V4L2

- Video for Linux -

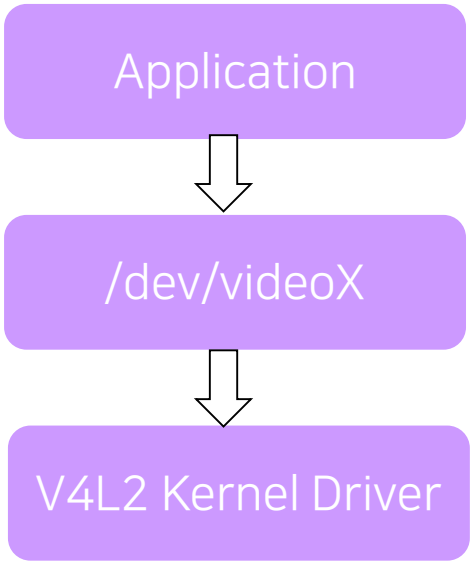
V4L2(Video For Linux Version 2)

What is V4L2?

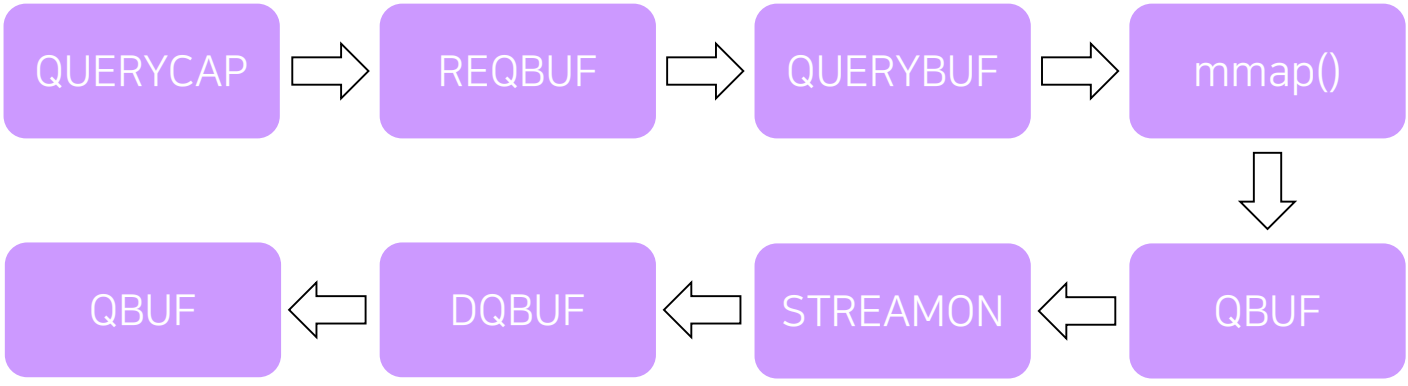
V4L2는 리눅스 시스템에서 카메라 입력을 받기 위한 표준 인터페이스이다. 최초 버전인 V4L 이후 1999년에 두번째 버전인 V4L2의 개발이 시작되었고, 초기 버전에서의 문제점들을 수정한 뒤 V4L2가 완성되었다.

V4l2 Driver

사용자 프로그램이 커널을 통해 시스템 하드웨어에 접근할 수 있도록 "dev" 디렉토리 내에 "video*"이라는 장치 파일이 생성된다. 사용자는 "/dev/video"를 통해 자료를 읽거나 기타 장치로 자료를 전송하는 것이 가능해진다. 일반적으로 Video Device는 Character Device로 생성된다. 이 Video Device는 일반 Character Device File과는 달리 read()나 write()로는 접근할 수 없으며 ioctl()을 이용해야 접근이 가능하다. 이때 Device에 접근하기 편하도록 커널에서 제공하는 것이 V4L2이다.



V4L은 Application에서 Video Device에 접근하여 특정 명령을 요청하면 V4L2 Kernel Driver가 요청 받은 명령을 동작한 후에 다시 Application에게 수행 결과를 반환하는 식으로 진행된다. 여기서 중요한 것은 Application에서 보내는 명령의 순서이다.



Application에서는 ioctl()을 이용해 명령들을 반드시 **순차적으로** 전송해야 한다. 여기서 순서가 바뀐다거나, 필요한 명령어가 빠지는 등의 상황이 생기면 명령을 수행하는 중에 오류가 발생할 수 있으므로 주의해야 한다.

이런 명령어들 중에서 QBUF, DQBUF는 반복적으로 수행해야 영상을 얻을 수 있고, 한번만 수행한다면 프레임을 1개만 얻을 수 있다.

Device Open

우리가 올바른 장치를 연결한다면, “video0”이 “/dev” 디렉토리에 나타날 것이다. 그럼 우리는 어떻게 장치를 열어서 사용할 수 있을까?

open() 함수와 사용해서 장치를 열을 수 있다.

```
int fd;
fd = open(“/dev/video0”, O_RDWR | O_NONBLOCK, 0);
if(fd == -1)
{
        perror(“Opening Video Device”);
        return 1;
}
```

여기서 O_NONBLOCK 옵션은 버퍼를 읽을 때 소프트웨어가 계속 차단되는 것을 방지한다.

VIDIOC_QUERYCAP

QUERYCAP(Query Capability)은 연결된 Device의 이름, 수행 가능한 동작 등 장치의 정보를 사용자 영역에 알려주는 역할을 한다.

여기서 우리는 기본적으로 capture가 가능한지 아닌지를 확인한다. V4L2는 일부 카메라를 지원하지 않기 때문에 이곳에서 오류를 발생시킬 것이다.

우리는 v4l2_capability 구조체와 VIDIOC_QUERYCAP을 사용할 것이다.

v4l2_capability 구조체의 선언이다.

```
struct v4l2_capability {
        __u8    driver[16];
        __u8    card[32];
        __u8    bus_info[32];
        __u32   version;
        __u32   capabilities;
        __u32   device_caps;
        __u32   reserved[3];
};
```

이런 것들을 통해 우리가 필요한 정보를 알아낼 수도 있다.

```
struct v4l2_capability caps = {0};
int fd;
fd = open("/dev/video0", O_RDWR | O_NONBLOCK, 0);
if(fd == -1)
{
        perror("Open Video Device");
        return 1;
}
if(-1 == ioctl(fd, VIDIOC_QUERYCAP, &caps))
{
        perror("Querying Capabilities");
        return 1;
}
printf("Driver Caps:₩n"
       "  Driver: %s₩n  Card: %s₩n  Bus: %s₩n  Version: %s₩n",
       caps.driver, caps.card, caps.bus_info, caps.version);
close(fd);
```

VIDIOC_S_FMT

V4L2는 웹캠이 지원하고 제공하는 이미지 형식과 colorspace를 쉽게 확인할 수 있는 인터페이스를 제공한다. v4l2_format 구조체는 이미지 포맷을 변경하는데 사용된다.

v4l2_format 구조체의 선언이다.

```
struct v4l2_format {
        __u32  type;
        union  {
            struct v4l2_pix_format          pix;
                                .
                                .
                                .
};
```

다른 구조체들이 궁금하다면 videodev2.h를 참고하길 바란다.

v4l2_pix_format 구조체의 선언이다.

```
struct v4l2_pix_format {
        __u32  width;
        __u32  height;
        __u32  pixelformat;
        __u32  field;
        __u32  bytesperline;
};
```

```
struct v4l2_format fmt = {0};
fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
fmt.fmt.pix.width = 640;
fmt.fmt.pix.height = 480;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_MJPEG;
fmt.fmt.pix.field = V4L2_FIELD_NONE;

if(-1 == ioctl(fd, VIDIOC_S_FMT, &룻))
{
        perror("Setting Pixel Format");
        return 1;
}
```

이미지 폭과 높이를 각각 640과 480으로 설정했다. Pixelformat은 카메라가 지원하는 형식을 확인해야 한다. 현재는 이미지 형식을 MJPEG로 설정하였다.

VIDIOC_REQBUFS

REQBUFS(Request Queue Buffer)는 Video Device에서 받아온 데이터를 저장하기 위한 Buffer를 할당하는 명령이다. 이곳에서도 구조체를 사용하게 되는데, 그 구조체가 바로 v4l2_requestbuffers이다. v4l2_requestbuffers는 디바이스 버퍼를 할당하는 데 사용된다. 우리는 VIDIOC_REQBUFS와 v4l2_requestbuffers를 함께 사용하게 될 것이다.

v4l2_requestbuffers 구조체의 선언이다.

```
struct v4l2_requestbuffers {
		__u32  count;
		__u32  type;
		__u32  memory;
		__u32  capabilities;
		__u32 reserved[1];
};
```

buffer의 크기나 타입, 방식 등을 설정하며 buffer를 할당한다.

```
struct v4l2_requestbuffers req = {0};
req.count = 1;
req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
req.memory = V4L2_MEMORY_MMAP;

If(-1 == ioctl(fd, VIDIOC_REQBUFS, &req))
{
		perror(“Requesting Buffer”);
		return 1;
}
```

VIDIOC_QUERYBUF

VIDIOC_QUERYBUF는 특정 buffer의 정보를 요청할 때 사용하는 명령이다. 이 명령은 buffer의 offset 정보를 얻어오게 되며, 이것은 mmap()의 인자로 사용된다. 이곳에서 우리는 v4l2_buffer 구조체를 사용하게 된다.

v4l2_buffer 구조체의 선언이다.

```
struct v4l2_buffer {
        __u32                   index;
        __u32                   type;
        __u32                    bytesused;
        __u32                   flags;
        __u32                   field;
        struct timeval          timestampt;
        struct v4l2_timecode    timecode;
        __u32                   sequence;
        __u32                   memory;
                                .
                                .
                                .
};
```

원하는 buffe의 index를 넘겨주고 그 buffer의 offset 정보를 가져온다.

```
struct v4l2_buffer buf = {0};
buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory = V4L2_MEMORY_MMAP;
buf.index  = buf_index;
if(-1 == ioctl(fd, VIDIOC_QUERYBUF, &buf))
{
        perror("Querying Buffer");
        return 1;
}
```

mmap()

QUERYBUF로 얻어온 offset 정보는 mmap()에서 사용된다. mmap()을 사용해 buffer에 메모리를 할당 받는다. mmap() 호출은 파일 기술자 fd가 가리키는 객체를 파일에서 offset 바이트 지점을 기준으로 len 바이트만큼 메모리에 맵핑하도록 커널에 요청한다.

```
void * mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

- addr:할당받기 원하는 메모리 주소. 보통 0을 사용한다.
- len: 맵핑시킬 메모리 영역의 길이
- prot: 메모리 보호 정책(맵핑할 파일 디스크립터와 속성이 같아야 함)
    - PROT_READ: 해당 페이지 읽기 가능
    - PROT_WRITE: 해당 페이지 쓰기 가능
    - PROT_EXEC: 해당 페이지 실행 가능
    - PROT_NONE: 해당 페이지 접근 불가
- flags
    - MAP_SHARED: 공유 메모리 맵 방식
    - MAP_PRIVATE: 복사 메모리 맵 방식
    - MAP_FIXED: 지정된 주소 이외의 다른 주소를 선택하지 않음
- fd: 파일 기술자
- offset: 맵핑 시 len의 시작점 지정

앞서 설명했듯이 mmap()은 QUERYBUF에서 얻어온 offset 정보를 인자로 삼아 이루어진다.

```
struct v4l2_buffer buf = {0};
buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory = V4L2_MEMORY_MMAP;
buf.index  = buf_index;
if(-1 == ioctl(fd, VIDIOC_QUERYBUF, &buf))
{
        perror("Querying Buffer");
        return 1;
}
buffer = mmap(NULL, buf.length, PROT_READ | PROT_WRITE,
            MAP_SHARED, fd, buf.m.offset);
```

이곳에서 맵핑되는 사용자 영역의 buffe는 Video Device에서 얻어온 영상 정보를 담고 있다.

VIDIOC_QBUF

VIDIOC_QBUF(Queue Buffer)는 Video Device에게 새로운 프레임을 요청하는 명령이다. 이곳이 직접적으로 영상 정보를 요청하는 부분이라고 할 수 있다.

```
struct v4l2_buffer buf;
buf.type        = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory      = V4L2_MEMORY_MMAP;
if(-1 == ioctl(fd, VIDIOC_QBUF, &buf))
{
        perror("Query Buffer");
        return 1;
}
```

VIDIOC_STREAMON

STREAMON(Streaming on)이란 Video Device의 Stream을 활성화시키는 명령어이다. Stream이란 일반적으로 데이터, 패킷, 비트 등의 일련의 연속성을 갖는 흐름을 의미한다. 이 말은 Video Device에서 Stream을 활성화시키면 연속적인 이미지 데이터를 얻어올 수 있다는 말이다. 이곳에서는 enum v4l2_buf_type을 사용한다.

```
enum v4l2_buf_type type {
        V4L2_BUF_TYPE_VIDEO_CAPTURE        = 1;
        V4L2_BUF_TYPE_VIDEO_OUTPUT         = 2;
        V4L2_BUF_TYPE_VIDEO_OVERLAY        = 3;
        V4L2_BUF_TYPE_VBI_CAPTURE          = 4;
        V4L2_BUF_TYPE_VBI_OUTPUT           = 5;

                                .
                                .
                                .
};
```

```
enum v4l2_buf_type type;

type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
if(-1 == ioctl(fd, VIDIOC)STREAMON, &type))
{
        perror("Strart Capture");
        return 1;
}
```

Capture Image

이제 남은 것은 캡처한 프레임을 버퍼에 저장하는 것 뿐이다. 방법은 입출력을 관리하고자 하는 파일의 그룹을 fd_set이라는 곳에 집어 넣고, 그 값이 변했는지를 확인하는 방식이다. 그러기 위해서는 select()라는 함수를 사용하게 된다.

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
                fd_set *exceptfds, struct timeval *timeout);
```

- nfds: 검사 대상이 되는 fd의 범위. 일반적으로 디스크립터는 생성될 때마다 값이 1씩 증가하기 때문에 가장 큰 파일 디스크립터 값이 1을 더해서 인자에 넣으면 된다.
- fd_set: 관리하는 파일의 지정 번호가 등록 되어 있는 비트 배열 구조체
  - readfds: 읽을 데이터가 있는지 검사하기 위한 파일 목록
  - writefds: 쓰여진 데이터가 있는지 검사하기 위한 파일 목록
  - exceptfds: 파일에 예외 사항들이 있는지 검사하기 위한 파일 목록
- timeout: 시간의 제한 값

fd_set 구조체의 선언이다.

```
typedef struct fd_set {
        u_int        fd_count;
        SOCKET    fd_array[FD_SETSIZE];
} fd_set;
```

fd_set 자료형 관련 함수

- FD_ZERO(fd_set *fdset); fd_set 초기화 함수
- FD_SET(int fd, fd_set *fdset); 해당 파일디스크립터 fd를 1로 set
- FD_CLR(int fd, fd_set *fdset); 해당 파일디스크립터 fd를 0으로 set
- FD_ISSET(int fd, fd_set *fdset); 해당 파일디스크립터 fd가 1인지 확인

```
fd_set fds;
struct timeval tv;
int r;

FD_ZERO(&fds);
FD_SET(fd, &fds);

tv..tv_sec = 2;

R = select(fd+1, &fds, NULL, NULL, &tv);
if(r == -1)
{
		perror("Waiting for Frame");
		return 1;
}
if(r == 0)
{
		perror("Timeout");
		return 1;
}
```

VIDIOC_DQBUF

VIDIOC_DQBUF(Dequeue Buffer)는 QBUF 명령으로 얻어진 index를 토대로 메모리 영역에서 실제 이미지 데이터를 가져오는 명령이다. Video Device에 요청해 얻어진 프레임 정보를 이용하여 이미지 데이터를 사용자 영역으로 넘겨주는 일을 한다.

```
struct v4l2_buffer buf;
buf.type            = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory      = V4L2_MEMORY_MMAP;
if(-1 == ioctl(fd, VIDIOC_DQBUF, &buf))
{
            perror("Retrieving Frame");
            return 1;
}
```

VIDIOC_STREAMOFF

STREAMOFF(Streaming off)란 STREAMON과는 반대로, Video Device의 Stream을 비활성화시키는 명령어이다.

```
enum v4l2_buf_type type;

type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
if(-1 == ioctl(fd, VIDIOC)STREAMON, &type))
{
            perror("Strart Capture");
            return 1;
}
```

munmap()

munmap()함수는 mmap()으로 할당된 메모리 영역을 해제하는 함수다.

int * munmap(void *addr, size_t len);

- addr: mmap에 의해 반환된 주소
- len: mmap에 의해 지정된 크기

```
for(int i = 0; i < nbuffers; ++i)
{
        if(-1 == munmap(buffers[i].start, buffers[i].length))
        {
                perror("munmap");
                return 1;
        }
}
free(buffers);
```

Device Close

우리는 맨 처음에 open() 함수를 통해 장치를 열 수 있었다. 모든 과정이 다 끝난 후, close()함수를 통해 장치를 닫게 된다.

```
if(-1 == close(fd))
{
        perror("Device Close");
        return 1;
}
fd = -1;
```

Processing Image

데이터를 받아서 파일로 만드는 부분이다. 파일을 저장한 후 write() 함수를 이용하여 파일을 쓴다.

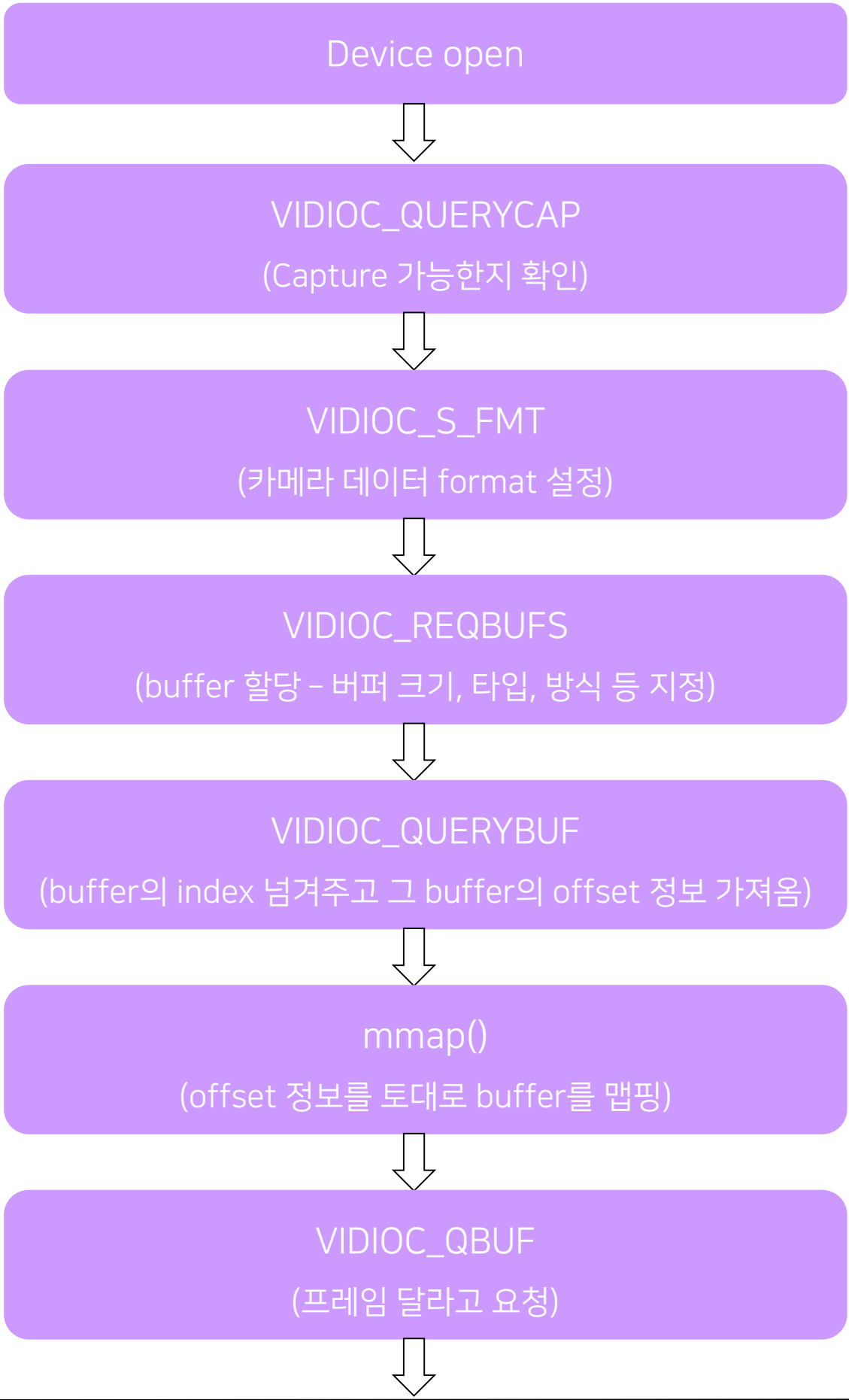
ssize_t write(int fd, const void *buf, size_t n);

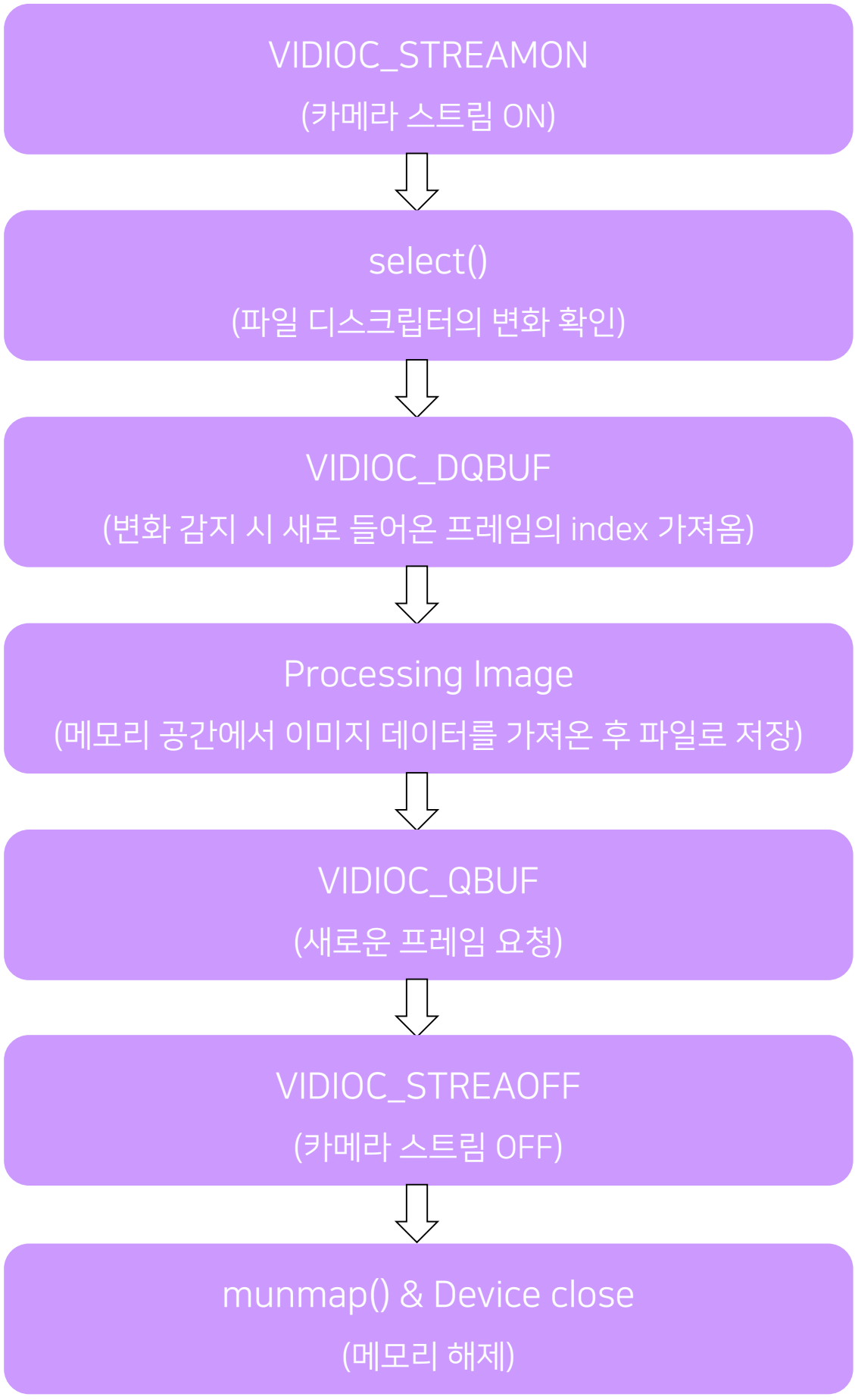
- fd: 파일 디스크립터
- buf: 파일에 쓰기를 할 내용을 담은 버퍼
- n: 쓰기를 할 바이트 개수

```
int num = 0;
int fp;
char outfile[15];

sprintf(outfile, “image_test%d.jpeg”, num++);
fp =  open(outfile, O_RDWR | O_CREAT);
write(fp, start, bytesused);
close(fp);
```

V4l2 Sequence




VIDIOC_STREAMON
(카메라 스트림 ON)
select()
(파일 디스크립터의 변화 확인)
VIDIOC_DQBUF
(변화 감지 시 새로 들어온 프레임의 index 가져옴)
Processing Image
(메모리 공간에서 이미지 데이터를 가져온 후 파일로 저장)
VIDIOC_QBUF
(새로운 프레임 요청)
VIDIOC_STREAOFF
(카메라 스트림 OFF)
munmap() & Device close
(메모리 해제)