Linux Device Drivers

Dr. Wolfgang Koch

Friedrich Schiller University Jena

Department of Mathematics and Computer Science

Jena, Germany

wolfgang.koch@uni-jena.de

Linux Device Drivers

1. Introduction
2. Kernel Modules
3. Char Drivers
4. Advanced Char Drivers
5. Interrupts

5. Interrupts

- Blocking I/O
- poll( )
- Asynchronous Notification
- Hardware Management
- Interrupts
- The Bottom Half

Blocking I/O

problem with `read()`:
there are no data **yet** but we're not at end-of-file
(the hardware will deliver data soon)
– similar problems with `write()` or `open()`

answer:
go to sleep, waiting for data
(another solution: asynchronous I/O)

- put a process (application + driver routine) to sleep
- wake it up later on
- application: ask whether an I/O operation will block
- notify the application asynchronously

Blocking I/O

- put a process to sleep
  (we cannot sleep while holding a spinlock
  or in interrupt context)

basic data type: wait queue (`wait_queue_head_t`)

declaration and initialization:

```
wait_queue_head_t  queue ;
init_waitqueue_head (&queue);
```

or if declared as global (statically)

```
DECLARE_WAIT_QUEUE_HEAD (queue);
```

(one step, at compile time)

Blocking I/O

- put a process to sleep
  old, strongly deprecated method :

```
sleep_on(wait_queue_head_t *);
interruptible_sleep_on(wait_queue_head_t *);
```

the latter is interruptible by signals and almost always used:

```
while (! condition){
   ...    // gap
   interruptible_sleep_on(&wq);
}
```

chance of race condition: if a wakeup arrives in the gap,
it is missed and the process may sleep forever

Blocking I/O

- put a process to sleep
  modern way (avoids race conditions)

```
void wait_event(wq, condition);
int  wait_event_interruptible(wq, condition);
long wait_event_timeout(wq, condition,
                            n_jiffies);
```

use of `wait_event_interruptible()`
– it returns `-ERESTARTSYS` if it was interrupted by a signal:

```
if (wait_event_interruptible(wq, condition))
          return -ERESTARTSYS;
```

then upper layers of the kernel will handle the event

Blocking I/O

- wake up a process (i.e. all processes waiting on the queue):

```
wake_up(wait_queue_head_t *);
wake_up_interruptible(wait_queue_head_t *);
```

the latter wakes up only processes that are in interruptible sleeps
you may use `wake_up()` in both cases, but using
`wake_up_interruptible()` preserves consistency

the wake up often is done in an interrupt handler,
once new data has arrived

Blocking I/O

- put a process to sleep

`wait_event_interruptible()` etc. includes:
– sets the state of the process to `TASK_INTERRUPTIBLE` (it is in an interruptible sleep)
– the task is added to the wait queue
– `schedule` is called, it relinquishes the processor (context switch)

- wake up a process:

`schedule` returns only when somebody else calls `wake_up()` etc. – it sets the state of the process to `TASK_RUNNING` and removes the entry from the wait queue

Blocking I/O

we demonstrate blocking I/O (blocking `read()`) with the circular buffer (the version with one shared buffer on several opens)

we add global:

```
#include <linux/sched.h>
DECLARE_WAIT_QUEUE_HEAD (wq);
```

we add at the beginning of `read()`:

```
if (wait_event_interruptible(wq, (ir!=iw)))
        return -ERESTARTSYS;
```

we add at the end of `write()`:

```
if (i>0) wake_up_interruptible(&wq);
```


Nonblocking Operations

there is a flag: `O_NONBLOCK`, which an application can set when opening a file/device:

```
fd=open(DEV_NAME, O_RDWR | O_NONBLOCK);
```

the driver can access the flags in `filp->f_flags`:

```
if ((ir == iw)     // buffer empty
    && (filp->f_flags & O_NONBLOCK))
                      return -EAGAIN;

if (wait_event_interruptible(wq, (ir!=iw)))
         return -ERESTARTSYS;
```


Blocking I/O

blocking may be useful with other functions too, as `write()` or `open()`

`wake_up()` etc. awakens every process in the wait queue – if we use blocking on both `read()`and `write()`, we have to use two different wait queues to wake up the right method

functions that can block (and functions calling functions which may block) must be re-entrant

the nonblocking flag only has an effect with `read()`, `write()` and `open()`

poll( )

using blocking I/O an application may want to know in advance whether an I/O operation will block (to avoid it)
– two very similar functions: `poll()` and `select()`
(`poll()` – System V, `select()` – BSD Unix)

`poll()` can simultaneously poll several data streams (and wait for the first occurrence of new data)

user space:

```
#include <sys/poll.h>
int poll(struct pollfd *, uint n, int timeout);
```


poll( )

user space:

```
#include <sys/poll.h>
int poll(struct pollfd *, uint n, int timeout);
```

`poll()` takes an array of n structures (for n streams) of type:

```
struct pollfd{
  int   fd;         /* file descriptor  */
  short events;     /* requested events */
  short revents;    /* returned events  */
}
```

and a timeout in milliseconds in case of waiting
(`0` – no wait, negative – infinite timeout)

poll( )

```
struct pollfd{
  int   fd;         /* file descriptor  */
  short events;     /* requested events */
  short revents;    /* returned events  */
}
```

the following bits in the event masks are defined in `<sys/poll.h>`

```
#define POLLIN   0x0001   /* there is data to read */
#define POLLPRI  0x0002   /* there is urgent data to
                             read */
#define POLLOUT  0x0004   /* writing now will not
                             block */
#define POLLERR  0x0008   /* Error condition */
#define POLLHUP  0x0010   /* Hung up – EOF   */
```


poll( )

we use `poll()` in the following way:

```
#include <sys/poll.h>
struct   pollfd  pfd;
...

 pfd.fd=fd; pfd.events=0x05;  // POLLIN | POLLOUT

  int k = poll(&pfd, 1, 0);

  if(k<0) perror(" poll() ");
  printf(" poll %2d, revents: %02x \n",
            k,pfd.revents);
```

poll( )

kernel space prototype:

```
#include <linux/poll.h>
unsigned int poll(struct file *, poll_table *);
```

it serves both the `poll()`and the `select()` system call,
the argument `poll_table` is used for `poll_wait()`

return value: bit mask describing operations that won’t block

```
  static struct file_operations fops = {
    read:    device_read,
    ...
    poll:    device_poll,
    owner:   THIS_MODULE
  };
```


poll( )

we include a `device_poll()` in our blocking driver:

```
static unsigned int
device_poll(struct file *filp, poll_table *wait)
{
  unsigned int mask=0;
  int ih;

  if(ir != iw) mask |= POLLIN;  // buffer not empty
                                // read won't block
  ih=ir-1; if(ih<0) ih+=N;
  if(iw != ih) mask |= POLLOUT; // buffer not full
                                // write won't block
  return mask;
}
```


poll( )

if the `poll()` system call is to **wait** for a blocking operation (possibly with timeout), we call `poll_wait()`on that wait queue (`poll` doesn’t wait if the effective return value is not zero)

```
static unsigned int
device_poll(struct file *filp, poll_table *wait)
{
  unsigned int mask=0;

  poll_wait(filp, &wq , wait);

  if(ir != iw) mask |= POLLIN;  // buffer not empty
  return mask;
}
```


Asynchronous Notification

with `poll()` we can ask whether there are new data, but there also is the possibility of asynchr. notification from a file/device

user space, gets a signal:

```
  #include <signal.h>
  signal(SIGIO, sighandler);
```

we have to specify the process as the "owner" of the file/device: `F_SETOWN` command using `fcntl()`, the process ID is saved in `filp->f_owner` (so the kernel knows who to notify)

and we must set the `FASYNC` flag in the device by means of the `F_SETFL` `fcntl()` command

Asynchronous Notification

```
#include <signal.h>

void sigc(int sig)
{
  printf(" new data available \n");
  //signal(SIGIO,sigc);
}

int main(){ ...

  signal(SIGIO, sigc);

  fcntl(fd, F_SETOWN, getpid());
  int oflags = fcntl(fd, F_GETFL);
  fcntl(fd, F_SETFL, oflags | FASYNC);
```


Asynchronous Notification

kernel space:

- when `F_SETFL` is executed to turn on (or off) `FASYNC`, the drivers `fasync()` method is called to notify the driver
- when data arrives, all the registered processes must be sent a `SIGIO` signal

there is one data structure (`fasync_struct`) and two functions:

```
int fasync_helper(fd, filp, int mode,
                   fasync_struct **fa);

void kill_fasync(fasync_struct **fa,
           int sig, int band /* POLL_IN */ );
```


Asynchronous Notification

when `F_SETFL` is executed to turn on (or off) `FASYNC`, the drivers `fasync()` method is called to notify the driver (we include `fasync` in the `file_operations` structure)

```
#include <linux/fs.h>
static struct fasync_struct *async;

static int device_fasync(
           int fd, struct file *filp, int mode)
{
  return fasync_helper(fd, filp, mode, &async);
}
```

`fasync` adds files to or removes files from the list of interested processes (empty list: `async = NULL`)

Asynchronous Notification

when data arrives, all the registered processes must be sent a `SIGIO` signal

at the end of `write()`:

```
if (i>0) {
 if (async) kill_fasync(&async, SIGIO, POLL_IN);
}
```

we have to invoke our `fasync` method when the file is closed, to remove the file from the list:

```
device_fasync(-1, filp, 0);
```

Hardware Management

device driver – abstraction layer between software concepts and hardware circuitry, needs to talk with both sides

every peripheral device is controlled by writing and reading its **hardware registers**, they are either in

- memory address space (memory-mapped I/O) or in
- I/O address space (ports, I/O ports)

I/O address space – a separate address space, separated by additional lines on the control bus, special CPU instructions (memory: `Load, Store, Move` – IO space: `In, Out`)

Hardware Management

- memory address space (memory mapped I/O)
- I/O address space (I/O ports)

not all CPUs have I/O address space, but Intel processors (x86, Pentium, ...) do

memory-mapped I/O is often preferred – CPU cores access memory much more efficiently

for processors without separate I/O space there are the same macros in `<asm/io.h>: inb(), inw(), outb(), outw()` – they fake port I/O by remapping port addresses to memory addresses

Hardware Management

access to hardware registers and RAM memory is very similar – but there is one **important difference**:

I/O operations have **side effects** (the desired reaction of the device)

since memory access speed is critical to CPU performance, the no-side-effect case has been optimised in several ways:

- values are cached (in registers or cache memory)
- instructions are reordered (by hardware or the compiler)

these optimisations can be fatal to correct I/O operations

Hardware Management

these optimisations can be fatal to correct I/O operations (at least in the case of memory-mapped I/O) →

hardware caching is disabled by Linux when accessing I/O regions (registers – you may use `volatile`)

reordering: memory barriers

```
void barrier(void)    <linux/kernel.h>
```

code will store to memory all values that are modified

```
void rmb(void)  <asm/system.h>
void wmb(void)
void  mb(void)
```

read / write / both memory barrier – any I/O accesses appearing before the barrier are completed

Hardware Management

exclusive access – I/O ports must be allocated before being used by the driver:

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

int check_region(ulong start, ulong len);

struct resource *request_region(
         ulong start, ulong len, char *name);

void release_region(ulong start, ulong len);
```

`check_region()` is deprecated (chance of race conditions)

check the result of `request_region()` against `NULL`,
name appears in `/proc/ioports`

Hardware Management

I/O ports must be allocated before being used by the driver

```
#define  PORT  0x378      // 0x378 – 0x37a
                          // parallel port
```

in `open()`:

```
if (! request_region(PORT, 3, "my_parallel" ))
  return -EBUSY;
outb( 0x31, PORT);
```

in `release()`:

```
outb( 0x39, PORT);            // first
release_region(PORT, 3);      // thereafter
```


Hardware Management

```
1 > less /proc/ioports
  0376-0376 : ide1
  03c0-03df : vesafb
  03e8-03ef : serial(auto)
2 > app
  File  /dev/mydev  open: 3  - OK
  ...
1 > less /proc/ioports
  0376-0376 : ide1
  0378-037a : my_parallel
  03c0-03df : vesafb
  03e8-03ef : serial(auto)
3 > app
  File  /dev/mydev  open: -1
  open  /dev/mydev: Device or resource busy
```


Hardware Management

I/O Ports: macros in `<asm/io.h>`

`unsigned` **`inb`**`(unsigned port)` – 8 bit

`inw(),inl()` – 16 bit, 32 bit
alignment, performance penalty (Intel)

`void` **`outb`**`(unsigned char byte, unsigned port)`
`outw(),outl()` – 8 bit, 16 bit, 32 bit

`inb_p()` etc. – pause, small delay for slow bus or hardware

string operations:
`void` **`insb`**`(unsigned port, void *addr, ulong count)`

Hardware Management

I/O Memory – similar to I/O ports

```
<linux/ioport.h> <asm/io.h>
```

```
( int  check_mem_region()  )
  void request_mem_region()
  void release_mem_region()
  /proc/iomem
```

Directly mapped I/O memory:

```
uint ioread8(address); ioread16(), ioread32()
void iowrite8(u8 value, address); etc.
memcpy_fromio(dest,src,num), memcpy_toio()
```

Software-mapped I/O memory (vitual memory)

```
void *ioremap(ulong phys_addr, ulong size);
void iounmap(void *adr);
```


Interrupts

Interrupt :

– synchronization of slow hardware devices with the processor

– a signal that the hardware can send when it wants the processor's attention

the CPU stops whatever it's doing (if it accepts the interrupt and if no one with a higher priority is currently served), saves certain parameters, and calls a service routine (handler)

interrupt handlers are limited in the actions they can perform – an interrupt has to be dealt with when convenient for the hardware, not the CPU – devices have a very small amount of RAM, if you don't read the information when available, it is lost

Interrupts

Control of Interrupts

avoid disabling of interrupts – but anyhow:

```
void disable_irq(int irq);
void enable_irq(int irq);
```

disabling all interrupts

```
void local_irq_save(ulong flags);   //disables
void local_irq_restore(ulong flags);
```


Interrupts

interrupt lines are a limited resource:

```
> less /proc/interrupts
           CPU0
  0:    582484      XT-PIC  timer
  1:      8317      XT-PIC  keyboard
  2:         0      XT-PIC  cascade
  5:      2059      XT-PIC  Allegro, Texas Instr ...
  8:         2      XT-PIC  rtc
  9:     55093      XT-PIC  acpi, usb-uhci
 10:         0      XT-PIC  Texas Instruments PCI4450
 11:         7      XT-PIC  eth0, ohci1394
 14:     10793      XT-PIC  ide0
 15:         3      XT-PIC  ide1
NMI:         0
LOC:         0
```

Interrupts

Installing an Interrupt Handler

```
<linux/interrupt.h>

int request_irq(
  unsigned int irq,      // the requested IRQ
  irqreturn_t
    (*handler)(int, void *, struct pt_regs),
                         // ptr to the handler
  unsigned long flags,   //
  const char *dev_name,  // -> /proc/interrupts
  void  *dev_id );       // ptr to private data

void free_irq(
  unsigned int irq,
  void     *dev_id );
```


Interrupts

Flags:

`SA_INTERRUPT`
"fast interrupt", executed with other interrupts disabled
(the interrupt being serviced is in all cases disabled in the interrupt controller)

`SA_SHIRQ`
the interrupt can be shared between devices

`SA_SAMPLE_RANDOM`
the interrupt can contribute to the entropy pool
(for truly random numbers)

Interrupts

Parallel Port and Interrupt

the parallel port has 3 registers (addr. 0x378 - 0x37a) :

DAT (0x378) R/W – Data Latch, pin1 ... pin9

LIN (0x379) R – Printer Status
Bit 6 (pin 10) - ACK - ready for next char
may generate an interrupt whenever this signal changes from low to high (I can connect pin9 and pin10)

PST (0x37a) R/W – Printer Controls
Bit 4 (0x10) – IRQ enable, ACK generates interrupts (PC – IRQ 7)

Interrupts

a first example driver:

```
static int intrz;

static irqreturn_t  intr_serv(
                       int irq,
                       void *dev_id,
                       struct pt_regs *regs)
{
  char val = inb(PORT);
  if ((val & 0x80) == 0 ) return IRQ_NONE;

  intrz++;

  return IRQ_HANDLED;
}
```

Interrupts

in `open()`:

```
intrz=0;

int k=request_irq(7, intr_serv, SA_INTERRUPT,
                   "IntrParallel", NULL);
if (k<0) return k;

//outb(0x10, PORT+2);     // Interrupt enable
sb = inb(PORT+2);
outb(sb | 0x10, PORT+2);
```

we do it in `open()`, not in `init()` –

limited number of intr. lines

Interrupts

in `release()`:

```
outb( sd & 0xEF, PORT+2);   // Intr disable
free_irq(7, NULL);          // thereafter
```

requested handler:

```
k=request_irq( 7, intr_serv, SA_INTERRUPT,
                 "IntrParallel", NULL);
```

– fast handler (`SA_INTERRUPT`), does not support interrupt sharing (`SA_SHIRQ` is missing)

every time now someone writes a char with the highest bit set (to be precise: changing the highest bit from 0 to 1) to the parallel port, an interrupt occurs and our handler is called

Interrupts

our interrupt handler installed:

```
> less /proc/interrupts
          CPU0
  0:    582484      XT-PIC  timer
  1:      8317      XT-PIC  keyboard
  2:         0      XT-PIC  cascade
  5:      2059      XT-PIC  Allegro, Texas Instr ...
  7:        25      XT-PIC  IntrParallel
  8:         2      XT-PIC  rtc
```

```
> less /proc/stat
...
intr 2441221 2262015 23089 0 3 3 4895 2620 25
     2 129207 0 7 3 0 19372 3 0 0 ...
```


Interrupts

interrupt handlers are limited in the actions they can perform

interrupt mode, **interrupt time** – the handler doesn't execute in the context of a process:

– no access to user space data
– the `current` pointer is not valid
– no sleeping or scheduling may be performed
  (e.g. `kmalloc(..., GPF_KERNEL)`)

Interrupts

fast / slow handlers

(much discussions in older versions)

fast – all interrupts are disabled
slow – interrupts are enabled, except the own one (being serviced)

on modern systems `SA_INTERRUPT` is only intended for a few specific situations (such as timer interrupts)

all interrupt handlers should be as short/fast as possible
→ bottom half processing (at a safer time)

Interrupts

the role of an interrupt handler:

- give feedback to the device
  (clear the "interrupt-pending-bit")
- read or write data from/to the device,
  safe incoming data in a buffer (if you don't read the information when available, it is lost)
- wake up processes sleeping on the device
  if the interrupt signals the arrival of new data

Interrupts

using arguments

```
static irqreturn_t intr_serv(int irq,
                      void *dev_id,
                      struct pt_regs *regs);
```

`int irq` the interrupt number
(may be useful for log messages)

`void *dev_id` ptr to client data (passed to `request_irq`)
(mandatory for shared interrupt lines)

`struct pt_regs *regs`
(for monitoring and debugging)

Interrupts

Interrupt Sharing

interrupt request lines are a scarce resource

differences in the installation (`request_irq(irq,...)`):

the `SA_SHIRQ` bit must be set
the `dev_id` argument must be unique
it is used in `free_irq(irq,dev_id)`
to select the correct handler to release

all handlers for that interrupt must agree in interrupt sharing

Interrupts

interrupt sharing – all handlers for that interrupt must agree in interrupt sharing

on an interrupt request the kernel invokes every handler registered for that interrupt, passing each its own `dev_id`

– each handler must be able to recognize the interrupt source it is responsible for

```
irqreturn_t intr_serv(int irq, void *dev_id,
                       struct pt_regs *regs)
{
   int value = inb(PORT);
   if (!(value & 0x80)) return IRQ_NONE;
   ...
}
```


The Bottom Half

all interrupt handlers should be as short/fast as possible in order not to keep interrupts blocked for long

conflict: work – speed

→ we split the interrupt handler

top half – registered with `request_irq()`, actually responds to the interrupt, saves device data to a buffer and schedules its bottom half – very fast

bottom half – scheduled by the top half to be executed later, at a safer time, can perform longish tasks

The Bottom Half

bottom half –

scheduled by the top half to be executed later, at a safer time: all interrupts are enabled during execution of the bottom half

this setup permits the top half to service a new interrupt while the bottom half is still working

otherwise same restrictions as for interrupt handlers, bottom half doesn't execute in the context of a process – no access to user space data, no sleeping or scheduling

The Bottom Half

bottom half – two different mechanisms:

- Tasklets
- Workqueues

Tasklets (`<linux/interrupt.h>`)

macro: `DECLARE_TASKLET(Name, Function, Data)`

the interrupt handler schedules its bottom half:

```
tasklet_schedule(&Name);
```

our bottom half function:

```
void Function(unsigned long Data);
```

The Bottom Half

an example driver:

```
static int    intrz, intra;
static struct timeval tv;

static void do_tasklet(unsigned long);
DECLARE_TASKLET(bh_tasklet, do_tasklet, &tv);
```

the interrupt handler (top half):

```
static void intr_serv(int irq, void *dev_id,
                      struct pt_regs *regs)
{
  intrz++; do_gettimeofday(&tv);
  tasklet_schedule(&bh_tasklet);
  return IRQ_HANDLED;
}
```


The Bottom Half

an example driver:

```
static int intrz, intra;
```

in `open()`:

```
 intrz=intra=0;
 int k=request_irq(7, intr_serv, 0,
                  "IntrParallel", NULL);
 if (k<0) return k;

 sb = inb(PORT+2);
 outb(sb | 0x10, PORT+2);  // Interrupt enable
```

argument `0` – the flag `SA_INTERRUPT` is not set

The Bottom Half

the bottom half tasklet function:

```
static void do_tasklet(unsigned long data)
{
  int del, sec,min,hr;

  del=intrz-intra; intra=intrz;

  time_t tt=((struct timeval *)data)->tv_sec;

  sec=tt%60;  tt/=60;
  min=tt%60;  tt/=60;  hr =tt%24 +2;

  sprintf(mess,"Interrupt 7 (n=%d, del=%d) \
    at %2d:%02d.%02d ",intrz,del,hr,min,sec);
}
```


The Bottom Half

the bottom half tasklet function:

```
static void do_tasklet(unsigned long data)
{
  ...

  time_t tt=((struct timeval *)data)->tv_sec;
  ...
}
```

`unsigned long` is kind of a dummy data type,
we use a (typecasted) pointer instead

```
   DECLARE_TASKLET(bh_tasklet, do_tasklet,
                  (unsigned long) &tv);
```

The Bottom Half

```
static int intrz, intra;

static void do_tasklet(unsigned long data)
{
  ...
  int del=intrz-intra; intra=intrz;
  ...
}
```

the bottom half should know how many interrupts have arrived since it was last called – but the method above is insecure (why?)

→ better implementation:

```
  del = intrz-intra; intra += del;
```


The Bottom Half

Race Conditions

```
  int del=intrz-intra; intra=intrz;
```

better implementation:

```
  del = intrz-intra; intra += del;
```

similar problems can arise with buffer management

→ using a circular buffer (and so avoiding shared variables) is an effective way of handling concurrent-access problems

The Bottom Half

Race Conditions

handling of concurrent-access problems:

- using a circular buffer
- using spinlocks to enforce mutual exclusion
- using lock variables that are accessed atomically

semaphores may not be used in interrupt handlers – they can put a process to sleep

The Bottom Half

- using spinlocks to enforce mutual exclusion

a spinlock works through a shared variable:
a function may acquire the lock by setting the variable,
any other function needing the lock will query it and
"spin" in a busy-wait loop until it is reset to "available"

```
<asm/spinlock.h>

spinlock_t my_lock = SPIN_LOCK_UNLOCKED;

spin_lock(my_lock);
  ...  // short critical section
spin_unlock(my_lock);
```

The Bottom Half

- using spinlocks to enforce mutual exclusion

```
spin_lock(my_lock); ... spin_unlock(my_lock);
```

to protect against interrupt handlers, use

```
spin_lock_irqsave(my_lock, flags);
spin_unlock_irqrestore(my_lock, flags);
```

they disable/enable interrupts –
otherwise you may end up in a deadlock

these are all macros, do not write: `..., &flags);`

note: in non-SMP machines the spinlock functions expand to nothing but possibly disabling/enabling interrupts

The Bottom Half

- using lock variables that are accessed atomically

atomic (noninterruptible) access to variables
for simple locking schemes

bit operations `<asm/bitops.h>`

```
void set_bit(nr, void *addr);
void clear_bit(nr, void *addr);
int  test_and_set_bit(nr, void *addr);  ...
```

```
while(test_and_set_bit(1,&lock) != 0) wait();
   ... // protected section
clear_bit(1,&lock);
```


The Bottom Half

- using lock variables that are accessed atomically

atomic integer operations `<asm/atomic.h>`

new data type: `atomic_t` (holds an `int`, on some machines not more than 24 bits available)

```
void atomic_set(atomic_t *v, int i);
int  atomic_read(atomic_t *v);
  (... add, sub, inc, dec)

int  atomic_inc_and_test(atomic_t *v);
int  atomic_add_and_test(int i, atomic_t *v);
     ...
```

(they return the previous value)

The End

Not covered in this tutorial:

- DMA and Bus Mastering
- USB Drivers
- Block Drivers
- Plug & Play
- Layered Drivers, kmod
- Intermodule Communication
- Version Support
- Network Drivers