INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

The United Nations University

SMR/774 - 18

THIRD COLLEGE ON MICROPROCESSOR-BASED REAL-TIME CONTROL - PRINCIPLES AND APPLICATIONS IN PHYSICS
26 September - 21 October 1994

ADDITIONAL MATERIAL (III) TO LECTURES PRESENTED BY

Ulrich RAICH
C.E.R.N.-European Organization
For Nuclear Research
P.S. Division
CH- 1211 Geneva
SWITZERLAND

These are preliminary lecture notes, intended only for distribution to participants.

Library for Device Driver Writers

- **register_chrdev**(unsigned int major,
  const char *name,
  struct file_operations *fops)
  registers a character device with the kernel

- **unregister_chrdev**(unsigned int major,
  const char *name)

- **printk** printf style debugging function

- **outb**(char value, unsigned short port)
  unsigned int **inb**(unsigned short port)

- void ***kmalloc**(unsigned int len, int priority)

- **MAJOR**(inode->i_rdev) get major number
  MINOR(inode->i_rdev) get minor number
  (inode is passed into each of the driver routines)

- **get_fs_byte**(char val, char *addr)
  put_fs_byte(char val, char *addr)
  get data from user space into system space

- **request_irq**(unsigned int irq, void *handler,
  SA_INTERRUPT, char *name)
  registers and interrupt handler with the system

- free_irq(unsigned int irq)
  unregister the interrupt handler

Implementing open,close,read,write

- Open: **icpt_open(struct inode *inode, struct file *file)**
  —if Busy is set return EBUSY error
  —initialize Port A to programmed output
  —initialize Port B to programmed input
  —set Busy flag
- Close: **ictp_close(struct inode *inode, struct file *file)**
  —reset the Busy flag
- Write: **ictp_write(struct inode *inode, struct file *file, char *buf, int count)**
  —uses get_fs_byte in order to transfer the data from user-space to system-space
  —copy of non driver code from first lecture
  —some printks added
  —problem: No check for data validity is made
- Read: ictp_read **(struct inode *inode, struct file *file, char *buf, int count)**
  —Code of non driver program is taken oven
  —uses put_fs_byte to transfer data to user space
  —adds a few printks
- The entry points of these routines are put into the fops jumptable

A test program for this new driver is needed. The program will call each of the newly installed driver routines.

Code example: The ictp_write routine

```
/*
 * Write requests on the ictp device.
 */
static int ictp_write(struct inode * inode, struct file * file,
                                char * buf, int count)
{

        char        c, *temp = buf;
        unsigned char ctemp, digit;

      switch (ictp_mode) {
      case ICTP_MODE_RAW:
                  temp = buf;
                     while (count > 0) {
                      c = get_fs_byte(temp);
                      outb(c,ICTP_A);
                      count--;
                      temp++;
                     }

                     return temp-buf;
                     break;
         default:
                     return -EINVAL;
              break;
                   }

}
```

Switching the Buzzer on and off

The buzzer is switched using the ioctl function ICTP_SET_BUZZER with the arguments ICTP_BUZZER_ON and ICTP_BUZZER_OFF

How does the code look like?

The Buzzer is connected to PC4 (see later)

Programming Port C the manual says:

Any of the eight bits of port C can be Set or Reset with a single out instruction. When Port C is being used as status/control for Port A or Port B these bits can be set or reset by using the Set/Reset operation just as if they were data output ports.

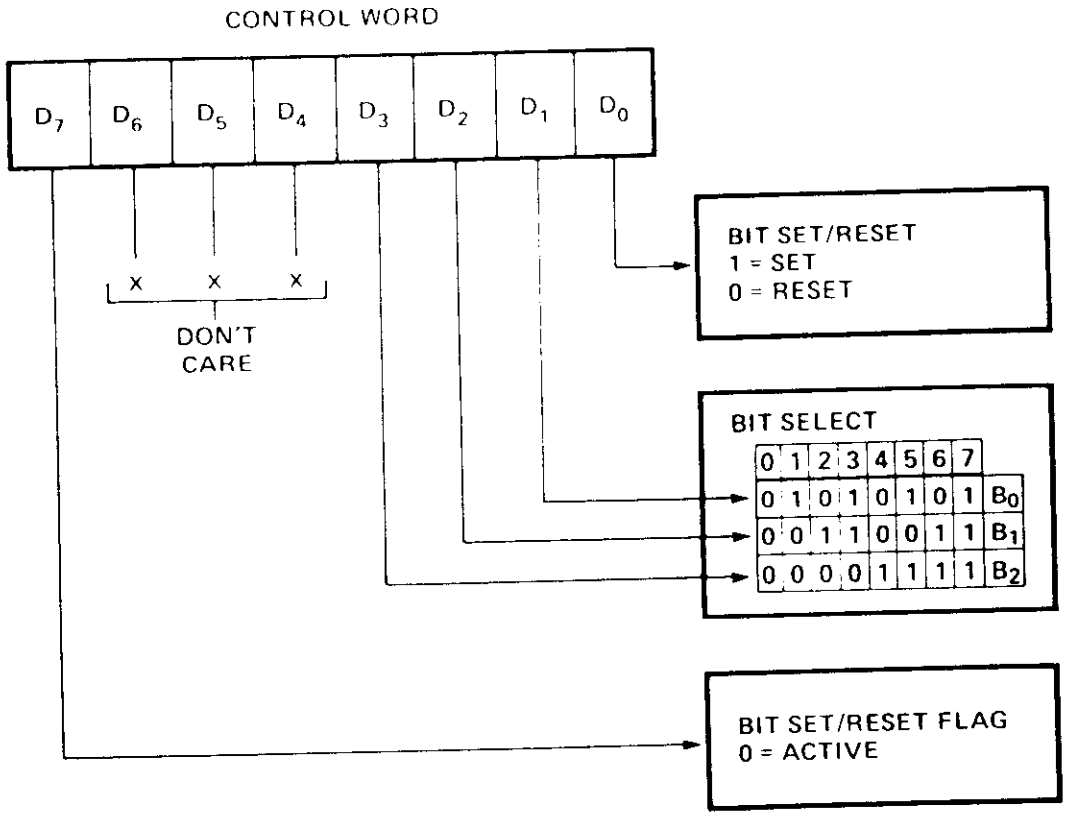


Figure 5. Bit Set/Reset Format

The Buzzer ioctl code

```
#define ICTP_SILENCE          0x09
#define ICTP_NOISE            0x08
/*
 * Handle ioctl calls
 */

static int ictp_ioctl(struct inode * inode, struct file *
file,
                      unsigned int cmd, unsigned long
arg)
{
   unsigned char port_C_status;
   switch (cmd) {

   case ICTP_SET_BUZZER:
     printk("ictp: ioctl set buzzer function entered!\n");
     if (arg == ICTP_BUZZER_ON) {
       outb(ICTP_NOISE,ICTP_S);
       return 0;
     }
     else if (arg == ICTP_BUZZER_OFF) {
       outb(ICTP_SILENCE,ICTP_S);
       return 0;
     }
     else
       return -EINVAL;
     break;

   case ICTP_GET_BUZZER:
     printk("ictp: ioctl get buzzer function entered!\n");
     port_C_status = inb(ICTP_C);
     if (port_C_status & ICTP_BUZZER_BIT)
       return ICTP_BUZZER_OFF;
     else
       return ICTP_BUZZER_ON;
     break;

   default: return -EINVAL;
       }
}
```

Treating Interrupts

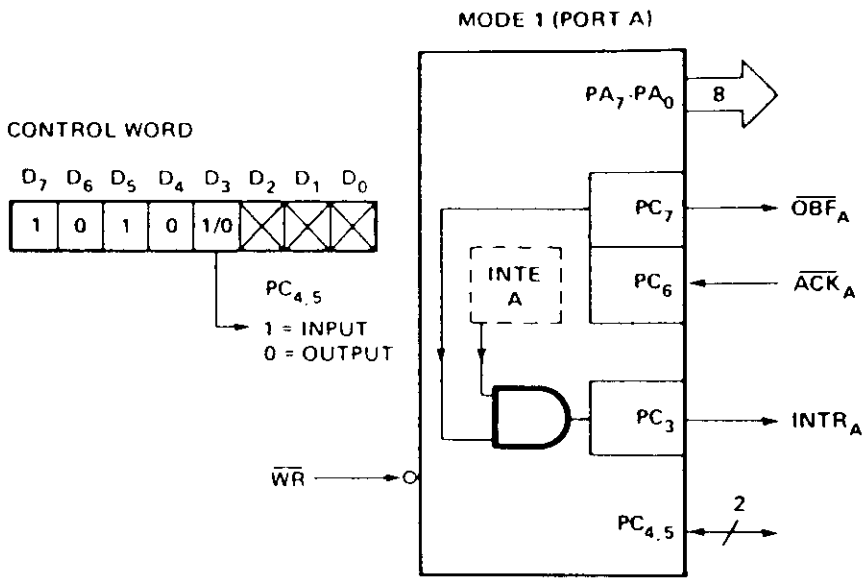
We want to read the number of interrupts, that have arrived after the last read.

With Read... Sleep(1)... Read the second read will give the number of interrupts that arrived during 1 second.
Useful for Voltage to Frequency converter.

Steps to be taken:

- Already a read routine exists, so we need 3 minor devices:
  —minor 0: read switches
  —minor 1: read no of interrupts on IRQ5
  —minor 2: read no of interrupts on IRQ7
- read routine needs a change
  minor = MINOR(inode -> i_rdev) gives minor number
  switch (minor) ... is needed
- open routine needs a change
  —We must change the 8255 initialization mode to **strobed output** (IRQ7) or **strobed input** (IRQ5) depending on the minor number
  (Why not always strobed input mode?)
  —An interrupt service routine must be written and hooked into the system
- close routine must free the interrupt
  (Take out the interrupt service routine from the system)
- ioctl routine to enable and disable the interrupt on 8255 level is needed

Interrupts from Port A
Hardware Connections and Timing



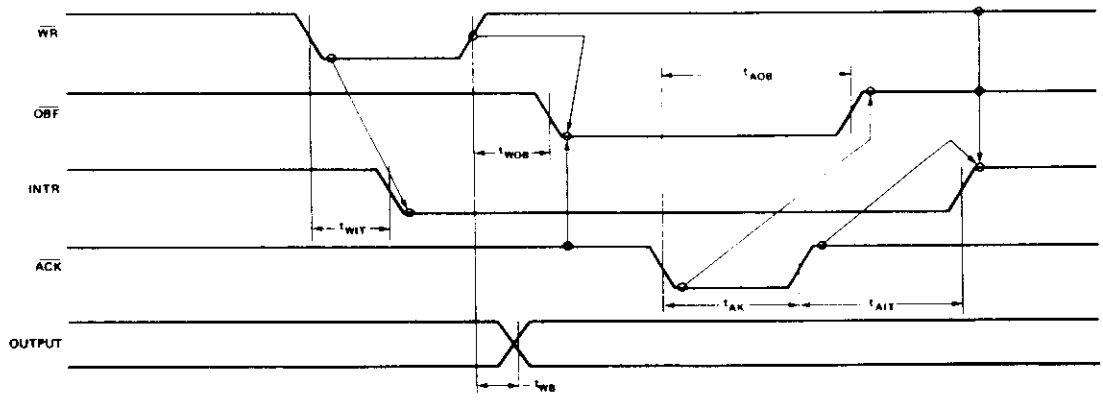
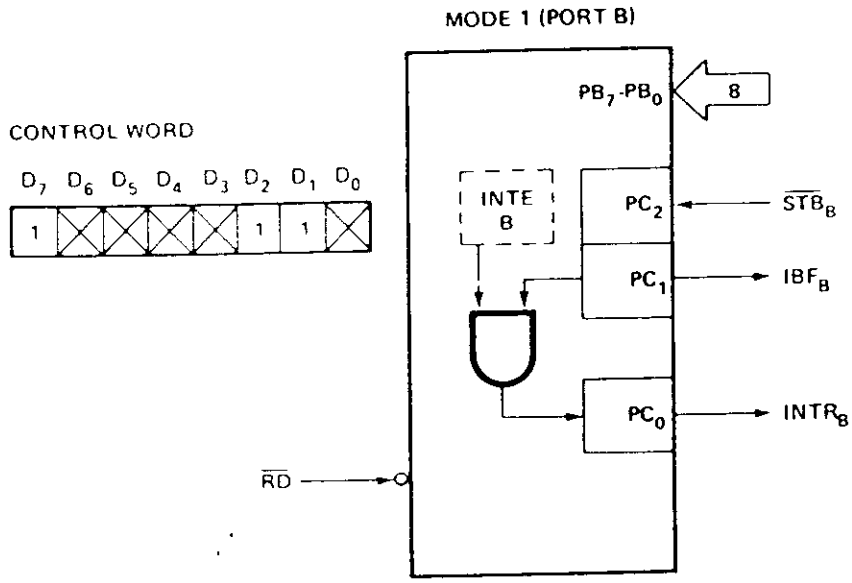


Figure 9. Mode 1 (Strobed Output)

Interrupts from Port B
Hardware Connections and Timing



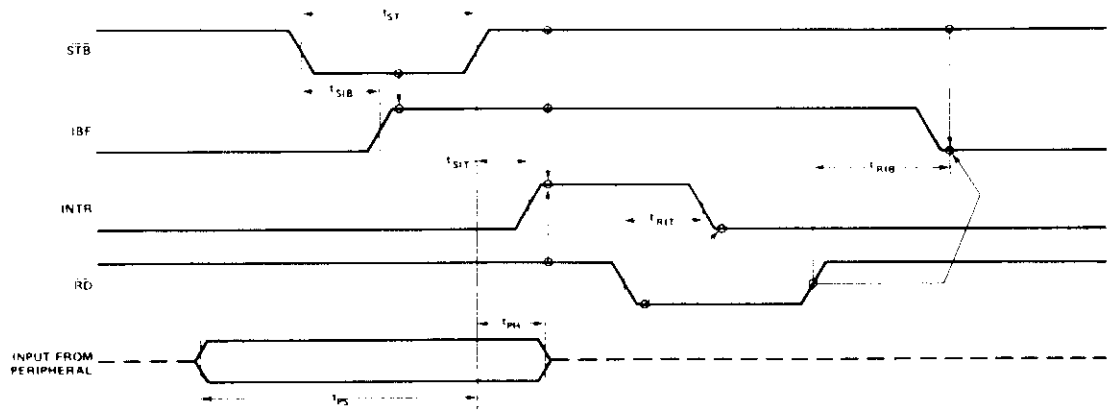


Figure 7. MODE 1 (Strobed Input)

Interrupt code examples

```
/*
  first the tough part: the interrupt code
*/

static void ictp_irq7_interrupt(int irq)
{
  unsigned char dummy;
  dummy = 0xff;
  outb(ICTP_A,dummy);  /* this just clears the interrupt */

  irq7_count++;

}

static int ictp_open(struct inode * inode, struct file *
file)
{
        unsigned int      minor = MINOR(inode->i_rdev);
        unsigned char     command;
        int               ret_code;

        switch (minor) {
/*
  this allows interrupts on the push button
*/
        case ICTP_READ_IRQ7_COUNT
          ret_code=request_irq(irq7,ictp_irq7_interrupt,
                               SA_INTERRUPT,"ictp");
          if (ret_code) {
              printk("ictp: unable to use interupt 7\n");
            return ret_code;
          }
          else {
            printk("ictp: irq7 registered\n");
            command = ICTP_MODE_SELECT |
                      ICTP_A_MODE_1| ICTP_B_MODE_0 |
                      ICTP_INPUT_B;
            outb(command,ICTP_S);        /* strobed output */
/*
  kill the buzzer
 first setup port C to bit set (bit set/reset mode with set
  bit on! )
*/
            outb(ICTP_SILENCE,ICTP_S);
          }
          break; ....
```

Result?

insmod complains that
request_irq and **free_irq** cannot be found
(link error when loading the ictp.o module)

Check where these functions are defined: in the kernel

= > A new kernel version is needed

ftp to a machine where the newest kernel versions are located

The very newest kernel versions exist only in **source form**!
Copy it onto a floppy but ... compressed kernel is 1.6 Mbytes big!
=> Split the kernel into smaller parts, copy these parts onto floppy reassemble the kernel on the hard disk
Compile the new kernel
Install the new modules package

.... Compile error: wrong number of arguments for request_irq
=> Several driver callable routines have changed!!!
New documentation is NOT available
=> Get the source file of the lp: driver and just copy!

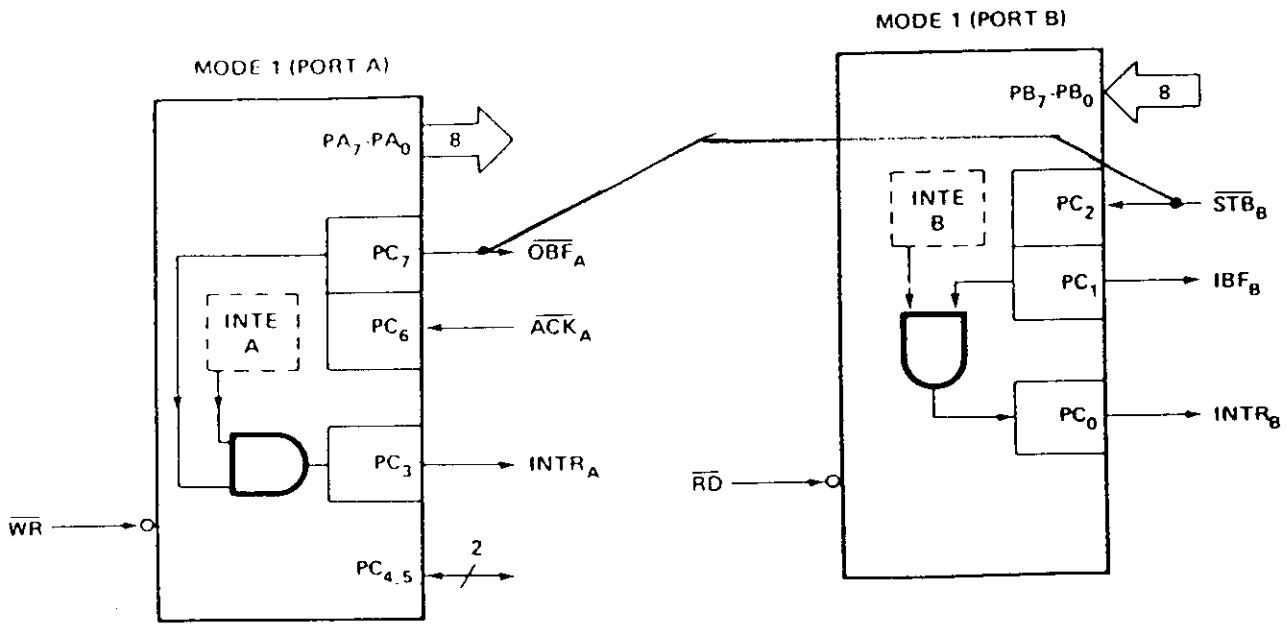
The driver now compiles and can be installed with insmod!

Get both interrupts to work

Change the test program to open and test IRQ5

Result: Influence of IRQ7 on IRQ5 ???

The IBF and OBF signals were shortened!



Now the whole bloody thing works!!!

Conclusion: The interplay of hardware and software can make big trouble!

Future developments of the driver

In the actual driver all read calls are non blocking
=> no way to synchronize to external signals

We can make the driver blocking if no interrupts have arrived since the last read:

```
static struct wait_queue *wait_queue = 0;
static int ictp_read(struct inode *inode, struct file *file,
                     char *buffer, int count)
{
   if (irq5_count == 0) {
     printk(" Process going to sleep on IRQ5\n");
     interruptible_sleep_on(&wait_queue);
     printk("Process has waken up\n");
   }
   put_fs_byte(irq5_count,1);
   return 1;
}
```

In the interrupt service routine we put:

```
wake_up_interruptible(&wait_queue);
```