개발문서

> 본 문서는 한화비전 VEDA 부트캠프 임베디드 리눅스 프로그래밍 과정(2025.12.01 ~ 2025.12.22)을 완료하고, 학업 성취도를 평가하기 위한 심화 실습 평가(2025.12.23 ~ 2025.12.29)에 대한 개발 문서입니다.

1) 프로젝트 개요

1-1 프로젝트 목적

본 프로젝트는 한화비전 VEDA 부트캠프 '임베디드 리눅스 프로그래밍' 과정에서 학습한 핵심 이론과 실습 내용을 실제 시스템으로 구현하여 그 작동 원리를 깊이 있게 이해하는 데 목적이 있다.

라즈베리 파이 환경을 기반으로 TCP/IP 소켓 통신을 활용하여 LED, 부저, 조도 센서, 7-세그먼트 등 다양한 하드웨어를 원격으로 제어하고 모니터링하는 시스템을 구축한다. 특히 멀티스레드 기반의 장치 제어, 데몬 프로세스, 종료 시그널 처리, 동적 라이브러리, 메이크 파일 작성 등을 직접 구현함으로써, 하드웨어 제어부터 네트워크 통신 및 운영체제에 이르기까지 리눅스 시스템 전반에 대한 이해할 수 있다.

1-2 주요 기능

- **장치 제어**

  라즈베리파이 4에서 WiringPi 라이브러리를 사용하여 LED, 패시브 부저, 조도 센서, 7-세그먼트를 제어한다.

- **Makefile**

  과제에서 요구하는 디렉터리 구조를 반영하여, 서버와 클라이언트 실행 파일 및 장치별 공유 라이브러리(.so) 생성을 자동화한다.

- **동적 라이브러리**

  장치별 제어 소스 파일을 만들고, 이를 공유 라이브러리(.so)로 만들어 코드 상에서 동적으로 로드(dlopen, dlsym)하여 사용한다.

- **TCP/IP 소켓**

  TCP/IP 소켓 통신을 구축하고 사용자가 클라이언트 프로그램을 통해 하드웨어 제어 명령을 보내면 서버에서 이를 제어한다.

- **데몬 프로세스**

  서버를 백그라운드 서비스 형태로 실행하여 터미널과의 연결을 끊고, 서버 프로그램을 실행한 사용자가 로그아웃 하여도 서버 프로그램이 종료되지 않도록 만든다.

- **멀티 스레딩**

  pthread 라이브러리를 사용하여 장치 제어 함수를 스레드에서 실행함으로써 장치 제어와 클라이언트 요청 처리를 동시에 수행한다.

- **종료 시그널 처리**

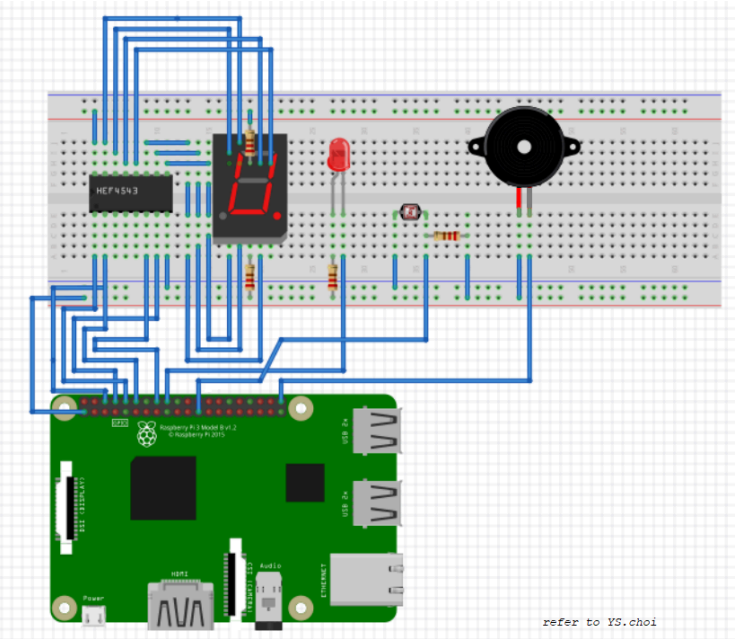
클라이언트 프로그램의 비정상 종료를 방지하기 위해, 사용자의 종료 메뉴 선택 또는 SIGINT 신호에만 프로그램이 종료되도록 시그널 처리한다.

2) 개발 일정

1일차: 기초 환경 구축

- **회로 구성**

  과제 테스트 환경에 맞추어 라즈베리파이, LED, 조도 센서, 패시브 부저, 7-세그먼트와 7447 BCD 디코더를 사용하는 회로를 구성하였다.



장치명	Physical 핀	WiringPi 핀	인터페이스 방식
7447 - D (MSB)	16	15	Digital Output
7447 - C	12	16	Digital Output
7447 - B	10	1	Digital Output
7447 - A (LSB)	8	4	Digital Output
LED	32	26	Soft PWM
포토레지스터 (CDS)	23	14	Digital Input
패시브 부저	40	29	Soft Tone

- **장치 테스트**
  - 장치는 수업에서 사용한 교재 `사물인터넷을 위한 리눅스 프로그래밍 with 라즈베리파이`와 함께 제공된 예제 중 Chapter3 파트의 소스코드를 이용해 테스트하였다.
  - 예제 깃허브 링크https://github.com/valentis/LinuxProgrammingWithRaspberryPi.git
- **프로젝트 디렉터리 구조**

```
📁rpi-remote-control-system/
├──📁code/
│   ├──📁client/
│   │   └── client.c        # 클라이언트 코드
│   └──📁server/
│       ├── buzzer.c        # 부저 제어
│       ├── cds.c           # 조도 센서 제어
│       ├── common.h        # 프로토콜 및 매크로 상수 정의
│       ├── fnd.c           # 7-세그먼트 제어
│       ├── led.c           # LED 제어
│       └── server.c        # 서버 코드
├──📁exec/                  # 빌드 시 생성되는 실행 파일 디렉터리
│   └──📁lib/               # 빌드 시 생성되는 공유 라이브러리(.so) 디렉터리
├── .gitignore
└── Makefile
```

- **Git 원격 저장소 생성과 라즈베리파이 연결**
  - 과제 깃허브 링크

    https://github.com/joco9822/rpi-remote-control-system.git
- **장치 제어 소스 파일(.c) 작성**

2일차: Makefile 및 동적 라이브러리

- **Makefile 작성**
  - all 타겟에 의존성을 설정하여 전체 프로젝트를 한 번에 빌드하도록 구현한다.
  - 자동 변수 활용: $@(타겟), $<(첫 번째 의존성), %(패턴) 등의 기호를 사용하여 효율적인 컴파일 규칙을 정의한다.
- **모듈화**
  - dlopen, dlsym 함수를 통해 실행 시점에 장치 라이브러리를 동적으로 로드하는 구조를 검증한다.

3일차: 멀티스레드 구현

- **장치 실행 함수를 멀티 스레드로 실행**
  - pthread_create와 pthread_detach를 사용하여 조도 센서 값에 따른 LED 제어, 부저 음악 연주, 7-세그먼트 카운트다운을 독립적인 실행 흐름으로 분리한다.
  - 장치 제어 시 발생하는 지연(delay)이 메인 서버 로직 및 네트워크 수신을 방해하지 않도록 처리한다.

4일차

- **종료 시그널 처리**
  - 클라이언트 프로그램의 비정상 종료를 방지하기 위해, 사용자의 종료 메뉴 선택 또는 SIGINT 신호에만 프로그램이 종료되도록 시그널 처리한다.

- **데몬 프로세스**
  - 서버를 백그라운드 프로세스로 동작시켜 터미널 종료 후에도 서비스가 유지되도록 설정한다.
- **문서 작성**
  - 프로젝트 사양을 담은 개발 문서와 README.md, 실행 결과를 기록한 running.txt를 작성한다.

3) 세부 구현 내용

3-1 장치 제어

1. **LED 제어: led.c**
   - `softPwm` 라이브러리를 사용하여 소프트웨어적으로 PWM(Pulse Width Modulation) 신호를 생성한다. 이는 디지털 신호의 HIGH 유지 시간(Duty Cycle)을 조절하여 LED에 공급되는 평균 전력을 제어함으로써 밝기를 변화시키는 방식이다.
   - `led_init`에서 0~100 범위를 설정하며, 회로 구성상 `softPwmWrite` 값이 100일 때 LED가 꺼지고 0일 때 가장 밝게 켜지도록 구현되었다. `led_control` 함수는 단순 ON/OFF 외에도 2~100 사이의 값을 받아 세밀한 밝기 조절 기능을 수행한다.
2. **패시브 부저 제어: buzzer.c**
   - `softTone` 라이브러리를 통해 가변 주파수의 방형파를 생성한다. 패시브 부저는 인가되는 신호의 주파수에 따라 진동수가 달라져 서로 다른 음을 낼 수 있는 특성을 가진다.
   - `buzzer_init`으로 톤 생성 핀을 초기화하고, `buzzer_control`에서 전달받은 주파수 데이터(data)를 `softToneWrite` 함수로 출력하여 멜로디를 연주한다.
3. **조도 센서 제어: cds.c**
   - 빛의 세기에 따라 저항값이 변하는 포토레지스터와 고정 저항(10kΩ)을 직렬로 연결한 분압 회로를 활용한다. 주변이 어두워지면 센서 저항이 커져 GPIO 입력 전압이 특정 임계치를 넘게 되고, 이를 통해 디지털 `HIGH`(1) 값을 읽어온다.
   - 핀 모드를 `INPUT`으로 설정하여 라즈베리 파이가 센서의 전압 상태를 감지하도록 한다. `cds_read` 함수는 `digitalRead`를 호출하여 현재 주변의 밝기 상태를 반환한다.
4. **7-세그먼트 제어: fnd.c**
   - 7447 BCD 디코더 IC를 사용하여 4비트의 이진수 데이터를 7-세그먼트 구동 신호로 변환한다. 이는 GPIO 핀 4개만으로 0부터 9까지의 숫자를 표시할 수 있게 하여 효율적인 자원 관리를 가능하게 한다.
   - 4개의 출력 핀(A, B, C, D)을 각각 20,21,22,23의 가중치에 대응시킨다. `fnd_control` 내에서 비트 연산(`&`)을 사용해 정수 데이터의 각 비트 값을 추출하고, 해당 결과에 따라 핀의 `HIGH`/`LOW` 상태를 결정한다.

3-2 Makefile

- **빌드 메커니즘**

- `all` 타겟을 기점으로 라이브러리와 실행 파일 간의 의존성 트리를 구성한다. `make` 실행 시 파일의 존재 여부와 수정 시간을 확인하여 변경된 소스만 다시 컴파일하는 효율적인 빌드를 수행한다.

- **자동 변수 활용**
  - 규칙 내에서 `$@`는 타겟 파일명을, `$<`는 첫 번째 의존성 소스 파일을 가리킨다. 이를 통해 수많은 장치 파일을 개별적으로 기술하지 않고 하나의 규칙으로 처리한다.
- **패턴 규칙**
  - `lib%.so`와 같이 `%` 기호를 사용하여 소스 파일명과 타겟 라이브러리명을 일대일로 매칭한다. 새로운 장치 소스가 추가되어도 Makefile 수정 없이 자동으로 동적 라이브러리를 빌드할 수 있는 확장성을 가진다.
- **컴파일 옵션**
  - 위치 독립적 코드(`fPIC`)와 공유 객체 생성(`shared`) 옵션을 사용하여 실행 중에 동적으로 로드 가능한 `.so` 파일을 제작한다.

3-3 TCP/IP 소켓 통신

- **통신 프로토콜 설계: common.h**
  - 서버와 클라이언트 간의 데이터 규격을 맞추기 위해 Protocol 구조체를 정의한다. 구조체는 명령 종류(type), 대상 장치 ID(deviceid), 데이터 값(data), 메시지(msg)로 구성된다. `#pragma pack(push, 1)`을 사용하여 구조체 멤버 사이의 패딩을 제거하고 데이터 크기를 최적화하여 네트워크 전송 효율을 높인다.
- **서버측 소켓 구현: server.c)**
  - `socket` 함수로 IPv4, TCP 타입의 소켓을 생성한다. `bind` 과정을 통해 서버의 포트 번호를 5100으로 설정하고, `listen` 함수를 호출하여 클라이언트의 접속 요청을 대기하는 상태로 진입한다. `accept` 함수는 클라이언트의 연결 요청을 수락하여 통신을 위한 새로운 전용 소켓을 생성하며, 이후 `read` 함수를 통해 클라이언트가 보낸 패킷을 수신한다.
- **클라이언트측 소켓 구현: client.c**
  - 프로그램 실행 시 인자로 받은 서버의 IP 주소와 지정된 포트(5100) 정보를 바탕으로 `connect` 함수를 호출하여 서버에 연결을 요청한다. 사용자가 메뉴를 선택하면 해당 명령에 맞는 데이터를 Protocol 구조체에 담고, `write` 함수를 사용하여 서버로 패킷을 전송한다.
- **동작 원리**
  - TCP는 연결 지향성 프로토콜로, 데이터의 신뢰성 있는 전송을 보장한다. 서버는 무한 루프를 돌며 클라이언트의 접속을 상시 대기하며, 연결된 후에는 패킷 내부의 `deviceid`를 분석하여 적절한 하드웨어 제어 로직을 호출하는 인터페이스 역할을 수행한다. 클라이언트와 서버가 동일한 구조체를 공유하므로 복잡한 파싱 과정 없이 직관적인 명령 전달이 가능하다.

3-4 서버 데몬 프로세스

- **백그라운드 실행 환경 구축**
  - 서버 프로그램을 터미널 제어에서 분리하여 백그라운드에서 상주하며 실행되는 서비스 형태로 구현한다. 이를 통해 사용자가 로그아웃하거나 터미널 창을 닫아도 서버가 중단되지 않고 클라이언트의 접속을 상시 대기할 수 있다.

- **구현 단계 및 원리**
  1. **fork() 호출**: 부모 프로세스를 종료하고 자식 프로세스만 남겨 프로세스를 백그라운드로 전환한다. 자식 프로세스는 고아 프로세스가 되어 `init`(PID 1) 프로세스에 입양된다.
  2. **setsid() 호출**: 새로운 세션을 생성하여 프로세스를 제어 터미널(TTY)과의 관계에서 완전히 독립시킨다. 이를 통해 터미널 종료 시 발생하는 시그널의 영향을 받지 않게 된다.
  3. **디렉터리 및 마스크 설정**: `chdir("/")`를 통해 실행 경로를 루트로 변경하여 파일 시스템 언마운트를 방해하지 않도록 하고, `umask(0)`로 프로세스가 생성하는 파일의 권한을 초기화한다.
  4. **표준 입출력 차단**: `stdin`, `stdout`, `stderr`를 닫거나 `/dev/null`로 리다이렉션하여 터미널로의 불필요한 입출력을 차단하고 자원 낭비를 방지한다.

3-5 멀티 스레딩을 통한 장치 제어

- **병렬 처리의 필요성**
  - 서버가 클라이언트의 제어 명령을 대기(Blocking)하는 동안에도 조도 감시, 부저 연주, 카운트다운과 같은 지연 시간(Delay)이 발생하는 작업을 동시에 수행하기 위해 멀티스레딩을 적용한다. 이를 통해 특정 장치의 동작이 전체 시스템의 응답성을 떨어뜨리는 것을 방지한다.
- **스레드 생성 및 관리**
  - `pthread` 라이브러리를 사용하여 장치별 독립적인 실행 흐름을 생성한다.
  - `pthread_create`를 호출하여 각 스레드 함수(`cds_thread`, `buz_thread`, `fnd_thread`)를 실행한다.
  - `pthread_detach`를 사용하여 스레드 종료 시 별도의 `join` 과정 없이 시스템 자원이 즉시 자동 회수되도록 설정한다.
- **스레드별 세부 로직**
  1. 조도 감시 스레드 (`cds_thread`)

     `cds_flag`가 `ON`인 동안 무한 루프를 돌며 조도 센서 값을 읽는다. 센서 값에 따라 어두우면 LED를 켜고 밝으면 끄는 실시간 감시 기능을 수행한다.
  2. 부저 연주 스레드 (`buz_thread`)

     미리 정의된 음계 배열(`notes`)을 순회하며 멜로디를 재생한다. `buz_flag`를 수시로 체크하여 연주 도중에도 즉시 정지 명령을 처리할 수 있도록 설계되었다.
  3. 카운트다운 스레드 (`fnd_thread`)

     전달받은 숫자부터 0까지 1초 간격으로 감소하며 FND에 표시한다. 숫자가 0에 도달하면 부저로 알람을 울린 후 종료된다.
- **공유 자원 제어**
  - `cds_flag`, `buz_flag`, `fnd_flag`와 같은 전역 플래그 변수를 사용하여 메인 스레드에서 각 서브 스레드의 실행 및 종료 상태를 제어한다. 클라이언트로부터 중단 명령이 오면 해당 플래그를 `OFF`로 변경하여 스레드가 안전하게 루프를 빠져나와 종료되도록 구현한다.

3-6 클라이언트 종료 시그널 처리

- **접속 종료 감지 및 자원 정리**

    - `server.c`에서 `read()` 함수가 0을 반환하면 클라이언트와의 연결이 끊어진 것으로 간주하여 통신 루프를 탈출한다. 연결이 종료되면 즉시 `cds_flag`, `buz_flag`, `fnd_flag`를 `OFF`로 전환하여 실행 중인 모든 장치 제어 스레드를 중지시킨다. 이후 `led_ctl(OFF)`, `buz_ctl(OFF)` 등을 호출하여 하드웨어를 안전한 상태로 초기화하고 클라이언트 소켓을 닫는다.

- **SIGINT (종료 시그널) 핸들링**
    - 사용자가 `Ctrl+C` 등을 통해 서버를 강제 종료할 때, 시스템에 등록된 시그널 핸들러가 동작하도록 설계한다. 이는 프로세스가 갑자기 종료되어 GPIO 출력이나 스레드가 비정상적으로 남는 것을 방지하기 위함이다. 핸들러 내부에는 스레드 종료 대기 및 모든 하드웨어 장치를 초기화하는 로직을 포함하여 자원 유실(Leak)을 막는다.
- **SIGPIPE (비정상 종료) 방지**
    - TCP 통신 중 클라이언트가 예기치 않게 종료된 상태에서 서버가 데이터를 전송(write)하려고 하면 `SIGPIPE` 시그널이 발생하여 서버 프로세스가 종료될 수 있다. 이를 방지하기 위해 해당 시그널을 무시(`SIG_IGN`) 처리하거나 핸들러를 통해 예외 처리를 수행함으로써 서버의 연속적인 가동성을 보장한다.

4) 문제점 및 보완 사항

4-1 문제점

- **1:1 연결의 한계**

  현재 서버는 `accept` 이후 `while` 루프를 통해 한 클라이언트의 요청을 처리하므로, 먼저 접속한 클라이언트가 연결을 끊기 전까지는 다른 클라이언트가 접속할 수 없는 구조적 한계가 있다.

- **동적 라이브러리 경로 의존성**

  `dlopen` 호출 시 `./lib/`와 같은 상대 경로를 사용하고 있어, 서버 프로그램 실행 위치에 따라 라이브러리 로드에 실패할 가능성이 존재한다.

- **데이터 피드백 부재**

  클라이언트는 장치에 제어 명령을 보낼 수 있지만, 조도 센서의 현재 값이나 카운트다운의 진행 상태를 서버로부터 실시간으로 수신하여 화면에 표시하는 기능이 부족하다.

- **스레드 자원 관리**

  전역 플래그를 사용하여 스레드를 종료하지만, `pthread_detach` 상태에서 스레드가 루프를 완전히 빠져나오기 전에 서버가 종료될 경우 하드웨어 자원이 불안정한 상태로 남을 위험이 있다.

4-2 보완 사항

- **다중 접속 지원**

  서버에 `fork()`를 이용한 멀티 프로세스 방식이나 `pthread`를 이용한 멀티 스레드 접속 처리, 또는 `select/epoll`과 같은 I/O 멀티플렉싱 기법을 도입하여 여러 클라이언트의 동시 접속을 지원해야 한다.

- **센서 데이터 역전송 구현**

  서버에서 센서 값을 읽어 클라이언트로 주기적으로 전송하는 기능을 추가하고, 클라이언트 UI에서 실시간 모니터링이 가능하도록 개선한다.

- **설정 파일 도입**

  하드웨어 핀 번호나 라이브러리 경로 등을 소스 코드 내에 하드코딩하지 않고, 외부 설정 파일(JSON 또는 CONF)에서 읽어오도록 구현하여 시스템 유연성을 확보한다.

- **동기화 객체 적용**

  현재는 단순 플래그 변수를 사용하여 스레드를 제어하지만, 공유 자원에 대한 안전한 접근을 위해 뮤텍스(Mutex)나 세마포어(Semaphore)를 적용하여 데이터 무결성을 보장한다.