**Memory Mapped IO/IO ports**

**Memory mapped IO**

I/O devices are mapped into the system memory map along with RAM and ROM.  To access a hardware device, simply read or write to those 'special' addresses using the normal memory access instructions.  The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device.  The disadvantage to this method is that the entire address bus must be fully decoded for every device.   For example, a machine with a 32-bit address bus would require logic gates to resolve the state of all 32 address lines to properly decode the specific address of any device.  This increases the cost of adding hardware to the machine.

**Port mapped IO**

I/O devices are mapped into a separate address space.  This is usually accomplished by having a different set of signal lines to indicate a memory access versus a port access.  The address lines are usually shared between the two address spaces, but less of them are used for accessing ports.  An example of this is the standard PC which uses 16 bits of port address space, but 32 bits of memory address space.  The advantage to this system is that less logic is needed to decode a discrete address and therefore less cost to add hardware devices to a machine.  On the older PC compatible machines, only 10 bits of address space were decoded for I/O ports and so there were only 1024 unique port locations; modern PC's decode all 16 address lines.  To read or write from a hardware device, special port I/O instructions are used.  From a software perspective, I feel that this is a slight disadvantage because more instructions are required to accomplish the same task.  For instance, if you wanted to test one bit on a memory mapped port, there is a single instruction to test a bit in memory, but for ports you must read the data into a register, then test the bit.

e.g

ARM does not support Port mapped IO. It supports memory mapped IO.The api’s used for memory mapped IO is

#define \_\_raw\_writeb(v,a) (\_\_chk\_io\_ptr(a), \*(volatile unsigned char \_\_force \*)(a) = (v))

#define \_\_raw\_writew(v,a) (\_\_chk\_io\_ptr(a), \*(volatile unsigned short \_\_force \*)(a) = (v))

#define \_\_raw\_writel(v,a) (\_\_chk\_io\_ptr(a), \*(volatile unsigned int \_\_force \*)(a) = (v))

#define \_\_raw\_readb(a) (\_\_chk\_io\_ptr(a), \*(volatile unsigned char \_\_force \*)(a))

#define \_\_raw\_readw(a) (\_\_chk\_io\_ptr(a), \*(volatile unsigned short \_\_force \*)(a))

#define \_\_raw\_readl(a) (\_\_chk\_io\_ptr(a), \*(volatile unsigned int \_\_force \*)(a))

Since kernel works on virtual memory Concept,the CPU memory has to be mapped with the system memory with some linux api’s.But before that the CPU peripheral physical address has to be defines as a platform device peripheral in arch/arm/mac-at91/devices.c file like

static struct atmel\_lcdfb\_info lcdc\_data;

static struct resource lcdc\_resources[] = {

[0] = {

.start = AT91SAM9263\_LCDC\_BASE,//Physical base address for LCD

.end = AT91SAM9263\_LCDC\_BASE + SZ\_4K - 1,//end address

.flags = IORESOURCE\_MEM,

},

[1] = {

.start = AT91SAM9263\_ID\_LCDC,

.end = AT91SAM9263\_ID\_LCDC,

.flags = IORESOURCE\_IRQ,

},

};

static struct platform\_device at91\_lcdc\_device = {

.name = "atmel\_lcdfb",

.id = 0,

.dev = {

.platform\_data = &lcdc\_data,

},

.resource = lcdc\_resources,

.num\_resources = ARRAY\_SIZE(lcdc\_resources),

};

void \_\_init at91\_add\_device\_lcdc(struct atmel\_lcdfb\_info \*data)

{

if (!data)

return;

lcdc\_data = \*data;

platform\_device\_register(&at91\_lcdc\_device);

}

Now in drivers/video/atmel\_fb.c

static struct platform\_driver atmel\_lcdfb\_driver = {

.remove = \_\_exit\_p(atmel\_lcdfb\_remove),

.suspend = atmel\_lcdfb\_suspend,

.resume = atmel\_lcdfb\_resume,

.driver = {

.name = "atmel\_lcdfb", //it should match with the above platform device structure in arch/arm/devices.c

.owner = THIS\_MODULE,

},

};

Then usually in probe function

*request\_mem\_region(info->fix.smem\_start, info->fix.smem\_len, pdev->name);*

*ioremap(info->fix.smem\_start, info->fix.smem\_len);*

I/O memory regions must be allocated prior to use. The interface for allocation of memory regions (defined in *<linux/ioport.h>*) is:

*struct resource \*request\_mem\_region(unsigned long start, unsigned long len, char \*name);*

This function allocates a memory region of len bytes, starting at start. If all goes well, a non-NULL pointer is returned; otherwise the return value is NULL. All I/O memory allocations are listed in */proc/iomem*. Memory regions should be freed when no longer needed:

*void release\_mem\_region(unsigned long start, unsigned long len);*

There is also an old function for checking I/O memory region availability:

*int check\_mem\_region(unsigned long start, unsigned long len);*

Allocation of I/O memory is not the only required step before that memory may be accessed. You must also ensure that this I/O memory has been made accessible to the kernel. Getting at I/O memory is not just a matter of dereferencing a pointer; on many systems, I/O memory is not directly accessible in this way at all. So a mapping must be set up first. This is the role of the *ioremap* function.

Once equipped with *ioremap* (and *iounmap*), a device driver can access any I/O memory address, whether or not it is directly mapped to virtual address space. Remember, though, that the addresses returned from *ioremap* should not be dereferenced directly; instead, accessor functions provided by the kernel should be used.

#include <asm/io.h>

void \*ioremap(unsigned long phys\_addr, unsigned long size);

void \*ioremap\_nocache(unsigned long phys\_addr, unsigned long size);

void iounmap(void \* addr);

For Port mapped IO,the api’s are

unsigned inb(unsigned port);

void outb(unsigned char byte, unsigned port);

Read or write byte ports (eight bits wide). The port argument is defined as unsigned long for some platforms and unsigned short for others. The return type of *inb* is also different across architectures. But again in linux kernel source these functions are redefined as readb/writeb only i.e

#define outb(v,p) \_\_raw\_writeb(v,\_\_io(p))

#define inb(p) ({ \_\_u8 \_\_v = \_\_raw\_readb(\_\_io(p)); \_\_v; })