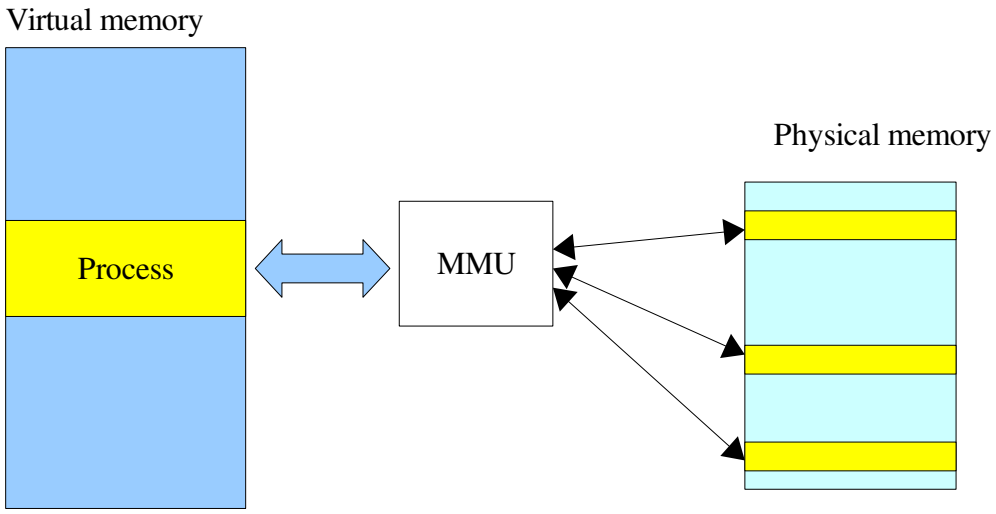
Direct Port I/O on the EP9302 from a "userland" process

Virtual memory vs physical memory

The I/O ports on ARM processors are memory mapped. That is, the port registers are hard wired to fixed physical memory locations. In order to interact with ports therefore, a program must be able to write to these particular memory locations. This is a problem for linux and indeed other operating systems that enable and make use of a Memory Management Unit (MMU).
Memory management units allow operating systems make more flexible use of physical memory. A program running (a process) on a system with memory management enabled sees a view of memory as shown below:



As far as the process is concerned, it may be operating in an address space ranging from say 1GB up to 1.1GB in the virtual memory space. It will therefore make use of addresses in the range 0x40000000 (1GB) up to 0x46000000. The hardware platform in question may actually only have 256MB of physical memory installed. The MMU will remap the processes address space to physical memory without the process being aware that this has happened. The process has in fact no way of directly accessing physical memory in this scheme - so how can we do direct port I/O. In order to see how we need first to look at a linux function called mmap (Memory Map).

The mmap was designed to allow processes point at a file from disk and map it to memory i.e. get the operating system to pretend that the file is actually part of memory (a bit like a paging file or virtual memory file). This can be useful as it allows programs use pointers to access data within the file rather than the more cumbersome file functions such as fread, fwrite, fseek and so on. When it comes to physical memory, mmap has an extra bit of functionality. It can be used to map specific parts of physical into the process's virtual memory space. Thus, an application can ask the OS (via mmap) to configure the mmu such that a particular address range in physical memory (e.g. the address range that contains port registers) into its virtual memory space. The function mmap returns an address which represents an area with virtual memory. Write and reads from/to this address will be routed through the MMU to the originally requested area in physical memory.

The call to mmap in the code below is as follows:

```
PortAddressSpace = (unsigned char *)mmap(0,getpagesize(), \
                                  PROT_READ | PROT_WRITE, MAP_SHARED,  \
                                  FileHandle,IOPORT_BASE);
```

The man (help) page for mmap lists its parameters as:
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
where:
start : This parameter can be used to specify where (in virtual memory) you would like to map or link to be made. A value of 0 implies that you are not too concerned and the kernel will choose for you.

length: MMU's don't map single bytes. This would be a complete waste of memory considering the overhead involved. In 32 bit systems, when a mapping is made, the MMU needs to relate a 32bit or 4 byte virtual address to a 32 bit physical address. Thus, a minimum of 8 bytes are required to relate virtual memory locations to physical ones. If single byte mapping were to be allowed then most of memory could be taken up with a mapping management table. To avoid this problem, MMU's usually operate with a 'granularity' called a 'page'. For many systems a page of memory is 4KB. Thus, the smallest amount of memory that can be mapped is 4K. Mappings also occur on 4KB boundaries so physical memory locations starting at addresses such as:
0000 0000, 0000 1000, 0000 2000, ... FFFFF000 i.e. the 3 least significant hex digits are zero. The actual page size may not be 4KB - you can find this out by calling getpagesize(). In the case of the call to mmap below, a single page is sufficient to contain all of the GPIO registers so the length of a single page is requested.

prot: This specifies the protection flags for the mapped memory (read, write etc)

fd: This specifies the file handle for the file that is to be mapped to memory. When a real file is involved, this is simply the value returned when the file open function on this file was called. If we want mmap to map physical memory we would appear to have a difficulty here. To get around this problem, we first open the pseudo file "/dev/mem" with a file open function. The file handle returned is then passed in as 'fd' in the function call. The pseudo file '/dev/mem' is backed by the kernel/device driver and represents the system's physical memory - elevated privilege is required to access this file of course.

offset: This is the address of the page you want remapped from physical memory (if that is indeed what you are doing). In the case of the EP9302 processor, the address at which the GPIO registers start is 0x80840000.

The program below (leds.c) can be used to turn on or off the green and red LED's on the EP9302 development board. The LED's are attached to the two least significant bits of GPIO port E. GPIO ports have data registers (DRA,DRB,...DRE etc) from which data can be read/written. They also have data direction registers (DDRA, DDRB, ... DDRE and so on). The data registers allow programmers configure individual port bits to be either inputs or outputs (a '1' in a DDR bit implies that the

corresponding DR bit is an output). The program was compiled with static linking using the arm-elf-gcc compiler.
To run type the following into a terminal window (on the 9302 board)

leds 0 0

This turns both LED's off
and

leds 1 1

to turn them both on (or any combination of 1's and 0's)

leds.c

```
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>
#include <fcntl.h>
#define IOPORT_BASE 0x80840000
#define PORTA_DR_OFFSET 0x0
#define PORTA_DDR_OFFSET 0x10
#define PORTB_DR_OFFSET 0x4
#define PORTB_DDR_OFFSET 0x14

#define PORTE_DR_OFFSET 0x20
#define PORTE_DDR_OFFSET 0x24


unsigned char *PortAddressSpace;
#define DDRA ( *(unsigned char *)(PortAddressSpace+PORTA_DDR_OFFSET))
#define DRA ( *(unsigned char *)(PortAddressSpace+PORTA_DR_OFFSET))
#define DDRB ( *(unsigned char *)(PortAddressSpace+PORTB_DDR_OFFSET))
#define DRB ( *(unsigned char *)(PortAddressSpace+PORTB_DR_OFFSET))
#define DDRE ( *(unsigned char *)(PortAddressSpace+PORTE_DDR_OFFSET))
#define DRE ( *(unsigned char *)(PortAddressSpace+PORTE_DR_OFFSET))

int main(int argc, char **argv) {
    int FileHandle;
    int GreenState, RedState;
    printf("\nNote: This program must be run with root privilege\n");
    if (argc !=3) {
            printf("usage: leds G R\n");
            printf("where G = 0 or 1.  1 turns Green LED on\n");
            printf("where R = 0 or 1.  1 turns RED LED on\n");
            return(-1);
    }

    GreenState = atoi(argv[1]);
    RedState = atoi(argv[2]);

    // open up /dev/mem
    FileHandle = open("/dev/mem",O_RDWR);
    printf("The pagesize for this processor is %d\n",getpagesize());
    PortAddressSpace = (unsigned char *)mmap(0,getpagesize(), \
                                PROT_READ | PROT_WRITE, MAP_SHARED,  \
                                FileHandle,IOPORT_BASE);
```

```
    if (PortAddressSpace == MAP_FAILED) {
        printf("\nUnable to map memory space\n");
        return (-2);
    }

    DDRE = 0xff;
    DDRB |= 0xff;
    if (GreenState)
        DRE |= 0x1;
    else
        DRE &= 0xfe;
    if (RedState)
        DRE |= 0x2;
    else
        DRE &=0xfd;

    munmap(PortAddressSpace,getpagesize());
    close(FileHandle);
    return(0);
}
```