리눅스
디바이스 응용 애플리케이션

Department of Electronics Eng.
Embedded Control System Lab.

1. 디바이스 응용 애플리케이션

1.1. Frame Buffer

프레임버퍼란 일반적으로 리눅스 시스템에서 그래픽 등을 표현할 수 있도록 LCD 컨트롤러가 제공해 주는 하드웨어를 말한다. 사용자는 제공된 버퍼에 영상 정보를 담을 수 있고, 담겨진 영상정보는 LCD를 통해서 표시된다. 일반적으로 리눅스에서는 해당 하드웨어의 장치 드라이버를 제공해주고 있다. 호스트 시스템의 경우는 대개 파일이름이 fb로 끝난다. 임베디드 디바이스 경우에는 프레임 버퍼를 기본으로 가지며, /dev/fb0에 데이터를 쓰는 것만으로 접근이 가능하다. SDL이라는 그래픽 라이브러리나 QT 등도 최종적으로는 Frame Buffer에 데이터를 기록한다. 다만, 해당 라이브러리에서 제공되는 편리한 인터페이스를 통해 화면에 표시해 준다는 것이 다를 뿐이다.

1.1.1. 프레임버퍼에 접근

리눅스에서는 프레임버퍼와 연결된 스페셜 파일을 통해서 프레임버퍼에 접근해야 한다. 아래 그림은 임베디드 보드의 스페셜 파일들이다. 파일들 중에서 프레임버퍼에 대한 스페셜(노드) 파일이 fb0이므로 응용 프로그램에서 접근하려면 /dev/fb0를 열어서 사용하면 된다.

```
# ls -al /dev/f*
```

※ 프레임버퍼 디렉토리 복사

```
<Host PC>
root@ubuntu:/work/achroimx6q# cp -a
/media/Achro-i.MX6Q-1/DVD-1_SRC/examples/linux/application/frame_buffer.tar.gz /root/temp
root@ubuntu:/work/achroimx6q# cd /root/temp
root@ubuntu:~/temp# tar xvf frame_buffer.tar.gz -C /work/achroimx6q/
root@ubuntu:~/temp# cd /work/achroimx6q/framebuffer/
```

1.1.2. 프레임버퍼 정보 가져오기

아래 프로그램은 프레임버퍼가 제공되는 정보를 시스템에서 가져오는 프로그램이다. 프로그램을 통해 사용자의 PC 혹은 Target 보드에서 사용되는 프레임버퍼의 정보들을 확인 할 수 있다.

① fbinfo.c 소스코드 작성

```
/* Filename : fbinfo.c */

 #include <stdio.h>
 #include <stdlib.h>
 #include <unistd.h>
 #include <fcntl.h>
 #include <sys/ioctl.h>
 #include <linux/fb.h>  // Frame Buffer API

 int main(int argc, char** argv)
 {
      int check;
      int frame_fd;
      struct fb_var_screeninfo st_fvs;  // 프레임버퍼의 가변 정보
      struct fb_fix_screeninfo st_ffs;  // 프레임버퍼의 고정 정보
      frame_fd = open("/dev/fb0",O_RDWR);
      if(frame_fd < 0)
      {
           perror("Frame Buffer Open Error!");
           exit(1);
      }

      check = ioctl(frame_fd, FBIOGET_VSCREENINFO,&st_fvs);
      if(check < 0)
      {
           perror("Get Information Error - VSCREENINFO!");
           exit(1);
      }

      check = ioctl(frame_fd,FBIOGET_FSCREENINFO,&st_ffs);
      if(check < 0)
      {
          perror("Get Information Error - FSCREENINFO!");
          exit(1);
      }

      system("clear");
      printf("==================₩n");
      printf("Frame Buffer Info₩n");
      printf("------------------₩n");
      printf("X - res    : %d₩n",st_fvs.xres);
      printf("Y - res    : %d₩n",st_fvs.yres);
      printf("X - v_res  : %d₩n",st_fvs.xres_virtual);
      printf("Y - v_res  : %d₩n",st_fvs.yres_virtual);
      printf("Bit/Pixel  : %d₩n",st_fvs.bits_per_pixel);
```

```
        printf("------------------₩n");
        printf("Buff Size : %d₩n",st_ffs.smem_len);
        printf("==================₩n");

        close(frame_fd);

        return 0;
    }
```

프레임버퍼의 상태가 들어가는 구조체를 만들고, 디바이스 드라이버를 통해서 프레임 버퍼의 정보를 가져와 앞에서 만들어 둔 구조체에 담는다. 만약, 정보를 가져오는 과정에서 문제가 발생하면 에러 처리 라인을 통과하고 프로그램이 종료된다. 좌표 X와 Y의 시작점과 마지막 점의 포지션과 1픽셀 당 비트수(Bits Per Pixel)의 값을 출력을 해주고 있다.

① 실행방법

프로그램을 컴파일을 하고, 임베디드 보드에서 실행한다.

■ 실행방법 1

호스트 PC에서 크로스 컴파일을 한 다음, cp 명령과 sync 명령을 차례로 수행해서 SD카드로 복사한다. 다음으로 임베디드 보드에 SD카드를 삽입하고 전원을 인가한 다음 터미널에서 실행한다.

<Host PC>

```
# arm-none-linux-gnueabi-gcc -o fbinfo fbinfo.c
# cp -a fbinfo /media/AchroIMX6Q_System/
# sync
```

■ 실행방법 2

호스트 PC에서 크로스 컴파일을 한 다음, nfs설정 디렉터리로 복사한다.

<Host PC>

```
root@ubuntu:/work/achroimx6q/framebuffer# arm-none-linux-gnueabi-gcc -o
fbinfo fbinfo.c
root@ubuntu:/work/achroimx6q/framebuffer# cp -a fbinfo /nfsroot/
root@ubuntu:/work/achroimx6q/framebuffer# sync
```

복사가 완료되면 임베디드 보드에서는 호스트의 nfs디렉터리를 마운트 하여 프로그램을 실행한다.

<Target Board>

```
[root@ACHRO ~]# mount -t nfs [호스트 PC의 IP]:/nfsroot /mnt/nfs -o rw,
rsize=1024,nolock
[root@ACHRO ~]# cd /mnt/nfs
[root@ACHRO nfs]# ./fbinfo
```

■ 실행결과

프로그램의 실행 결과로 보아, 현재 사용하는 타겟 보드의 프레임버퍼는 1024 * 600을 지원하고 있으며, 하나의 픽셀을 24비트로 구현하고 있음을 알 수 있다. 아래의 v_res는 가상 해상도를 의미한다. 기타 프레임 버퍼 구조체를 통하여 더 많은 정보를 얻을 수 있다. 일부 정보를 수정하여 적용시켜 볼 수 있지만 타겟 보드에서 사용하는 LCD의 지원해상도(width*height)와 Bits Per Pixel(depth)가 정해져 있기 때문에 변경이 용이하지 않거나 변경되지 않을 수도 있다.

1.1.3. 프레임버퍼 정보 변경하기

프레임버퍼의 정보를 변경하는 이유는 간단하다. 어디까지나 데이터 크기의 절약이 목적일 수도 있고, 정보의 올바른 출력을 위한 것일 수도 있다. 그 외에 다른 이유가 있을 수도 있다. 쉽게 말해서 8bpp의 특정 이미지를 16bpp로 설정된 LCD에 출력하면 정상적인 화면이 나올지를 추측해 보자. 한 픽셀을 출력하는데 16비트가 필요한 16bpp의 경우, 8bpp는 절반 밖에 미치지 못한다. 즉, 8bpp 이미지의 두 픽셀이 16bpp의 한 픽셀이 되므로 정상적인 영상이 그려지지는 않을 것이다. bpp는 같지만 이미지 크기가 512 * 300 일 때, 1024 * 600으로 출력하고 있다면, 이것도 위와 마찬가지로 원래 이미지 크기가 절반 밖에 되지 않기 때문에 정확한 영상이 출력되지 않을 수 있다. 해결할 수 있는 방법은 프레임버퍼 프로그램을 작성할 때, 320 * 240 크기에 맞춰서 코딩을 하게 되면 출력이 가능하다. 이미지를 제외한 나머지 영역을 0으로 설정해서 해당 프레임버퍼 크기와 맞춰주면 되기 때문이다.

아래 소스코드는 프레임버퍼의 정보를 바꾸어주는 프로그램이다. main 함수의 argument를 이용해서, 사용자가 입력하는 값으로 프레임버퍼를 설정해 주는 프로그램을 구현할 수도 있다. 앞서도 언급 했었지만 임베디드 장치의 경우 지원되는 해상도가 제한적이기 때문에 변경 시, LCD에 출력되지 않는 등의 문제가 생길 수 있다. 이 경우, 터미널에서 원래의 정보로 수정하거나 임베디드 시스템을 리부팅 하여야 한다.

① 소스 작성

```
/* FILENAME : fb_set.c  */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/fb.h>

typedef struct _user_fb {  // 사용자 지정 정보를 담는 구조체 함수 타입 선언
    int xres;
    int yres;
    int bpps;
} user_fb;

int main(int argc, char** argv)
{
    int frame_fd;
    int check;
    struct fb_var_screeninfo st_fvs;

    user_fb my_fb = {400,240,24};    // my_fb 생성과 동시에 초기화

    frame_fd = open("/dev/fb0",O_RDWR);
    if(frame_fd < 0)
    {
        perror("Frame Buffer Open Error₩n");
        exit(1);
    }

    if(check=ioctl(frame_fd, FBIOGET_VSCREENINFO,&st_fvs))
    {
        perror("GET Information Error - VSCREENINFO!");
        exit(1);
    }

    st_fvs.xres = my_fb.xres;  //사용자 지정값을 프레임버퍼 구조체 값으로 변경
    st_fvs.yres = my_fb.yres;
    st_fvs.bits_per_pixel = my_fb.bpps;

    if(check=ioctl(frame_fd, FBIOPUT_VSCREENINFO,&st_fvs))
    {
        perror("PUT Information Error - VSCREENINFO!");
        exit(1);
    }
```

```
        printf("==================₩n");
        printf("Frame Buffer Info₩n");
        printf("------------------₩n");
        printf("X - res    : %d₩n",st_fvs.xres);
        printf("Y - res    : %d₩n",st_fvs.yres);
        printf("X - v_res  : %d₩n",st_fvs.xres_virtual);
        printf("Y - v_res  : %d₩n",st_fvs.yres_virtual);
        printf("Bit/Pixel  : %d₩n",st_fvs.bits_per_pixel);
        printf("==================₩n₩n");

        close(frame_fd);

        return 0;
    }
```

my_fb의 값은 사용자가 변경할 값을 모아놓은 user_fb 구조체형으로 만든 것이다. 프레임버퍼의 설정 값이 들어있는 구조체의 멤버들 값으로 st_fvs를 바꾸는 내용이 39-41라인에 적혀 있다. 43번째 라인을 보면 사용자의 값으로 바뀐 st_fvs 구조체를 프레임버퍼에 적용시키고 있는 것을 확인할 수 있다.

실행 방법은 이전 예제와 동일하게, SD카드에 복사한 다음 실행하는 방법과 nfs를 이용해서 실행하는 방법이 있다.

② 컴파일

<Host PC>

```
root@ubuntu:/work/achroimx6q/framebuffer# arm-none-linux-gnueabi-gcc -o fbset fbset.c
root@ubuntu:/work/achroimx6q/framebuffer# cp -a fbset /nfsroot/
```

<Target Board>

```
[root@ACHRO ~]# mount -t nfs [호스트 PC의 IP]:/nfsroot /mnt/nfs -o rw,rsize=1024,nolock
[root@ACHRO ~]# cd /mnt/nfs
[root@ACHRO nfs]# ./fbset
```

■ 실행결과

위 프로그램은 이전의 정보 확인 프로그램에서 사용자가 몇 가지 정보를 고쳐 이를 적용한 다음, 다시 현재의 프레임버퍼 정보를 출력하는 예제이다. 이 예제를 통해서 프레임버퍼의 설정을 변경할 수 있지만 변경 가능 여부는 LCD의 특성에 따른다. 지원되지 않는 설정에는 화면이 출력되지 않는다.

1.1.4. Frame Buffer Example - Dot

위에서 프레임 버퍼의 정보를 가져와서 출력 하거나 설정을 변경하는 예제를 다루어 보았다. 이제 실제 프레임버퍼를 사용하는 예제를 다뤄 보고자 한다. 다음은 프레임버퍼에 데이터를 넣어서 변화를 확인하는 예제이다. 우선, 프레임버퍼(LCD)의 특정 위치에 점을 출력하는 예제를 작성해보자.

① 소스작성

```
/* FILENAME : fbpixel.c  */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <linux/fb.h>

typedef unsigned int U32;
typedef short U16;

U16 makepixel(U32  r, U32 g, U32 b)
{
    U16 x = (U16)(r >> 3);
    U16 y = (U16)(g >> 2);
    U16 z = (U16)(b >> 3);

    return (z|(x<<11)|(y<<5));
}

int main(int argc, char** argv)
{
    int check;
    int frame_fd;
    U16 pixel;  //  U16은 short 즉, 16비트.
    int offset;
    struct fb_var_screeninfo fvs;
```

```
    if((frame_fd = open("/dev/fb0",O_RDWR))<0) {
        perror("Frame Buffer Open Error!");
        exit(1);
    }

    if((check=ioctl(frame_fd,FBIOGET_VSCREENINFO,&fvs))<0) {
        perror("Get Information Error - VSCREENINFO!");
        exit(1);
    }

    // 시작위치 설정
    if(lseek(frame_fd, 0, SEEK_SET) < 0) {
        perror("LSeek Error.");
         exit(1);
    }

    offset = 120*fvs.xres*sizeof(pixel)+100*sizeof(pixel);
    if(lseek(frame_fd,offset,SEEK_SET) < 0) {   // 오프셋 위치로 이동
        perror("LSeek Error!");
        exit(1);
    }
    pixel = makepixel(255,0,0);
    write(frame_fd, &pixel, fvs.bits_per_pixel/(sizeof(pixel)));   // 화면에 출력

    offset = 120*fvs.xres*sizeof(pixel)+120*sizeof(pixel);
    if(lseek(frame_fd,offset,SEEK_SET) < 0) {
        perror("LSeek Error!");
        exit(1);
    }
    pixel = makepixel(0,255,0);
    write(frame_fd, &pixel, fvs.bits_per_pixel/(sizeof(pixel)));

    offset = 120*fvs.xres*sizeof(pixel)+140*sizeof(pixel);
    if(lseek(frame_fd,offset,SEEK_SET) < 0) {
        perror("LSeek Error!");
        exit(1);
    }
    pixel = makepixel(0,0,255);
    write(frame_fd, &pixel, fvs.bits_per_pixel/(sizeof(pixel)));

    close(frame_fd);
    return 0;
}
```

② 컴파일

<Host PC>

```
root@ubuntu:/work/achroimx6q/framebuffer# arm-none-linux-gnueabi-gcc -o fbpixel fbpixel.c
root@ubuntu:/work/achroimx6q/framebuffer# cp -a fbpixel /nfsroot/
```

<Target Board>

```
[root@ACHRO ~]# mount -t nfs [호스트 PC의 IP]:/nfsroot /mnt/nfs -o rw,rsize=1024,nolock
[root@ACHRO ~]# cd /mnt/nfs
[root@ACHRO nfs]# ./fbpixel
```

③ 실행결과

실행 방법은 이전 예제에서 몇 차례 언급했기 때문에 여기서는 생략한다. 위 소스코드는 프레임버퍼에 점을 찍는 예제로서, 시작 위치 0,0 부터 오프셋(Offset)까지의 거리를 만들고, 오프셋 위치에서 make_pixel 함수를 이용하여 사용자가 원하는 색상의 점(dot)을 LCD에 찍는 예제이다. 화면을 구분하기 위해서 좌표 개념을 도입하긴 했지만, 실제로 찍히는 메모리는 직선적인 구조이다. 그렇기 때문에 좌표 계산을 위해 lseek를 사용하고 있다.

1.1.5. LCD에 사각형 그리기

앞에서 점(dot)을 그리는 예제를 작성해 보았다. 이제 면을 그리는 예제를 작성

해 볼 차례다. 사각형의 경우에는 점을 연속해서 일정 영역 안에 그리면 된다. 즉, 시작 위치에서 끝 위치까지 점을 찍으면 되는 것이다.

① 소스작성

```
/* FILENAME : fbrect.c  */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <linux/fb.h>

typedef unsigned short int U32;
typedef short U16;
U16 makepixel(U32  r, U32 g, U32 b)
{
    U16 x = (U16)(r >> 3);
    U16 y = (U16)(g >> 2);
    U16 z = (U16)(b >> 3);

    return (z|(x<<11)|(y<<5));
}

int main(int argc, char** argv)
{
    int check;
    int frame_fd;
    U16 pixel;
    int offset;
    int posx1, posy1, posx2, posy2;
    int repx, repy;
    struct fb_var_screeninfo fvs;

    if((frame_fd = open("/dev/fb0",O_RDWR))<0) {
        perror("Frame Buffer Open Error!");
        exit(1);
    }

    if((check=ioctl(frame_fd,FBIOGET_VSCREENINFO,&fvs))<0) {
        perror("Get Information Error - VSCREENINFO!");
        exit(1);
    }
```

```
    if(lseek(frame_fd, 0, SEEK_SET) < 0) {   // Set Pixel Map
        perror("LSeek Error.");
        exit(1);
    }

    pixel = makepixel(0,0,255); // 색상 만들기
    posx1 = 100;  // 사각형의 크기를 결정한다.
    posx2 = 150;  // rect(100,150,120,170)을 구현한 것이다.
    posy1 = 120;
    posy2 = 170;

    for(repy=posy1; repy < posy2; repy++) {
        offset = repy * fvs.xres * (sizeof(pixel)) + posx1 * (sizeof(pixel));
        if(lseek(frame_fd, offset, SEEK_SET) < 0) {
            perror("LSeek Error!");
            exit(1);
        }
        for(repx = posx1; repx <= posx2; repx++)
            write(frame_fd, &pixel,(sizeof(pixel)));
    }
    close(frame_fd);
    return 0;
}
```

아래 소스코드는 사각형을 그리는 예제이다. 일반적으로 C프로그래밍에서 사각의 값을 구하는 것과 동일하며, 색상을 넣는 부분은 이전 소스코드와 동일하다. 이전에 작성했던 프로그램과 마찬가지로 32bpp 설정이 아니면 사각형의 정보를 기록하고 프로그램이 종료된다. 시작 위치에서 마지막 위치까지 반복 하면서 write 함수를 통해 LCD에 점(dot)을 찍어주고 있음을 알 수 있다.

② 컴파일

<Host PC>

```
root@ubuntu:/work/achroimx6q/framebuffer# arm-none-linux-gnueabi-gcc -o fbrect fbrect.c
root@ubuntu:/work/achroimx6q/framebuffer# cp -a fbrect /nfsroot/
```

<Target Board>

```
[root@ACHRO ~]# mount -t nfs [호스트 PC의 IP]:/nfsroot /mnt/nfs -o rw,
rsize=1024,nolock
[root@ACHRO ~]# cd /mnt/nfs
[root@ACHRO nfs]# ./fbrect
```

③ 실행 결과

타겟보드에 0,0,255(Red, Green, Blue)인 파란색 값의 사각형이 그려진 것을 확인 할 수 있다. 그러면 비압축 데이터(BMP)의 이미지를 화면에 출력할 수 있지 않을까? 비압축 데이터 중 BMP 파일의 경우는 헤더를 제외하면 순수한 RGB 값의 연속이기 때문에 위와 같은 방법으로 이미지 파일을 열어서 화면에 출력하는 것이 어렵지 않을 것이다.

1.1.6. I/O 방식과 memory mapping 방식 비교

① write()를 이용한 일반적인 랜덤 좌표 사각형 그리기

아래 소스코드는 위의 사각형 그리기 예제에서 난수를 발생시켜 시작점과 끝점을 랜덤하게 변형하여 출력하는 예제이다. 예제를 끝내기 위해서는 Ctrl+C 키를 입력하여 종료 할 수 있다.

```
/* FILENAME : fbranrect.c
   AUTHOR   : Hong, Sung-Hyun
   E-mail   : largest@huins.com */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <linux/fb.h>

typedef short U16;  // 자료형 선언
void swap(int *swapa, int *swapb);  // 박스 좌표 스왑
U16 random_pixel(void);   // 32비트 랜덤 색상값 생성

int main(int argc, char** argv) {
    int check, frame_fd, offset;
    U16 pixel;
    int posx1, posy1, posx2, posy2, repx, repy;
    struct fb_var_screeninfo fvs;

    if((frame_fd = open("/dev/fb0",O_RDWR))<0) {
        perror("Frame Buffer Open Error!");
```

```
        exit(1);
    }

    if((check=ioctl(frame_fd,FBIOGET_VSCREENINFO,&fvs))<0) {
        perror("Get Information Error - VSCREENINFO!");
        exit(1);
    }

    if(lseek(frame_fd, 0, SEEK_SET) < 0) {
        perror("LSeek Error.");
        exit(1);
    }

    while(1) {
        pixel = random_pixel(); // 랜덤 색상 값 구함
        posx1 = (int)((fvs.xres*1.0*rand())/(RAND_MAX+1.0));  // 랜덤좌표 x1
        posx2 = (int)((fvs.xres*1.0*rand())/(RAND_MAX+1.0));  // 랜덤좌표 x2
        posy1 = (int)((fvs.yres*1.0*rand())/(RAND_MAX+1.0));  // 랜덤좌표 y1
        posy2 = (int)((fvs.yres*1.0*rand())/(RAND_MAX+1.0));  // 랜덤좌표 y2
        swap(&posx1, &posx2); // 항목당 초기위치 설정
        swap(&posy1, &posy2); // 가독성상 표현, 조건문 사용이 우선되어야 함.

        msleep(500); // 0.5초 대기

        for(repy=posy1; repy < posy2; repy++) {
            offset = repy * fvs.xres * (sizeof(pixel)) + posx1 * (sizeof(pixel));
            if(lseek(frame_fd, offset, SEEK_SET) < 0) {
                perror("LSeek Error!");
                exit(1);
            }
            for(repx = posx1; repx <= posx2; repx++)
                write(frame_fd, &pixel,(sizeof(pixel)));
        } // End of For
    } // End of While
    close(frame_fd);

    return 0;
}

U16 random_pixel(void)
{   // 생상값 생성 후 반환
    return rand();
}

void swap(int *swapa, int *swapb) {
```

```
        int temp;
        if(*swapa > *swapb) { // 값 비교후 바꾸어야 된다면 스왑
            temp = *swapb;
            *swapb = *swapa;
            *swapa = temp;
        }
    }
```

<Host PC>

```
root@ubuntu:/work/achroimx6q/framebuffer# arm-none-linux-gnueabi-gcc -o fbranrect fbranrect.c
root@ubuntu:/work/achroimx6q/framebuffer# cp -a fbranrect /nfsroot/
```

<Target Board>

```
[root@ACHRO ~]# mount -t nfs [호스트 PC의 IP]:/nfsroot /mnt/nfs -o rw,rsize=1024,nolock
[root@ACHRO ~]# cd /mnt/nfs
[root@ACHRO nfs]# ./fbranrect
```

사각형 각각의 좌표를 난수를 이용하여 생성하고 있다. 그리고 swap 함수를 통해서 좌표 초기 값과 최종 값을 바꿔주고 있다. 이 조건에서 처음에 오는 초기 값이 더 작아야 한다. 이는 반복문 루틴을 최소화하기 위해서 이다. 스왑 함수를 구현한 부분에 조건문이 들어가 있을 것을 볼 수 있다. 가독성을 위해서 위와 같이 작성했지만 스왑 함수로 진입하기 이전에 조건 검사를 해야 한다. 스왑 함수로 진입하는 시점에 함수가 메모리에 로드 되고, 각 인자 값을 가지고 있는 변수를 위한 메모리 사용이 발생하기 때문이다. 스와핑 함수 내에서는 포인터가 참조하는 주소에 포인터가 가리키는 값(실제 데이터의 값이 있는 번지)를 넣어주어서 반환 값없이 데이터를 넘겨주고 있다. 지금까지와는 달리, 처음에 함수원형을 선언하고 가장 뒤부터 각 함수의 기능을 넣고 있다. 만약, 최초에 함수에 대한 선언이 없으면 위 프로그램은 에러를 발생시킬 것이고 컴파일조차 되지 않을 것이다.

② mmap을 이용한 일반적인 랜덤 좌표 사각형 그리기

일반적으로 mmap을 사용하는 이유는 성능 향상을 위해서 이다. lseek 등을 사용해서 프레임버퍼를 제어하면, 위치에 대해 lseek나 관련 포지션 설정할 때 마다 좌표를 잡기 위해서 시스템 콜을 발생시키고, 시스템 콜의 발생은 시스템의 처리 능력에 영향을 줄 수 있기 때문이다. mmap을 사용하게 되면 응용 프로그램의 메모리에 해당 디바이스를 맵핑 시켜주게 된다. 맵핑이 일어나면 포인터를 이용해 해당

디바이스를 읽거나 쓸 수 있다. mmap 함수 형식 및 함수 디스크립트는 아래와 같다.

```
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

mmap() : 응용 프로그램의 주소공간에 하드웨어 혹은 파일의 IO 영역을 대응

```
#include   <unstd.h>
#include   <sys/mman.h>
void *   mmap(void *start, size_t length, int prot, int flags, int fs, off_t offset);
```

start : 요청한 물리 주소 공간을 맵핑하려는 주소(Default = 0)
length : 맵핑 하려는 크기 (PAGE_SZIE 배수 단위)
prot : 메모리 보호 모드 - PROT_EXEC(실행), PROT_READ(읽기),
PROT_WRITE(쓰기) ,PROT_NONE(접근금지)
flags : 공유(MAP_SHARED) / 단독 설정 (MAP_PRIVATE)
fd : 파일 디스크립터
offset : 맵핑시키려는 물리 주소 (PAGE SIZE 단위로 지정)

성공시 맵핑된 대상 메모리의 주소를 반환하며, 실패하면 MAP_FAILED(-1) 반환

아래 소스코드는 이전의 랜덤 사각형을 그리는 예제를 mmap을 이용하도록 작성한 예제이다. 직접 실행해 보고 write를 이용한 방법과 mmap을 이용한 방법의 차이점을 확인해 보자.

```
/* FILENAME : fbmranrect.c  */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <unistd.h>
#include <linux/fb.h>

void swap(int *swapa, int *swapb);
short random_pixel(void);

int main(int argc, char** argv)   {
    int check, frame_fd;
    short pixel;
    int offset, posx1, posy1, posx2, posy2;
    int repx, repy, count = 1000;
    short* pfbdata;
    struct fb_var_screeninfo fvs;
```

```
    if((frame_fd = open("/dev/fb0",O_RDWR))<0) {
        perror("Frame Buffer Open Error!");
        exit(1);
    }

    if((check=ioctl(frame_fd,FBIOGET_VSCREENINFO,&fvs))<0) {
        perror("Get Information Error - VSCREENINFO!");
        exit(1);
    }

    pfbdata = (short *) mmap(0, fvs.xres*fvs.yres*(sizeof(pixel)), PROT_READ| ₩
              PROT_WRITE, MAP_SHARED, frame_fd, 0);  // 메모리 맵핑
    if((unsigned)pfbdata == (unsigned)-1) {
        perror("Error Mapping!₩n");
    }
    while(1<count--) {

        pixel = random_pixel();

        posx1 = (int)((fvs.xres*1.0*rand())/(RAND_MAX+1.0));
        posx2 = (int)((fvs.xres*1.0*rand())/(RAND_MAX+1.0));
        posy1 = (int)((fvs.yres*1.0*rand())/(RAND_MAX+1.0));
        posy2 = (int)((fvs.yres*1.0*rand())/(RAND_MAX+1.0));

        swap(&posx1, &posx2);
        swap(&posy1, &posy2);

        for(repy=posy1; repy <= posy2; repy++) {
            offset = repy * fvs.xres;

            for(repx=posx1;repx<=posx2;repx++)
               *(pfbdata+offset+repx) = pixel;
        } // End of For
    } // End of While

    munmap(pfbdata,fvs.xres*fvs.yres*(sizeof(pixel)));  // 맵핑된 메모리 해제
    close(frame_fd);
    return 0;
}

short random_pixel(void)  {
    return (int)(65536.0*rand()/(RAND_MAX+1.0));
}
```

```
void swap(int *swapa, int *swapb) {
    int temp;
    if(*swapa > *swapb) {
        temp = *swapb;
        *swapb = *swapa;
        *swapa = temp;
    }
}
```

<Host PC>

```
root@ubuntu:/work/achroimx6q/framebuffer# arm-none-linux-gnueabi-gcc -o fbmranrect fbmranrect.c
root@ubuntu:/work/achroimx6q/framebuffer# cp -a fbranrect /nfsroot/
```

<Target Board>

```
[root@ACHRO ~]# mount -t nfs [호스트 PC의 IP]:/nfsroot /mnt/nfs -o rw,rsize=1024,nolock
[root@ACHRO ~]# cd /mnt/nfs
[root@ACHRO nfs]# ./fbmranrect
```

③ 결과

임의 좌표를 생성하여 사각형을 그리기 때문에 화면에 다음과 같은 이미지가 표시될 것이다. 이미지 크기는 랜덤으로 결정되므로 정확한 비교는 되지 못한다. 하지만 응용 어플리케이션의 구동 속도에서 차이가 확연하게 나타날 것이다.