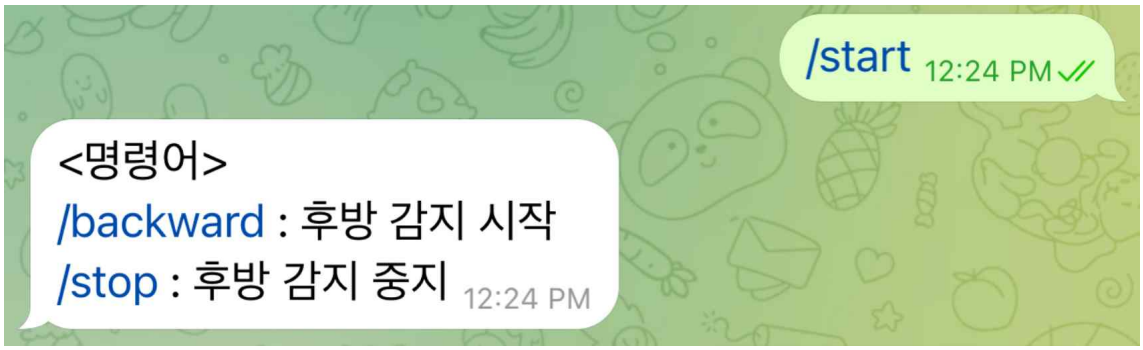
· 회로 구현





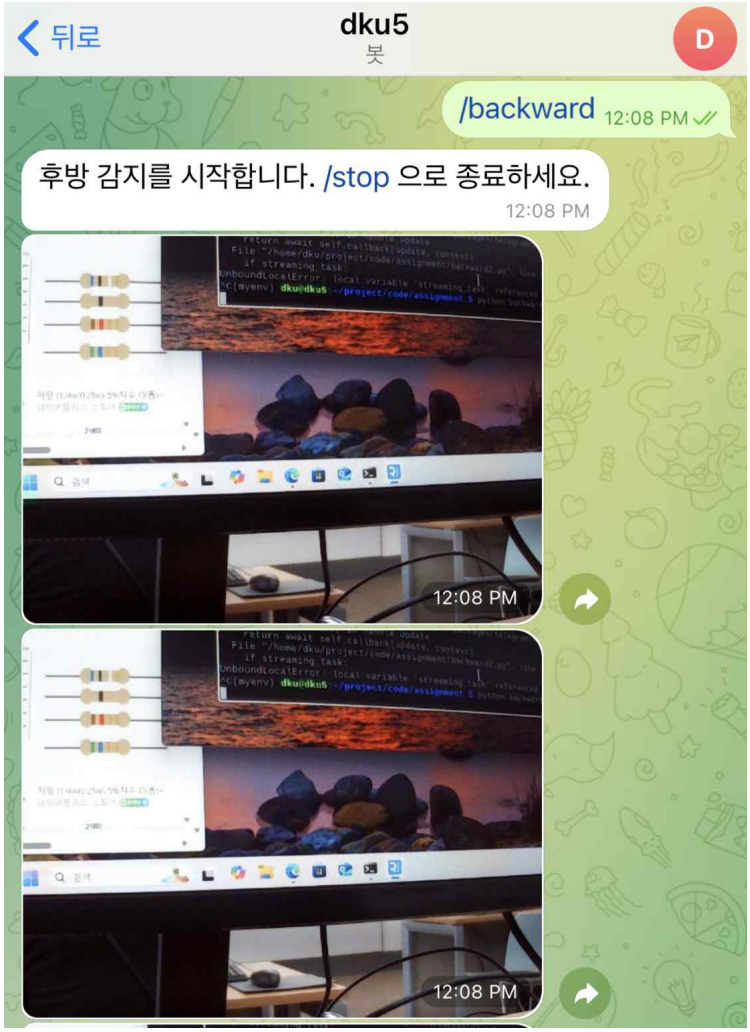
· **/start 명령어 사용**

사용 가능한 기능들에 대한 안내가 출력된다.

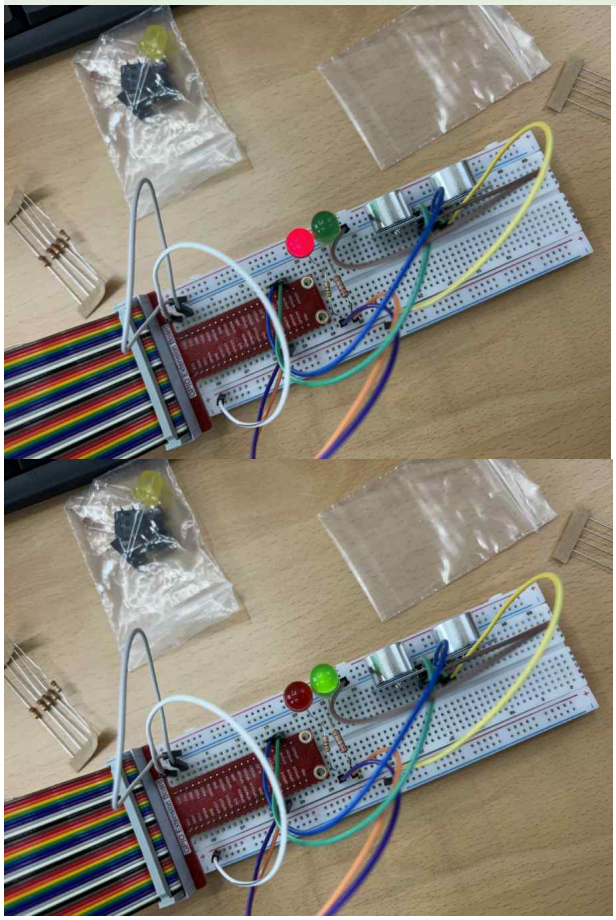


· **/backward 명령어 사용**

후방 감지를 시작한다는 안내 후 후방을 촬영해 실시간으로 사진을 전송하기 시작하고, LED 2개도 번갈아가며 깜빡이기 시작한다. 이때, /backward를 또 사용하면 이미 작동 중이라는 메시지가 출력된다.

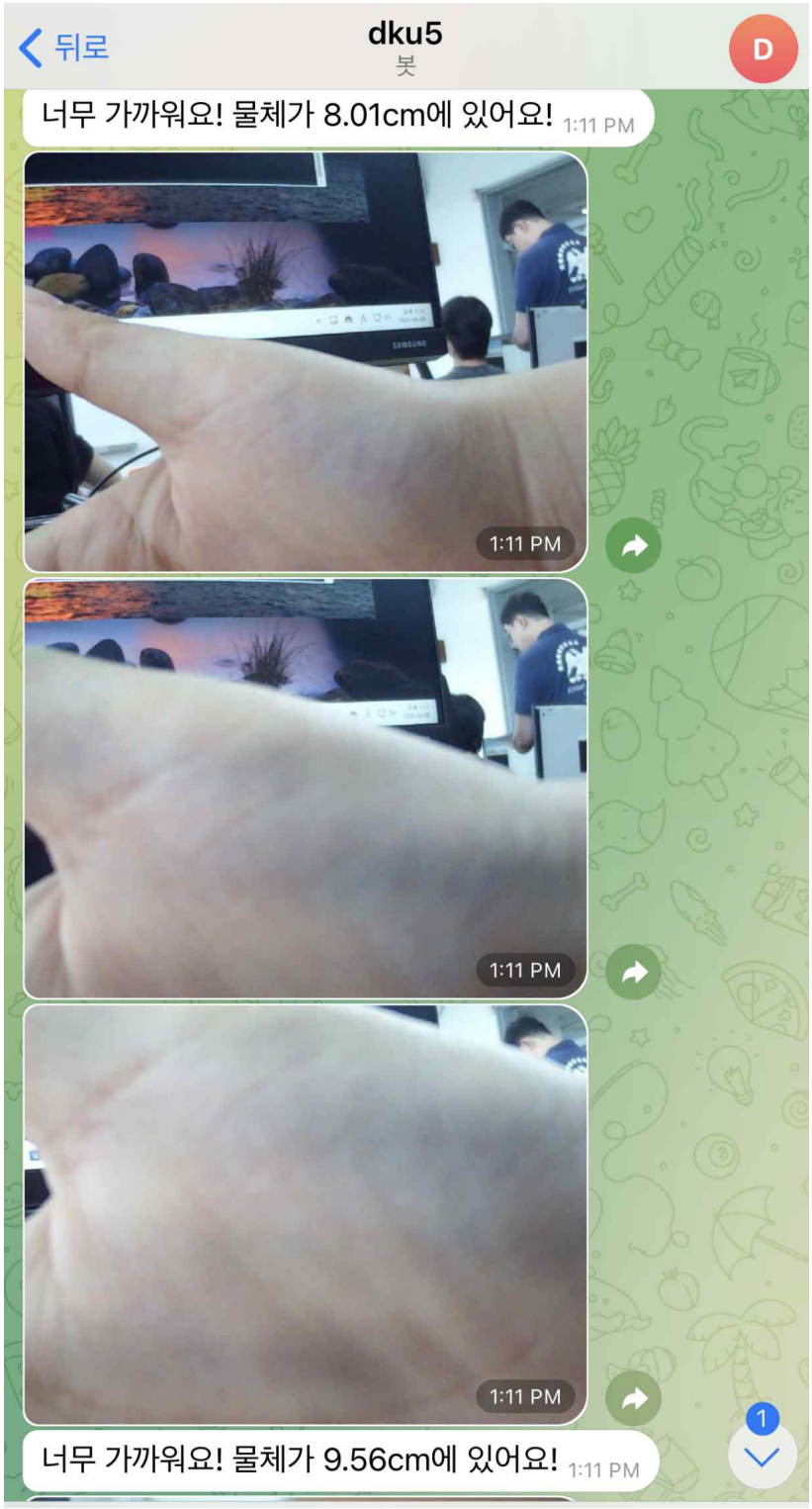



12:22 PM
/backward 12:22 PM
12:22 PM
이미 작동 중입니다. 12:22 PM



· 거리가 10cm 미만일 경우 + 5초 경과

후방에 있는 물체와의 거리가 10cm 미만일 경우 경고 메시지가 전송된다. 이전 경고 메시지 전송 후 지난 시간이 5초를 초과할 경우, 새로운 경고 메시지가 전송된다. 이때, LED 2개도 더 빠른 속도로 깜빡이기 시작하고, 10cm보다 멀어질 경우 속도가 다시 느려진다. 〈실행 영상〉



· **/stop 명령어 사용**

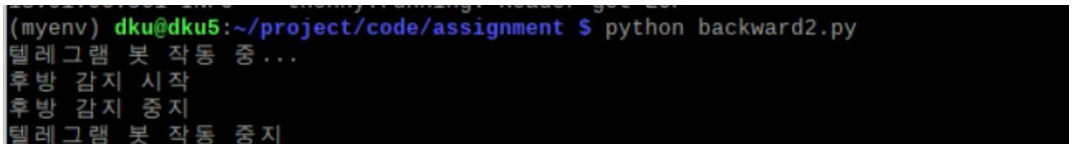
후방 감지를 중단한다는 메시지가 전송되고, 프로그램이 종료된다. 이때, /stop을 또 사용하면 후방 감지가 실행 중이지 않다는 메시지가 전송된다.

· **카메라가 정상적으로 작동하지 않는 경우**

카메라가 정상적으로 작동하지 않는 경우에 /backward를 사용하면 카메라를 열 수 없다는 메시지가 전송된다.

· 터미널 창

/start 명령 사용 시 텔레그램 봇이 작동 중이라는 메시지, /backward 명령 사용 시 후방 감지 시작 메시지, /stop 명령 사용 시 후방 감지 중지 및 텔레그램 봇 작동 중지 메시지가 출력된다.



2) 시스템의 한계점 및 개선 방향

· 실시간성

본 시스템은 텔레그램이 실시간 영상 스트리밍 기능을 지원하지 않기 때문에 1.5초 간격으로 사진을 촬영하여 전송하는 방식으로 동작하고 있다. 이로 인해 실시간 반응에 한계가 존재한다. 따라서 텔레그램은 명령어 입력 및 제어 용도로만 사용하고 웹 페이지(URL)나 전용 앱을 통해 실시간 영상을 확인할 수 있는 구조로 개선하는 것이 실시간성을 높이는 데 도움이 될 것으로 보인다.

· 경고 메시지 간격

본 시스템은 후방 물체와의 거리가 10cm 미만일 경우 경고 메시지를 전송하며, 이전 메시지 전송 시점으로부터 5초 이상 경과했을 때에만 새로운 경고 메시지를 전송하는 방식으로 동작한다. 하지만 이 5초의 간격 동안 사용자의 주의가 늦춰질 수 있다는 문제점이 존재한다. 따라서 경고 간격을 거리를 기반으로 조절하는 것이 더 효과적이라고 생각된다.

· 재사용

본 시스템은 /backward 명령어를 통해 후방 감지를 시작하고, /stop 명령어를 통해 이를 중지하면서 관련 태스크 및 GPIO 핀 등의 리소스를 정리하는 방식으로 동작한다. 하지만 /stop 이후 시스템을 재사용하고자 할 경우, /start 명령어를 입력한 뒤 /backward 명령어를 다시 입력해야 하는 번거로운 과정을 거치게 된다. 따라서 리소스를 정리하는 과정을 별도의 명령어(/reset)로 분리하여 후방 감지 기능이 종료된 이후 명령어 하나(/backward)로 재사용이 가능하도록 개선하는 것이 좋다.

Ⅲ. 결론

본 프로젝트에서는 라즈베리파이와 초음파 센서, LED, 카메라 모듈 등의 하드웨어와 텔레그램 봇을 활용해 후방 감지 시스템을 설계하고 구현하였다. 사용자가 텔레그램을 통해 명령을 내리면 초음파 센서를 통해 후방의 물체를 감지하여 LED 깜빡임 간격 조절과 경고 메시지 전송을 수행하며, 1.5초의 간격으로 후방 사진을 사용자에게 전송하는 방식으로 동작한다. 이 과정에서 비동기 프로그래밍을 활용해 여러 기능을 동시에 처리하도록 하였으며, 예외 처리와 리소스 관리를 통해 시스템 안정성도

확보하였다. 동작 테스트를 마친 후에는 실시간성, 재사용, 경고 메시지 간격 등과 같은 현재 시스템의 한계점들을 생각해보고, 각각에 대한 개선 방안에 대해서도 함께 고민해보았다. 특히 실시간성의 부분에서 아쉬움이 많이 남았기에 추후에는 영상 스트리밍 기술에 대해 공부하고 이를 적용해봄으로써 실시간성을 보다 향상시켜보고 싶다.

[부록]

1) 전체 소스 코드

```
import cv2
import logging
import os
import asyncio
import RPi.GPIO as GPIO
import time
from telegram import Update
from telegram.ext import ApplicationBuilder, CommandHandler, ContextTypes

# 텔레그램 토큰
TOKEN = '7866300477:AAHtXn7e33aU5L0VHhDXVp9egl3CgSFisuA'

# 로그 설정
logging.basicConfig(format = '%(asctime)s - %(name)s - %(levelname)s - %(message)s',
level=logging.WARNING)

# 핸들러 함수
# /backward 명령어 입력 시
# /stop 명령어가 입력될 때까지
# 카메라로부터 받은 사진을 실시간으로 텔레그램 채팅창에 전송

# /start 명령어
async def start(update: Update, context: ContextTypes.DEFAULT_TYPE):
    await update.message.reply_text("〈명령어〉₩n/backward : 후방 감지 시작₩n/stop : 후방
감지 중지")
```

```
# 핀 설정
LED_PIN_1 = 6
LED_PIN_2 = 5
TRIG = 20
ECHO = 16

GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(LED_PIN_1, GPIO.OUT)
GPIO.setup(LED_PIN_2, GPIO.OUT)
GPIO.setup(TRIG, GPIO.OUT)
GPIO.setup(ECHO, GPIO.IN)

# 거리 측정 함수
def measure_distance():
    GPIO.output(TRIG, False)
    time.sleep(0.05)
    GPIO.output(TRIG, True)
    time.sleep(0.00001)
    GPIO.output(TRIG, False)

    pulse_start = time.time()
    timeout = pulse_start + 0.04

    while GPIO.input(ECHO) == 0 and time.time() < timeout:
        pulse_start = time.time()

    pulse_end = time.time()

    while GPIO.input(ECHO) == 1 and time.time() < timeout:
        pulse_end = time.time()

    pulse_duration = pulse_end - pulse_start
```

```
    distance = pulse_duration * 17150
    return round(distance, 2)

def get_blink_interval(distance):
    if distance < 10:
        return 0.1
    elif distance > 50:
        return 1.0
    return max(0.1, min(1.0, distance / 50))

# 전역 변수
streaming_task = None
is_streaming = False
led_task = None

# LED 깜빡임 함수
async def blink_led():
    global is_streaming
    while is_streaming:
        dist = measure_distance()
        interval = get_blink_interval(dist)

        # LED1 ON, LED2 OFF
        GPIO.output(LED_PIN_1, GPIO.HIGH)
        GPIO.output(LED_PIN_2, GPIO.LOW)
        await asyncio.sleep(interval)

        # LED1 OFF, LED2 ON
        GPIO.output(LED_PIN_1, GPIO.LOW)
        GPIO.output(LED_PIN_2, GPIO.HIGH)
        await asyncio.sleep(interval)

    # 꺼질 때 둘 다 OFF
```

```
    GPIO.output(LED_PIN_1, GPIO.LOW)
    GPIO.output(LED_PIN_2, GPIO.LOW)

# 사진 캡처 함수
async def stream(context, chat_id):
    global is_streaming
    is_streaming = True
    last_warning_time = 0

    try:
        # 카메라 열기
        camera = cv2.VideoCapture(0, cv2.CAP_V4L)
        camera.set(cv2.CAP_PROP_BUFFERSIZE, 1)
        if not camera.isOpened():
            await context.bot.send_message(chat_id=chat_id, text="카메라를 열 수 없습니다.")
            is_streaming = False
            return

        while is_streaming:

            # 거리 측정
            dist = measure_distance()

            # 경고 메시지 전송 (5초 간격)
            current_time = time.time()
            if dist < 10 and (current_time - last_warning_time > 5):
                await context.bot.send_message(
                    chat_id=chat_id,
                    text=f"    너무 가까워요! 물체가 {dist}cm에 있어요!"
                )
                last_warning_time = current_time

            # 사진 전송
```

```
            ret, frame = camera.read()
            if not ret:
                break
            path = "/tmp/stream.jpg"
            cv2.imwrite(path, frame)
            with open(path, "rb") as photo:
                await context.bot.send_photo(chat_id=chat_id, photo=photo)
            os.remove(path)
            await asyncio.sleep(1.5)

    finally:
        camera.release()
        is_streaming = False

# /backward 명령어 → 영상 스트리밍 시작 (단, 텔레그램은 실시간 영상 서비스를 지원하지 않으므로
1.5초 간격으로 사진 전송)
async def backward(update: Update, context: ContextTypes.DEFAULT_TYPE):
    global streaming_task, led_task
    if streaming_task is not None and not streaming_task.done():
        await update.message.reply_text("이미 작동 중입니다.")
        return

    await update.message.reply_text("후방 감지를 시작합니다. /stop 으로 종료하세요.")
    print("후방 감지 시작")
    chat_id = update.effective_chat.id
    is_streaming = True

    # 스트리밍과 LED 깜빡임 동시에
    streaming_task = asyncio.create_task(stream(context, chat_id))
    led_task = asyncio.create_task(blink_led())

# /stop 명령어 → 영상 전송 종료
async def stop(update: Update, context: ContextTypes.DEFAULT_TYPE):
```

```
    global is_streaming, streaming_task, led_task
    if not is_streaming:
        await update.message.reply_text("후방 감지 실행 중이 아닙니다.")
        return

    is_streaming = False
    await update.message.reply_text("후방 감지를 중단했습니다.")
    print("텔레그램 봇 작동 중지")
    print("후방 감지 중지")

    # stop 후 예외처리
    try:
        if streaming_task:
            streaming_task.cancel()
            await streaming_task
    except asyncio.CancelledError:
        pass

    try:
        if led_task:
            led_task.cancel()
            await led_task
    except asyncio.CancelledError:
        pass

    streaming_task = None
    led_task = None

# main 함수
def main():
    app = ApplicationBuilder().token(TOKEN).build()
    app.add_handler(CommandHandler("start", start))
    app.add_handler(CommandHandler("backward", backward))
```

```
    app.add_handler(CommandHandler("stop", stop))

    try:
        app.run_polling()
        print("텔레그램 봇 작동 중...")

    finally:
        GPIO.cleanup()  # 여기서 한 번만 cleanup

if __name__ == '__main__':
    main()
```

2) 회로도