# Assignment 4 : 라즈베리 파이 간 GPIO 통신 시스템 설계 및 구현

## - 전자 종이를 위한 이미지 데이터 전송 시스템 -

202126856 소프트웨어학과 전민규

## [목차]

# I. 서론

## 1. 과제 목표

본 과제의 최종 목표는 **두 대의 라즈베리 파이(Raspberry Pi) 시스템 간에 범용 입출력 포트(GPIO)만을 이용하여 안정적인 통신 시스템을 설계하고 구현하는 것**이다. 이 과정에서 리눅스 커널의 디바이스 드라이버, 사용자 공간 API, 그리고 이를 활용하는 응용 프로그램에 이르는 계층적 소프트웨어 구조를 직접 설계하며 시스템 프로그래밍에 대한 전반적인 이해를 목표로 한다.

## 2. 프로젝트 개요

본 프로젝트에서는 **'전자 종이(E-paper) 디스플레이를 위한 이미지 데이터 전송 시스템'**을 주제로 선정하였다. 송신 측 라즈베리 파이에서 JPEG, PNG 등 다양한 형식의 이미지 파일을 입력받아 1비트 흑백 데이터로 변환하고, 자체적으로 설계한 5-pin 시리얼 통신 프로토콜을 통해 수신 측 라즈베리 파이로 전송한다. 수신 측은 전송받은 데이터를 검증하고 다시 이미지 파일로 복원하는 역할을 수행한다.

전자 종이 디스플레이는 화면을 변환하지 않고 이미지를 유지할 때 전력이 거의 들지 않아 마트 가격표 등 다양한 곳에서 사용되고 있다. 이러한 점에서 GPIO를 이용한 임베디드 시스템은 간단한 하드웨어만으로도 사용 가능해 **전자 종이의 가격적 메리트를 극대화**한다.

전자 종이는 앞서 언급한 가격표처럼 **화면 정보를 바꾸는 빈도가 적은 경우** 주로 사용되므로, 데이터 전송 프로토콜도 이에 맞춰 설계하였다. 인터럽트를 이용해 불필요한 CPU 풀링을 없애 유휴 상태 전력 소모를 크게 줄였다. 또한 실시간 전송 속도보다 **전달되는 데이터의 신뢰성 및 안정성을 보장**하기 위해 CRC32(순환 중복 검사) 오류 검출 방식과 블록 단위의 ACK/NACK 핸드셰이크를 통한 자동 재전송 메커니즘을 구현하여 통신의 신뢰성을 확보하는 데 중점을 두었다.

# II. 통신 시스템 설계

## 1. 물리적 설계

안정적이고 명확한 제어를 위해 **총 5개의 GPIO 핀**을 사용하는 프로토콜을 설계하였다. 데이터 프레임의 시작과 끝 신호(START/STOP), 수신 확인을 위한 신호(ACK/NACK)를 위한 전용 라인을 두어 소프트웨어 복잡성을 줄이고 타이밍 문제 발생 가능성을 최소화하였다. 각 핀의 역할과 입출력 방향은 아래와 같다.

| 신호(Signal) | 송신 측(TX) | 수신 측(RX) | 설명 |
|---|---|---|---|
| **CLOCK** | OUTPUT | INPUT | 데이터 비트 전송의 동기화를 위한 클럭 신호 |
| **DATA** | OUTPUT | INPUT | 실제 데이터가 직렬로 전송되는 라인 |
| **START/STOP** | OUTPUT | INPUT | 데이터 블록(프레임)의 시작과 끝을 알리는 제어 신호 |
| **ACK** | INPUT | OUTPUT | 수신 측이 데이터 블록을 성공적으로 수신했음을 알리는 확인 신호 |
| **NACK** | INPUT | OUTPUT | 수신 측이 데이터 블록 수신 중 오류를 감지했음을 알리는 오류 신호 |

## 2. 데이터 전송 프로토콜 설계

데이터는 크게 **헤더(Header), 데이터(Data), CRC** 세 부분(블록)으로 나뉘어 전송된다. TCP의 구조를 일부 참조하여 신뢰성을 높였다.

<table>
<tr><td rowspan="5">데이터<br>전송<br>프로토콜</td><td rowspan="4">헤더<br>(10Bytes)</td><td>width (2Bytes)</td><td>이미지의 너비</td></tr>
<tr><td>height (2Bytes)</td><td>이미지의 높이</td></tr>
<tr><td>data_length<br>(4Bytes)</td><td>순수 이미지 데이터의 크기</td></tr>
<tr><td>header_checksum<br>(2Bytes)</td><td>헤더를 위한 Checksum</td></tr>
<tr><td colspan="2">데이터(최대 1KB)</td><td>실제 이미지 데이터로, 1KB를 초과하면 독립적인 청크(Chunk)로 나누어 전송</td></tr>
<tr><td colspan="3">CRC(4Bytes)</td><td>데이터 무결성 확인을 위한 CRC값<br>(crc32() 함수 사용)</td></tr>
</table>

또한 데이터 전송 시 흐름 제어를 위해 **Handshake 과정**을 진행한다. 각 블록에 대해 송신 측은 반드시 수신 측의 응답을 기다린다. 수신 측이 블록을 정상적으로 수신하고 체크섬 검증에 성공하면 ACK 신호를, 데이터 오류를 발견하면 NACK 신호를 보낸다. 송신측은 NACK 신호를 받거나 타임아웃 시간(5000ms)이 지나면 최대 3회까지 재전송을 시도한다. 이를 통해 일시적인 노이즈나 타이밍 오류를 자체적으로 해결한다.

# III. 소프트웨어 구조 및 설계

## 1. 전체 시스템 아키텍처

본 시스템은 명확한 역할 분리와 모듈화를 위해 커널 디바이스 드라이버, 사용자 API 라이브러리, 응용 프로그램의 3계층 구조로 설계했다.

디바이스 드라이버는 GPIO를 직접 제어한다. 복잡한 5핀 통신 프로토콜, 인터럽트 처리, 동기화, 오류 제어 로직을 모두 내부에서 진행하며, 사용자에게는 **/dev/epaper_tx, /dev/epaper_rx 라는 표준 파일 인터페이스(write, read 등)만을 노출, 복잡성을 완전히 숨긴다.**

API 라이브러리는 이미지 처리에 특화된 고수준 함수를 제공하는 중간 계층이다. jpg, png와 같은 이미지 파일을 흑백 전자 종이에 맞게 변환하고, 필요한 경우 디더링(Dithering) 등 복잡한 데이터 가공을 진행한다. 이를 통해 응용 프로그램 개발자가 프로토콜이나 드라이버의 상세 구현을 몰라도 쉽게 이미지 전송 기능을 사용할 수 있게 한다.

응용 프로그램은 사용자와 직접 상호작용하는 계층으로, 커맨드 라인에서 간단한 명령어를 통해 이미지 송수신을 실행할 수 있도록 도와준다.

## 2. 디바이스 드라이버 및 API 설계

우선 디바이스 드라이버는 전자 종이용 데이터 전송이라는 특수성을 고려, 송신용 드라이버(tx_driver.c)와 수신용 드라이버(rx_driver.c)를 분리하여 작성하였다. 또한 platform_driver 모델을 기반으로 구현하여, **하드웨어 정보를 소스 코드에서 분리하고 디바이스 트리(.dts)를 통해 동적으로 설정**할 수 있도록 설계했다.

송신용 드라이버는 write가 호출되면, 전달된 전체 이미지 데이터(헤더 포함)를 프로토콜에 맞게 헤더, 데이터 청크, CRC로 나누어 순차적으로 전송한다. wait_queue와 mutex를 이용해 ACK/NACK 응답을 동기적으로 대기하고, 동시 접근을 제어한다.

수신용 드라이버는 START/STOP 및 CLOCK에 대한 GPIO 인터럽트(IRQ)를 등록하여 이벤트를 비동기적으로 처리한다. start_stop_irq_handler는 프레임의 시작과 끝을, clock_irq_handler는 각 비트의 수신을 담당한다. 모든 데이터가 성공적으로 수신 및 검증되면 image_ready 플래그를 설정하고, read 함수에서 대기 중인 사용자 프로세스를 wake_up_interruptible 함수로 깨운다.

한편, 송신측 API 라이브러리는 이미지 경로와 변환 옵션을 가지는 함수 epaper_send_image_advanced를 제공한다. 내부 이미지 처리 후 write를 진행하는 역할을 수행한다. 수신측 API 라이브러리는 epaper_receive_image 함수를 제공, poll 시스템콜을 이용해 데이터가 준비될 때까지 효율적으로 대기한다. 데이터가 준비되면 구조체에 저장하며, epaper_save_image_pbm 함수를 통해 **.pbm 파일로의 저장을 지원한다.**

## 3. 데이터 흐름 분석(송수신 과정)

프로토콜의 특성을 고려, 과제에서 제시된 예시인 Hello 문자열 대신 image.png를 전달한다고 가정하였다.

| **송신 과정** | |
|---|---|
| **1** | epaper_send 프로그램이 epaper_send_image API 함수를 호출한다. |
| **2** | API는 stb_image.h를 이용해 image.png를 메모리에 로드하고<br>1비트 흑백 데이터로 변환한다. |
| **3** | 이미지 너비, 높이, 데이터 크기를 담은 헤더를 생성하고,<br>데이터 버퍼 앞에 결합한다. |
| **4** | API는 open("/dev/epaper_tx", …)로 드라이버를 열고,<br>생성된 버퍼를 write() 시스템 콜로 드라이버에 전달한다. |
| **5** | tx_driver의 tx_write 함수가 활성화된다. 함수는 데이터를 헤더, 데이터 청크들,<br>CRC로 나누어 send_data_block을 통해 순차적으로 전송한다. |
| **6** | **send_data_block은 send_byte, send_bit 함수를 호출하여**<br>**GPIO 핀으로 클럭과 데이터 신호를 출력한다.**<br>각 블록 전송 후, **wait_for_response를 통해 ACK 신호가 올 때까지 대기한다.** |

| 수신 과정 | |
|---|---|
| **1** | epaper_receive 프로그램이 epaper_receive_image API를 호출하고,<br>API는 read("/dev/epaper_rx", ...)를 호출하며<br>드라이버로부터 데이터가 준비되기를 기다린다. |
| **2** | 송신 측에서 START/STOP 핀의 신호를 HIGH로 바꾸면,<br>**rx_driver의 start_stop_irq_handler가 실행된다.** |
| **3** | 드라이버는 수신 상태(RX_STATE_HEADER)로 전환하고, **CLOCK 핀의 상승 엣지마다 clock_irq_handler를 통해 DATA 핀의 값을 읽어 비트를 수집하고 바이트로 조립한다.** |
| **4** | 헤더 수신이 완료되면, START/STOP 신호가 LOW가 되고, 드라이버는 헤더 체크섬을 검증한다. **검증 성공 시 ACK를 보내고, 데이터 수신 상태(RX_STATE_DATA)로 전환한다.** |
| **5** | 모든 데이터 청크와 CRC가 위와 같은 방식으로 수신 및 검증되면, **드라이버는 image_ready 플래그를 true로 설정하고 wake_up_interruptible을 호출한다.** |
| **6** | API의 read 함수는 깨어나 드라이버의 버퍼로부터 완성된 이미지 데이터를 읽어와 사용자 프로그램에 반환한다.. |

# IV. 소프트웨어 구현

## 1. 핵심 기능 구현(드라이버 및 API)

드라이버에서 가장 핵심적인 기능은 **동기화**이다. 송신 드라이버는 wait_event_timeout을 사용하여 ACK/NACK 인터럽트가 발생하거나 타임아웃이 될 때까지 프로세스를 효율적으로 재운다. 수신 드라이버는 wait_event_interruptible을 사용하여 이미지 수신이 완료될 때까지 read를 호출한 사용자 프로세스를 대기시킨다. 이는 while(1)과 같은 비효율적인 폴링을 방지하고 시스템 자원을 효율적으로 사용하게 한다.

수신 드라이버는 **비동기 수신**을 통해 CPU 자원을 효율적으로 사용한다. 수신 드라이버는 request_irq를 통해 START/STOP과 CLOCK 핀의 신호 변화를 하드웨어 인터럽트로 감지한다. 특히 start_stop_irq_handler는 IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING 옵션을 사용하여 상승 엣지와 하강 엣지 모두에서 호출되도록 하여, 하나의 인터럽트 핸들러로 프레임의 시작과 끝을 모두 처리하는 효율적인 구조를 가진다.

API는 부가 기능으로 **전자 종이에 맞는 이미지 변환 기능**을 제공한다. 먼저 stb_image.h 라이브러리를 활용해 이미지를 효율적으로 메모리에 로드/언로드한다. apply_dithering 함수는

단순 흑백 변환 시 발생할 수 있는 이미지 품질 저하를 막기 위해 플로이드-스타인버그(Floyd-Steinberg) 디더링 알고리즘을 구현했다. 이 알고리즘은 픽셀을 흑백으로 변환할 때 발생하는 오차를 확산시켜, 제한된 색상으로도 풍부한 Scale을 표현하는 듯한 효과를 낸다.

각 드라이버 및 API는 Makefile을 활용, ‘make’ 및 ‘make install’을 이용해 쉽게 설치 및 사용할 수 있도록 하였다.

## 2. 응용 소프트웨어 구현

사용자 편의성을 극대화하기 위해 getopt_long 라이브러리를 사용하여 유연한 커맨드라인 인터페이스를 구현했다. 이를 통해 사용자는 소스 코드 수정 없이 다양한 옵션을 조합하여 프로그램을 실행할 수 있다.

송신 측은 ‘./epaper_send /path/to/image.jpg’을 통해 전송을 수행할 수 있으며, -w(너비 리사이징), -h(높이 리사이징), -D(--dither) 등의 옵션을 제공한다. 수신 측은 ‘./epaper_receive’을 통해 수신된 이미지를 저장할 수 있으며, -o(파일명 지정), -f(포맷 지정) 등의 옵션을 제공한다.

# V. 동작 검증 및 분석

## 1. 이미지 전송 테스트 및 검증

드라이버 및 API를 'make clean' - 'make' - 'make install'을 통해 설치해준 후 응용 프로그램을 make 및 실행하였다. GPIO 핀은 송신에 13, 5, 6, 16, 12번 핀, 수신에 21, 19, 26, 25, 20번 핀을 사용하였으며, .dts 파일로 만들어 설정하였다.(순서대로 CLOCK, DATA, START/STOP, SCK, NACK에 대응하며, 드라이버 make install시 자동 적용되나 최초 1회 재부팅 필요)

송신용 이미지 원본(3.png)

수신 완료 이미지(received_image.pbm)

송신용 응용 프로그램 실행 결과

```
pi@raspberrypi:~/project/programs $ sudo ./epaper_send 3.png
Image loaded: 104x105, 4 channels
Converting to 1-bit monochrome (1365 bytes)...
Sending image: 104x105, 1365 bytes data
Total transmission: 1375 bytes (header + data)
Sending 1375 bytes to TX driver...
Successfully sent 1375 bytes to TX driver
Successfully sent image
Image sent successfully!
```

수신용 응용 프로그램 실행 결과

```
pi@raspberrypi:~/project/programs $ sudo ./epaper_receive
Waiting for image data...
PBM image saved to received_image.pbm
Image saved successfully: 104x105 pixels, 1365 bytes
pi@raspberrypi:~/project/programs $
```

송수신 통합 로그(sudo dmesg 결과)

```
[  122.293885] TX write: 1375 bytes
[  122.293904] Header: width=104, height=105, data_length=1365, checksum=0
[  122.293910] Calculated CRC32: 0x1f861ace
[  122.293913] First 16 bytes of data: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
[  122.293917] Transmission attempt 1/3
[  122.293926] RX: Start signal received, beginning reception, rx_state=0
[  122.293928] RX: Set data_ptr=00000000362a27d7 for state 0
[  122.302304] RX: Stop signal received, byte_count=10, data_ptr=000000003274f4c0, rx_state=0
[  122.302313] RX: Debug - image_buffer=0000000000000000, header.data_length=1365
[  122.302316] RX: Header received: width=104, height=105, data_length=1365, checksum=1574
[  122.302319] RX: Calculated checksum: 1574, received: 1574
[  122.302322] RX: Header OK, sending ACK, waiting for 1365 bytes of data
[  122.312328] TX: ACK interrupt received
[  122.317339] TX: ACK received
[  122.317342] Sending image data (1365 bytes)
[  122.317346] RX: Start signal received, beginning reception, rx_state=1
[  122.317349] RX: Set data_ptr=000000009f0b388a for state 1
[  122.666049] RX: Stop signal received, byte_count=1024, data_ptr=0000000094bbc6e3, rx_state=1
[  122.666057] RX: Debug - image_buffer=000000009f0b388a, header.data_length=1365
[  122.666060] RX: Data chunk received (1024 bytes), total: 1024/1365
[  122.676064] RX: Waiting for more data (1024/1365 bytes received)
[  122.676068] TX: ACK interrupt received
[  122.681077] TX: ACK received
[  122.681081] RX: Start signal received, beginning reception, rx_state=1
[  122.681084] RX: Set data_ptr=0000000094bbc6e3 for state 1
[  122.800805] RX: Stop signal received, byte_count=341, data_ptr=000000008521e8e9, rx_state=1
[  122.800812] RX: Debug - image_buffer=000000009f0b388a, header.data_length=1365
[  122.800815] RX: Data chunk received (341 bytes), total: 1365/1365
[  122.810820] RX: All data received (1365 bytes), waiting for CRC32
[  122.810823] TX: ACK interrupt received
[  122.815833] TX: ACK received
[  122.815836] Data sent successfully, now sending CRC32 (4 bytes): 0x1f861ace
[  122.815840] RX: Start signal received, beginning reception, rx_state=2
[  122.815843] RX: Set data_ptr=000000008521e8e9 for state 2
[  122.822190] RX: Stop signal received, byte_count=4, data_ptr=000000009c3ae613, rx_state=2
[  122.822194] RX: Debug - image_buffer=000000009f0b388a, header.data_length=1365
[  122.822198] RX: CRC32 received: 0x1f861ace, expected: 0x1f861ace
[  122.822201] RX: CRC32 OK, sending ACK, image ready
[  122.832211] TX: ACK interrupt received
[  122.837647] TX: ACK received
[  122.837660] Transmission successful on attempt 1
[  122.837666] Write operation completed successfully
```

실행 결과 정상적으로 전송되어 메시지가 출력되었다. 수신 측은 송신된 이미지와 동일한 크기(가로x세로)의 흑백 이미지를 전달 받았으며, 그 과정은 위 로그와 같다.

## 2. 성능 분석

tx_driver.c는 데이터 세팅 후 10μs, 클럭 HIGH 설정 후 20μs, LOW 설정 후 10μs를 대기하므로 **이론상 클럭 주파수는 1 / 40μs = 25,000Hz (25kHz)**이다. 이는 File-based I/O의 특성과 안정성을 고려한 수치로, 기존 표준 프로토콜보다는 전송에 더 많은 시간이 걸린다. 그러나 전자 종이 이미지 전송에 특성상 전송 시간은 크게 중요하지 않기 때문에, 여유로운 대기 시간을 설정하여 신뢰성을 높이는 데 집중하였다. 25kHz는 이론상 수십 KB 크기의 이미지를 수 초 내에 전송 가능하다.

위의 실제 데이터 전송 예시에서는 이론적 성능인 **25,000Hz / 8 = 3.125KB/s**의 약 81% 수준인 2.53KB/s를 달성하였다. 이는 Handshake 및 추가 데이터 전송에 따른 오버헤드로 인한 것이다. 반면 헤더 체크섬, 전체 데이터에 대한 CRC32, 그리고 블록 단위 ACK/NACK 및 자동 재전송(ARQ) 메커니즘은 이론상 99.999999% 이상의 무결성을 보장한다.

또한 수신 드라이버는 인터럽트를 기반으로 동작하기 때문에, 수신을 대기하는 동안 CPU 점유율이 0%에 가까운 모습을 보였다. 이는 설계대로 수신 드라이버가 효율적으로 CPU 자원을 사용함을 알 수 있다.

## 3. SPI 프로토콜과의 비교 분석

본 시스템을 SPI(Serial Peripheral Interface) 프로토콜과 비교해보자. 우선 설계의 측면에서, SPI는 전용 하드웨어 컨트롤러를 통해 고속으로 데이터를 전송하는 것을 목표로 한다. 반면 본 시스템은 SW적인 신호 제어를 통해 안정성 있고 신뢰도가 높은 이미지 데이터 전송을 목표로 한다.

따라서 전송 속도도 SPI가 수십MHz로 매우 빠르다. 반면 기본적으로 에러 검출이나 재전송을 지원하지 않기 때문에, 별도의 구현이 필요하다. 반면 본 시스템은 역할이 명확히 구분된 5개의 GPIO 핀과 헤더 및 Checksum, ACK/NACK 기능을 내장하고 있어 신뢰성이 매우 높으며, 상위 계층에서 이를 신경 쓸 필요가 없다.

결론적으로, 본 설계는 SPI의 고속 성능은 없지만, 프로토콜 수준에서 신뢰성 있는 흐름 제어와 오류 복구 메커니즘을 내장하여 소프트웨어적으로 안정성을 극대화한 전자 종이 데이터 전송용 맞춤형 솔루션이라는 점에서 차별화된다.

# VI. 결론

## 1. 프로젝트 요약 및 고찰

본 프로젝트를 통해 라즈베리 파이의 GPIO를 직접 제어하여 두 시스템 간에 이미지를 안정적으로 전송하는 통신 시스템을 성공적으로 설계하고 구현했다. 3계층(드라이버-API-응용 프로그램)으로 분리된 소프트웨어 아키텍처를 적용하여 코드의 재사용성과 유지보수성을 확보했고, 커널 드라이버 개발 과정에서 인터럽트, 동기화 큐, mutex 등 시스템 프로그래밍의 핵심적인 개념들을 깊이 있게 학습하고 적용할 수 있었다.

특히 본 설계는 서론에서 언급한 주제, '**전자 종이 디스플레이'의 특성**을 깊이 있게 반영했다. 첫째로, 전자 종이의 낮은 화면 변경 빈도와 저전력 특성에 주목하여, 인터럽트 기반의 비동기 수신 드라이버를 구현했다. 이를 통해 데이터 수신 대기 상태에서 불필요한 CPU 폴링을 제거함으로써 유휴 상태의 전력 소모를 최소화하는 설계 목표를 달성했다.

둘째, 가격표나 상품 정보 표시와 같이 데이터의 정확성이 중요한 전자 종이의 활용 사례를 고려하여, **실시간 전송 속도보다 데이터의 신뢰성을 우선**했다. 성능 분석에서 확인되었듯, 이론적 최대 속도와 실제 성능의 차이는 비효율이 아닌, 의도적으로 설계된 프로토콜 오버헤드의 결과이다. CRC32 오류 검출과 블록 단위의 ACK/NACK Handshake 및 자동 재전송 메커니즘은 통신 중 발생할 수 있는 오류를 효과적으로 복구하며 100%에 가까운 데이터 무결성을 보장했다. 이는 '안정성'을 최우선으로 했던 설계 철학이 성공적으로 구현되었음을 증명한다.

사실 원래의 프로토콜 설계는 지금과 크게 달랐다. 2개의 제어 핀과 3개의 데이터 핀을 사용하는 병렬 전송 방식으로 설계했으나, 기존 8개 핀으로 1Byte씩 전송하는 프로토콜들과 달리 3개의 핀을 사용하다 보니 바이트가 파편화되어 순서를 정렬하는 데에 어려움과 오버헤드가 있었고, 이러한 점이 안정성을 보장해야 하는 전자 종이 데이터 송신과 맞지 않다고 생각해 지금의 구조로 변경하였다.

물론 소프트웨어로 GPIO를 직접 제어하는 방식의 특성상, 하드웨어 SPI에 비해 전송 속도에는 명확한 한계가 존재한다. 하지만 본 프로젝트는 단순히 빠른 시스템을 만드는 것을 넘어, 요구사항(저전력, 고신뢰성)을 분석하여 그에 맞는 최적의 시스템을 설계하고 구현하는 전 과정을 직접 경험했다는 점에서 큰 의미가 있다. 저수준의 하드웨어 신호 제어부터 고수준의 응용 프로그램 인터페이스 설계까지, 시스템 전체를 아우르는 통합적인 시각을 기를 수 있었다.