



LDD3 Ch. 9

Communicating With Hardware

ECEN 427

BYU Electrical & Computer
Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

Allocating device memory

“As you might expect, you should not go off and start pounding on I/O ports (or memory) without first ensuring that you have exclusive access to those ports. The kernel provides a registration interface that allows your driver to claim the ports/memory it needs.”

```
#include <linux/ioport.h>
```

```
struct resource *request_mem_region(unsigned long start, unsigned long  
len, char *name);
```

```
void release_mem_region(unsigned long start, unsigned long n);
```

This is a physical address. We will talk about how to determine the physical address of our devices next lecture



This isn't strictly necessary, but it should be done to “play nice” with other drivers.

Create a Virtual Address Mapping

“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.

The functions are called according to the following definition:

```
#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);
```

Does the behavior change?

```
void foo() {
    int * p = ...;
    p[0] = 2;
    p[1] = 1;
    printf("p0 is %d\n", p[0]);
}
```

Offset	Register	Desc
0x00	Config	Config options
0x04	Reset	Write 1 to reset all config options

```
void foo() {
    int * p = ...;
    p[0] = 2;
    printf("p0 is %d\n", p[0]);
    p[1] = 1;
}
```

```
void foo() {
    int * p = ...;
    p[1] = 1;
    p[0] = 2;
    printf("p0 is %d\n", p[0]);
}
```

Memory vs Registers

What is the difference between read/writing device registers and memory?

What issues could arise?

https://en.wikipedia.org/wiki/Memory_ordering

I/O vs. Normal Memory

“The main difference between I/O registers and RAM is that I/O operations have side effects, while memory operations have none: the only effect of a memory write is storing a value to a location, and a memory read returns the last value written there.

Because memory access speed is so critical to CPU performance, the no-side-effects case has been optimized in several ways: values are cached and read/write instructions are reordered.”

“The compiler can cache data values into CPU registers without writing them to memory, and even if it stores them, both write and read operations can operate on cache memory without ever reaching physical RAM. Reordering can also happen both at the compiler level and at the hardware level: often a sequence of instructions can be executed more quickly if it is run in an order different from that which appears in the program text, for example, to prevent interlocks in the RISC pipeline. On CISC processors, operations that take a significant amount of time can be executed concurrently with other, quicker ones.”

Example Code

```
void foo() {  
    int * p = ...;  
    p[0] = 2;  
    p[1] = 1;  
    printf("p0 is %d\n", p[0]);  
}
```

Offset	Register	Desc
0x00	Config	Config options
0x04	Reset	Write 1 to reset all config options

What might this code do?

Memory Operation Reordering

These optimizations are transparent and benign when applied to conventional memory (at least on uniprocessor systems), but they can be fatal to correct I/O operations, because they interfere with those “side effects” that are the main reason why a driver accesses I/O registers.

The processor cannot anticipate a situation in which some other process (running on a separate processor, or something happening inside an I/O controller) depends on the order of memory access. The compiler or the CPU may just try to outsmart you and reorder the operations you request; the result can be strange errors that are very difficult to debug. Therefore, a driver must ensure that no caching is performed and no read or write reordering takes place when accessing registers.

The problem with hardware caching is the easiest to face: the underlying hardware is already configured (either automatically or by Linux initialization code) to disable any hardware cache when accessing I/O regions (whether they are memory or port regions).

“The solution to compiler optimization and hardware reordering is to place a memory barrier between operations that must be visible to the hardware (or to another processor) in a particular order. Linux provides four macros to cover all possible ordering needs:”

```
void rmb(void);  
void read_barrier_depends(void);  
void wmb(void);  
void mb(void);
```

These functions insert hardware memory barriers in the compiled instruction flow; their actual instantiation is platform dependent. An *rmb* (read memory barrier) guarantees that any reads appearing before the barrier are completed prior to the execution of any subsequent read. *wmb* guarantees ordering in write operations, and the *mb* instruction guarantees both. Each of these functions is a superset of *barrier*.

Reading/Writing Device Memory

Remember, though, that the addresses returned from `ioremap` should not be dereferenced directly; instead, accessor functions provided by the kernel should be used.

On some platforms, you may get away with using the return value from `ioremap` as a pointer. Such use is not portable, and, increasingly, the kernel developers have been working to eliminate any such use. The proper way of getting at I/O memory is via a set of functions (defined via `<asm/io.h>`) provided for that purpose.

To read from I/O memory, use one of the following:

```
unsigned int ioread8(void *addr);  
unsigned int ioread16(void *addr);  
unsigned int ioread32(void *addr);
```

```
void iowrite8(u8 value, void *addr);  
void iowrite16(u16 value, void *addr);  
void iowrite32(u32 value, void *addr);
```

The `ioreadX/iowriteX` enforce ordering against other `ioreadX/iowriteX` operations
(but not necessarily against normal memory)