가)표지

시스템프로그래밍(F049-1)

과제 4 레포트

라즈베리 파이 간 GPIO 통신 시스템 설계 및 구현

아주대학교 소프트웨어학과

202220775

박민정

제출일: 2025.06.28.

나) 주제 선정 및 요구사항 분석

해당 과제는 지하 터널 등과 같이 기존의 통신이 차단된 환경에서의 서비스를 가정하였다. 생존자가 키보드를 통해 구조 메시지를 입력하면, 이를 통신 시스템으로 송신한다. 구조본부에서는 이를 수신하여 해당 메시지를 모스부호로 변환하고 확인할 수 있도록 하는 비상 통신 시스템을 구현하는 것을 목표로 한다.

본 시스템에서는 송신 신호를 모스부호(Morse Code)로 변환하여 전송하는 방식을 도입하였다. 이는 디지털 신호의 ON/OFF(1/0)만으로 표현이 가능하여 GPIO 핀만으로 구현할 수 있다는 장점이 있으며, 복잡한 프로토콜 없이도 명확한 정보 전달이 가능하다. 또한, 외부 전파나 네트워크 인프라가 전혀 없는 상황에서도 안정적인 송신이 가능하며, 실제 비상 구조 상황에서 오랜 기간 검증된 통신 방식이라는 점에서 실용성이 높다. 무엇보다 라즈베리파이만으로 구현할 수 있다는 점에서, 생존자가 최소 장비만 가지고도 구조 요청을 보낼 수 있는 현실적인 비상 통신 수단이 될 수 있을 것이다.

요구사항

시스템의 주요 요구사항은 다음과 같다. 송신측에서는 유저가 키보드를 통해 입력한 메시지를 시스템이 실시간으로 읽고, 이를 GPIO 출력을 통해 신호로 전송해야 한다. 수신측에서는 이 신호를 정확하게 감지하고, 이를 모스부호로 변환하여 구조본부 화면에 출력해야 한다.

하드웨어적으로는 라즈베리파이의 GPIO핀을 이용하여 송신과 수신이 이루어진다. 이를 위해 GPIO 26번 핀을 송신(TX)용으로, GPIO 17번 핀을 수신(RX)용으로 사용하였다. 입력 장치는 키보드이며, 제어 및 데이터 처리는 모두 커널 레벨에서 동작하는 디바이스 드라이버를 통해 수행하였다.

이외의 요구사항으로는 시스템의 신뢰성과 자원 효율성이 고려된다. 신호의 정확도와 일관성을 높이기 위해, 수신 측에서는 디바운싱 기법이나 polling 방식, 혹은 인터럽트를 활용한 방식이 사용될 수 있다. 이와 같은 방법들은 노이즈에 의한 잘못된 신호 수신을 방지하고, CPU 자원을 보다 효율적으로 사용할 수 있게 해준다. 또한, 향후 시스템 확장을 고려해 송신 속도 설정, 오류 검출 기능 등의 추가 기능이 적용될 수 있으며, 드라이버 구조는 이러한 확장을 염두에 두고 설계되었다.

다) 통신 기법 설계

1. 물리연결 설계

시스템은 라즈베리파이의 GPIO 핀을 이용한 단방향 통신 구조로 설계되었고, 핀 구성

은 다음과 같다. 송신 측에서는 GPIO 26번 핀을 사용하며, 이 핀은 출력 방향(direction out) 으로 설정되어 알파벳을 모스부호로 변환한 후 신호를 외부로 출력한다. 수신 측에서는 GPIO 17번 핀을 사용하며, 이 핀은 입력 방향(direction in) 으로 설정되어 외부로부터 전송된 신호를 실시간으로 감지하고 이를 내부로 전달한다.

두 GPIO 핀은 물리적으로 직접 점퍼선으로 연결되며, 별도의 통신 장치 없이 전기 연결만으로 통신이 이루어진다. 신호 충돌을 방지하기 위해 송신 측은 value 값을 0 또는 1로 변경하며 신호를 발생시키고, 수신 측은 해당 전압 변화를 주기적으로 감지한다. 신호의 안정성을 위해 기본적으로 송신 측 value는 0으로 초기화되어 있다가, 데이터 전송 시점에만 1로 변경된다.

2. 통신방식 설계

통신은 핀 간의 단방향 시리얼 비트 전송 방식을 통해 수행된다. 송신 측에서 사용자가 입력한 알파벳 문자는 드라이버 내부에서 6비트의 이진 값으로 변환되고, 이 이진 데이터를 TX 핀을 통해 1비트씩 순차적으로 송신한다. 수신 측은 해당 비트를 RX 핀에서 일정 간격으로 읽어들이며, 이를 다시 6비트 값으로 복원하고, 최종적으로 알파벳 혹은 모스부호로 변환하여 사용자에게 출력한다.

본 설계는 알파벳을 직접 모스부호의 점/선 신호로 송신하지 않고, 이진값을 통하는 점이 특징이다. 이는 통신 도중 점/선 길이 차이를 해석하는 데 생길 수 있는 오차를 방지하고, 모든 문자를 일정한 규격으로 처리하여 오류 가능성을 낮추기 위함이다.

결과적으로 이 방식은 최소한의 하드웨어 자원으로도 통신을 가능하게 하며, 신호의 길이나 모양 대신 비트 단위의 정량적인 전송 방식을 사용함으로써 구현의 단순성과 신뢰성을 확보할 수 있게 된다.

라) SW 설계

1.소프트웨어 구조 및 동작 설계

시스템은 커널 레벨에서 구현된 문자 디바이스 드라이버를 중심으로 구성되며, file_operations 구조체의 .read, .write 인터페이스를 통해 사용자 공간과 통신한다. 송신 측에서는 write() 시스템 콜을 통해 입력된 문자열이 드라이버로 전달되고, 내부적으로 copy_from_user()를 통해 커널 공간으로 복사된 후 char_to_sixbit() 함수로 6비트 이진수로 변환된다. 이후 send_sixbit_serial() 함수에서 각 비트를 LSB부터 GPIO TX 핀으로 출력하게 되며, 이 과정은 일정한 시간 간격(msleep)을 두고 이루어진다.

수신 측에서는 read() 호출 시 gpio_read()가 실행되며, receive_sixbit_serial()을 통해

GPIO RX 핀에서 순차적으로 6비트를 수신한다. 수신된 값은 sixbit_to_morse()를 통해 대응되는 모스부호 문자열로 변환되며 사용자 공간으로 전달된다.

GPIO 제어에는 linux/gpio/consumer.h에 정의된 gpiod API를 활용하였다. 기존의 정수형 GPIO 번호 제어와 달리, 구조체 기반의 GPIO 디스크립터를 사용하는 이 방식은 gpiod_get_value(), gpiod_set_value() 등의 함수를 통해 보다 안전하고 커널 친화적인 방식으로 GPIO를 제어할 수 있게 해주며, 방향 설정과 상태 관리가 직관적인 것이 장점이다.

2.자원 관리 및 최적화 설계 방식

디바이스는 /dev/gpiomorse 문자 디바이스로 등록되어 사용자 공간에서 쉽게 접근 가능하며, 드라이버 초기화 과정에서 TX/RX 핀에 대한 gpiod 디스크립터를 한 번만 획득한 후 이를 전역으로 유지함으로써 불필요한 반복 호출을 줄이고 커널 자원 사용을 최소화하였다.

문자 단위가 아닌 6비트 단위 전송 구조를 통해 송신 시 전송 시간을 줄일 수 있었고, 수신은 비동기 입출력 기반(SIGIO, fasync)을 통해 효율적으로 처리할 수 있도록 설계되었다. fasync_struct와 kill_fasync()를 활용하여 송신 측에서 데이터 전송이 완료되었을 때 수신기 측에 SIGIO 시그널을 전달하고, 수신 응용프로그램이 이 시그널을 통해 read()를 호출하도록 구성함으로써, 자원 낭비 없이 실시간 수신이 가능하도록 하였다.

3.데이터 흐름 분석 (예: "Hello" 입력)

송신 측에서 사용자가 "Hello" 문자열을 입력하고 디바이스에 write()하면, 드라이버 내부의 gpio_write()가 실행된다. 문자열은 copy_from_user()를 통해 커널로 복사된 뒤, 문자 단위로 char_to_sixbit() 함수를 통해 6비트 이진수로 변환된다. 이진수는 send_sixbit_serial()에서 LSB부터 순차적으로 TX 핀(GPIO 17)에 출력되며, 각 비트 사이에 짧은 시간 지연(msleep)이 삽입된다.

수신 측에서는 GPIO 26번 핀(RX)을 통해 해당 비트를 읽고, receive_sixbit_serial()에서 이를 6비트로 조합한다. 이후 sixbit_to_morse()로 변환되어 모스부호 문자열(예: 'H' → "....")이 사용자에게 전달된다. SIGIO가 활성화되어 있는 경우, 송신 후 kill_fasync() 호출로 수신 측 응용프로그램에 시그널이 전송되며, 이를 통해 즉시 read()가 이루어져 데이터를 처리할 수 있게 된다.

이 구조는 각 문자를 독립적으로 처리하며 패킷 손실, 정렬 문제 등 복잡한 예외 처리를 요구하지 않고, 시그널 기반 비동기 처리 구조를 통해 polling 없이도 자원 효율적인 통신이 가능하다.

4.예외 상황

비영문자 입력: 알파벳 이외의 문자는 char_to_sixbit()에서 -1을 반환하여 송신에서 제외되며, pr_warn()을 통해 커널 로그에 경고가 남는다. 전체 시스템 동작은 중단되지 않고 이후 문자를 정상 처리한다.

TX/RX 핀 할당 실패: gpio_to_desc() 실패 시 드라이버는 -ENODEV를 반환하고, 모듈 로드를 중단하여 하드웨어 설정 오류를 사전에 방지한다.

수신 대기 중 데이터 부재: 수신 read() 시점에 수신할 데이터가 없으면 msleep()을 통해 대기를 수행한다. 비동기 구조가 활성화 돼있는 경우 kill_fasync()를 통해 필요한 시점에만 read()가 수행된다.

잘못된 핀 연결: 핀 방향이 명확히 지정되었으나, 하드웨어적으로 핀 연결이 반대로 되어 있거나 사용자가 sysfs를 통해 방향을 변경하려 할 경우 통신이 정상 동작하지 않을 수 있다.

마) SW 구현

1. Device Driver (device_driver.c)

이 드라이버는 라즈베리파이의 GPIO 핀을 문자 디바이스 인터페이스로 추상화하여, 사용자로부터 문자열을 입력받아 비트 단위로 송신하거나, 반대로 비트를 수신하여 문자로 복원하는 역할을 수행한다.

- 초기화(gpio_driver_init)

module_init() 매크로에 의해 등록된 이 함수는 모듈이 로드될 때 실행되며, 문자 디바이스 등록(alloc_chrdev_region), file_operations 등록(cdev_init, cdev_add), /dev 노드 생성(class_create, device_create), 그리고 GPIO 디스크립터 획득(gpio_to_desc) 및 방향 설정(gpiod_direction_output, gpiod_direction_input) 등의 초기 작업을 수행한다. 이때 BCM 번호는 GPIOCHIP_BASE와 합산하여 내부 GPIO 번호로 변환된다.

- 송신(gpio_write)

사용자가 /dev/gpiomorse에 문자열을 쓰면 해당 함수가 호출된다. 내부 동작은 다음과 같다. 먼저 copy_from_user()를 통해 사용자 공간에서 문자열을 커널로 복사하고, 각 문자를 char_to_sixbit() 함수를 통해 6비트 이진수로 변환한 뒤, send_sixbit_serial() 함수를 호출하여 각 문자의 6비트 값을 LSB부터 GPIO TX 핀에 1비트씩 순차적으로 출력한다. 이때 각 비트 간에는 msleep(BIT_DELAY_MS) 지연을 주어 수신 측과 동기화 한다.

- 수신(gpio_read)

사용자 공간에서 read()를 호출하면, 드라이버는 receive_sixbit_serial() 함수를 통해 GPIO RX 핀에서 6비트를 일정 간격(msleep)으로 샘플링하여 수신하고, 이를 숫자로 조합한다. 이후 sixbit_to_char() 또는 sixbit_to_morse() 함수를 통해 알파벳이나 모스부호 문자열로 복원하며, copy_to_user()를 통해 사용자 공간으로 전달한다.

- 비동기 인터페이스(gpio_fasync)

fasync_helper()를 통해 사용자 프로세스를 fasync_struct 큐에 등록해두고, 수신 이벤트 발생 시 kill_fasync()를 호출해 SIGIO 시그널을 전달할 수 있도록 한다. 이 구조를 통해 read()를 반복 호출하지 않고도 실시간으로 데이터를 받아볼 수 있도록 설계하였다.

- 정리(gpio_driver_exit)

모듈 제거 시 호출되는 함수로, 문자 디바이스 등록 해제, class 및 device 삭제, GPIO 관련 자원 정리 등을 수행한다.

2. 송신 측 (test_send.c)

사용자가 문자열을 입력하면, 이는 디바이스 드라이버의 gpio_write()로 전달되며 내부에서 인코딩 및 송신이 처리된다.

예를 들어 echo "Hello" > /dev/gpiomorse 명령을 수행하면, write() 시스템 콜을 통해 드라이버의 gpio_write()가 호출된다. 드라이버는 입력된 문자열을 char_to_sixbit() 함수로 6비트 이진수로 변환하고, 각 문자를 send_sixbit_serial()을 통해 1비트씩 TX 핀에 출력한다. 이때 각 비트 출력 간에는 BIT_DELAY_MS 만큼 지연을 주어 수신기와의 타이밍을 맞춘다.

알파벳 외의 문자(숫자, 특수기호, 공백 등)는 드라이버에서 무시되며, 필요 시 pr_warn() 로그를 통해 커널 메시지로 경고가 출력된다. 전체 송신 과정은 매우 단순하게 구성되어 있어, 복잡한 알고리즘 없이도 안정적인 송신이 가능하다.

3. 수신 측 (test_recv.c)

수신 프로그램은 /dev/gpiomorse 디바이스 파일을 열고 read()를 통해 데이터를 수신하는 구조다. 사용자는 프로그램을 실행하기만 하면 수신 대기 상태가 되며, 송신된 데이터가 도달하면 이를 콘솔에 출력한다.

open("/dev/gpiomorse", O_RDONLY)을 통해 디바이스 파일을 읽기 전용으로 열고, 내부적으로는 드라이버의 gpio_open()이 실행된다. 이후 무한 루프 내에서 read(fd, buf, sizeof(buf))를 반복 호출하며 수신을 대기한다. 이때 gpio_read()가 실행되어 RX 핀에서 6비트 시리얼 데이터를 읽고, 문자 혹은 모스부호 문자열로 복원하여 사용자 공간으로 전달된다.

수신된 데이터는 printf()로 출력되며, 만약 read()의 반환값이 0일 경우 데이터가 아직 수신되지 않은 것으로 간주하고, usleep(100000)으로 0.1초 대기 후 다시 read()를 시도하는 구조다. 수신된 문자열이 "exit"일 경우 프로그램은 루프를 종료하고 디바이스 파일을 닫는다.

바) 동작 검증 및 성능 분석

해당 시스템은 커널 레벨에서 구현된 디바이스 드라이버를 통해 문자 데이터를 GPIO 핀으로 송수신하는 구조로 설계되었다.

이론적으로, 문자 하나를 6비트로 표현하고 각 비트를 약 200ms 간격으로 전송할 경우, 문자당 약 1.2초가 소요되며 "Hello"와 같은 5글자 문자열을 송신하는 데는 약 6초가 걸리게 된다. 일반적인 통신 기준으로는 많이 느린 편이지만, 구조 요청과 같이 짧고 명확한 메시지를 전달하는 재난 상황을 가정한다면 충분히 수용 가능한 수준이라고 판단된다.

신뢰성 측면에서는, 신호가 단순한 high/low로만 구성되기 때문에 전송 방식 자체는 비교적 안정적이다. 각 문자가 독립적으로 처리되므로 중간에 일부 데이터 손실이 발생하더라도 나머지 메시지를 복원할 수 있다는 점은 큰 장점이다. 다만, 현재 오류 검출 기능이 포함되어 있지 않아, 노이즈나 타이밍 문제에 의한 오류 가능성은 여전히 존재한다. 이 부분은 향후 인터럽트 기반 수신 방식이나 재전송 메커니즘, 중복 전송 등의 방법으로 개선이 가능할 것으로 보인다.

자원 효율성 측면에서는 매우 간단하고 효율적인 구조를 가진다. 시스템은 GPIO 두 개만을 사용하며, 별도의 하드웨어 UART 회로나 고속 통신 회로 없이도 송수신이 가능하다. 또한 커널 드라이버 내에서도 msleep() 기반의 간단한 루프를 통해 비트 타이밍을 조절하고 있으며, GPIO 디스크립터도 한 번만 할당해 재사용함으로써 커널 자원 낭비를 최소화하고 있다.

실용성 측면에서도, 본 시스템은 재난 현장, 고립 지역, 통신 두절 상황 등 일반적인 통신 인프라가 작동하지 않는 환경에서 유용하게 사용될 수 있다. 단방향 통신이라는 제한은 존재하지만, 라즈베리파이만 있으면 구조 신호를 송신할 수 있다는 점에서 접근성과 응급 대응 측면에서의 실용성은 충분하다고 생각된다.

UART 프로토콜과 비교

본 시스템은 기존의 직렬 통신 프로토콜인 UART와는 명확하게 다른 구조와 목적을 갖고 있다. UART는 하드웨어 수준에서 TX/RX 라인을 이용한 동기화, Baud rate 기반 타이밍 조절, 패리티 및 정지 비트를 통한 오류 검출, FIFO 버퍼 활용 등 고속 및 고신뢰 통

신에 최적화된 방식이다. 반면 본 시스템은 문자 단위가 아닌 6비트 시리얼 비트 단위로 동작하며, 하드웨어 타이밍 동기화나 버퍼 없이 단순한 시간 간격(만으로 송수신을 진행한다.

UART는 고속, 안정성, 범용성 면에서 뛰어나고 다양한 시스템과 쉽게 연동되지만, 사전에 Baud rate 설정이나 드라이버 구성, 외부 회로 설계가 필요해 구현이 상대적으로 복잡하다. 반대로 본 시스템은 속도나 정확도는 다소 떨어질 수 있지만, 단순한 입력만으로 곧바로 신호를 송신할 수 있어 접근성과 구현 난이도 측면에서는 훨씬 단순하고 실용적이다.

마지막으로, UART는 인터럽트 기반의 신뢰성 높은 통신 구조를 제공하지만, 본 시스템 역시 커널 드라이버 내부에서 예외 문자 필터링, 핀 상태 제어, 비동기 SIGIO 이벤트 처리 등을 통해 기본적인 신뢰성을 확보하고 있다. 결론적으로, 본 시스템은 대용량·고속·범용 통신에는 적합하지 않지만, 최소한의 자원으로 명확한 신호를 안정적으로 전달해야 하는 상황에서는 오히려 더 적합한 구조를 가지고 있으며, UART와는 전혀 다른 철학과 목적을 가진 특화형 통신 방식이라고 볼 수 있다.

사) 결론 및 고찰

해당 과제에서는 기존의 통신 인프라가 전혀 작동하지 않는 재난 상황을 가정하여, 라즈베리파이의 GPIO 핀을 이용한 단방향 통신 기반 비상 구조 메시지 시스템을 구현하였다. 라즈베리파이와 같은 저사양 장치가 어떻게 의미 있는 서비스를 제공할 수 있을지, 왜 이런 장치 기반의 통신 수단이 필요한지를 다시 생각해보는 계기였다.

처음에는 버튼 입력 기반의 모스부호 시스템을 구현하려고 했으나, 하드웨어 연결이나 디바이스 드라이버 구현이 쉽지 않았고 키보드 입력 기반으로 방향을 전환하게 되었다. 처음 계획한 대로 구현하지는 못했지만, 오히려 그 과정에서 디바이스 드라이버의 구조와 커널 예외 처리에 대해 더 깊이 이해할 수 있었다. 덕분에 인터럽트 기반 통신이나 오류 처리 구조에 대한 관심도 더 커졌고, 앞으로의 프로젝트에서 한 단계 더 나아갈 수 있는 계기가 되었다.

또한, GPIO 제어를 위한 방식으로 처음에는 sysfs를 사용하는 것이 가장 직관적이고 간단하다고 생각했으나, 이번 프로젝트를 통해 gpiod 디스크립터 기반의 방식이 실제 커널 레벨에서 더 권장되고 안정적인 방법임을 체감할 수 있었다. gpio_to_desc()와 gpiod_set_value() 등의 함수를 통해 핀의 상태를 제어하는 구조는 예외 처리 측면에서도 유리했고, 향후 다른 프로젝트에서도 적극적으로 활용할 수 있을 것이다.

단순한 하드웨어 제어를 넘어서, 실제 비상 상황에서도 동작 가능한 시스템을 설계하고

구현해보며, IoT나 엣지 AI 같은 분야에 대한 관심도 더 깊어졌다. 이번 프로젝트는 단방향 통신이라는 한계를 가지긴 했지만, 실용성과 학습 측면 모두에서 매우 의미 있는 경험이었다.

아) 참고 문헌

1. The Linux Kernel,「GPIO Descriptor Consumer Interface」,
https://docs.kernel.org/driver-api/gpio/consumer.html

2. TechNote,「Development of Kernel Module - 02 Character Device Drivers」,
https://www.lazenca.net/display/TEC/02.Character+Device+Drivers

3. Arduino Docs,「Universal Asynchronous Receiver-Transmitter (UART)」,
https://docs.arduino.cc/learn/communication/uart/