**Mmap**

It establishes a mapping between a range of addresses in a user process's address space and a portion of some "memory object" (typically a file, one of the special "devices" /dev/mem or /dev/kmem or some {memory-mapped} peripheral).  This allows the process to access a file at random byte offsets without using the seek {system call} or to access physical addresses or {kernel}'s {virtual address} space.  It can also be used as an alternative to writing a {device driver} since it is usually simpler to code and faster to use.

*#include<*[*sys/mman.h*](https://www.opengroup.org/onlinepubs/009695399/basedefs/sys/mman.h.html)*>  
void \*mmap(void \*addr, size\_t len, int prot, int flags, int fildes, off\_t off);*

pa=mmap(addr, len, prot, flags, fildes, off);

The *mmap*() function shall establish a mapping between the address space of the process at an address *pa* for *len* bytes to the memory object represented by the file descriptor *fildes* at offset *off* for *len* bytes. The value of *pa* is an implementation-defined function of the parameter *addr* and the values of *flags.*The parameter *prot* determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. The *prot* shall be either PROT\_NONE or the bitwise-inclusive OR of one or more of the other flags in the following table, defined in the [*<sys/mman.h>*](https://www.opengroup.org/onlinepubs/009695399/basedefs/sys/mman.h.html) header.

|  |  |
| --- | --- |
| **Symbolic Constant** | **Description** |
| PROT\_READ | Data can be read. |
| PROT\_WRITE | Data can be written. |
| PROT\_EXEC | Data can be executed. |
| PROT\_NONE | Data cannot be accessed. |

**Implementation example(Frame Buffer Interface)**

Drivers are a part of the kernel and hence run in kernel memory, whereas applications belong to user land and run in user memory. The only interface available to communicate between drivers and applications is the file operations

(the fops) such as open, read, write, and ioctl. Consider a simple write operation. The write call happens from the user process, with the data placed in a user buffer (allocated from user-space memory) and is passed over to the driver. The driver allocates a buffer in the kernel space and copies the user buffer to the kernel buffer using the *copy\_from\_user* kernel function and does the necessary action over the buffer. In the case of frame buffer

drivers, there is a need to copy/DMA it to actual frame buffer memory for output. If the application has to write over a specified offset then one has to call seek() followed by a write().

Now consider a graphics application. It has to write data all over the screen area. It might have to update one particular rectangle or sometimes the whole screen or sometimes just the blinking cursor. Each time performing seek(), followed by a write() is costly and time consuming. The fops interface provides the mmap() API for use in such applications. If a driver implements mmap() in its fops structure then the user application can directly obtain the user-space memory-mapped equivalent of the frame buffer hardware address. mmap() implementation is a must for the frame buffer class of drivers

All frame-buffer applications simply call open() of /dev/fb, issue necessary ioctl() to set the resolution, pixel width, refresh rate, and so on, and then finally call mmap(). The mmap implementation of the driver simply maps

the whole of the hardware video frame buffer. As a result the application gets a pointer to the frame buffer memory. Any changes done to this memory area are directly reflected on the display.

Example code

#include <unistd.h>

#include <stdio.h>

#include <fcntl.h>

#include <linux/fb.h>

#include <sys/mman.h>

main()

{

int fd = 0;

struct fb\_var\_screeninfo vinfo;

struct fb\_fix\_screeninfo finfo;

long int screensize = 0;

char \*map = 0;

int x = 0, y = 0;

long int location = 0;

fd = open("/dev/fb0", O\_RDWR);

if (!fd) {

printf("Error: cannot open framebuffer device.\n");

exit(-1);

}

if (ioctl(fd, FBIOGET\_FSCREENINFO, &finfo)) {

printf("Error reading fixed information.\n");

}

if (ioctl(fd, FBIOGET\_FSCREENINFO, &finfo)) {

printf("Error reading fixed information.\n");

//exit(2);

}

// printf("%dx%d, %dbpp\n", vinfo.xres, vinfo.yres, vinfo.bits\_per\_pixel);

screensize = vinfo.xres \* vinfo.yres \* vinfo.bits\_per\_pixel / 8;

// Mapping the device to memory

map = (char \*)mmap(0, screensize, PROT\_READ | PROT\_WRITE, MAP\_SHARED,fd, 0);

if ((int)map == -1)

printf("Error: failed to map framebuffer device to memory.\n");

printf("The framebuffer device was mapped to memory successfully.\n");

x = 100; y = 100; // Where we are going to put the pixel

// Figure out where in memory to put the pixel

for (y = 100; y < 300; y++)

for (x = 100; x < 300; x++) {

location = (x+vinfo.xoffset) \* (vinfo.bits\_per\_pixel/8) +

(y+vinfo.yoffset) \* finfo.line\_length;

if (vinfo.bits\_per\_pixel == 32) {

\*(map + location) = 100; // Some blue

\*(map + location + 1) = 15+(x-100)/2; // A little green

\*(map + location + 2) = 200-(y-100)/5; // A lot of red

\*(map + location + 3) = 0; // No transparency

} else { //assume 16bpp

int b = 10;

int g = (x-100)/6;

int r = 31-(y-100)/16;

unsigned short int t = r<<11 | g << 5 | b;

\*((unsigned short int\*)(map + location)) = t;

}

}

munmap(map, screensize);

close(fd);

return 0;}