Sistemsko programiranje – hardware communication

Jernej Vičič
Overview

1. Introduction
2. I/O ports and I/O memory
   - I/O registers and normal memory
3. I/O port usage
   - I/O port allocation
   - I/O port handling
   - I/O port access from user space
   - String instructions
   - Platform dependency
4. Example: I/O port access
   - Parallel port
   - Example driver
5. I/O memory
   - Allocation of I/O memory
   - I/O memory access
Introduction

- Input-Output (I/O),
- driver represents the intermediate level between hardware and program concepts,
- *scull* is a good entry point,
- access to I/O ports (ports)
- access the I/O memory,
Examples

- Examples will be as independent as possible from hardware,
- How I/O commands work:
  - digital I/O – PC parallel port.
- memory mapped I/O:
  - normal frame-buffer video memory.
I/O ports and I/O memory

- peripheral device is controlled by:
  - reading device registers,
  - writing to device registers,
- most of the devices have multiple registers,
- registers are available on consecutive addresses,
- memory map,
- I/O address space.
I/O ports and I/O memory

1. hardware level,
2. there is no conceptual difference between:
   - of the memory region,
   - region I/O.
3. Both are accessed by sending electrical signals to:
   - address bus,
   - control bus.
Implementation is CPU dependent:
- the address space of the peripheral units is separate from the rest of the memory,
- the address space of the peripheral units is not separate from the rest of the memory.

Intel x86:
- separate electrical connections for I/O ports,
- special instructions to access I/O ports.
mimic the most used architectures:

- fake reading and writing of I/O ports,
- using external integrated circuits (chips),
- additional functionality in the processor (embedded devices).

Linux implements I/O port concept on all platforms.
All devices do not use I/O ports:

- ISA peripherals use I/O ports,
- PCI devices remap registers into memory space.

This is the preferred use lately - no need for special instructions.
many similarities between device registers and memory, we need to look at optimizations:

- compiling,
- CPU (cache, recalculate the output of multiple readings and writings),

writing and reading in memory almost no side effects,

optimizations can be performed on memory,

if something is written, it is stored there, nothing happens (usually),

two different write actions can be replaced only with the last write,

the CPU changes the order of the writes (deletes unnecessary writes to the neighbouring done together).
there are side effects when writing to I/O,

essentially the side effects are ”the main thing”,

writing is just a tool,

we must prevent the described optimizations,

CPU optimization itself is not a problem:
  • we read and write to specific memory regions,
  • the optimization is already turned off for these regions.

We turn off the compiler optimization with special barriers that are visible to hardware or other processors.
Linux has 4 macros:

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

- instructs the compiler to set up a barrier,
- does not affect hardware,
- this call prevents the compiler from optimizing over the barrier,
- hardware may perform swapping.
I/O registers and normal memory

#include <asm/system.h>
void rmb(void);
void read_barrier_depends(void);
void wmb(void);
void mb(void);

- **rmb** – (read memory barrier) enables all reads that were instructed before this instruction to be executed before this execution,
- **wmb** – (write memory barrier) enables all writes that were instructed before this instruction to be executed before this execution,
- **mb** – (memory barrier) enables both.
I/O registers and normal memory

```c
void smp_rmb(void);
void smp_read_barrier_depends(void);
void smp_wmb(void);
void smp_mb(void);
```

- same functionality, but for SMP
- Symmetric Multiprocessor System – multiprocessor system with centralised shared memory and single system,
- on non-SMP systems the first function is automatically started.
correctly sets all the registers,
the order is important,
barrier \texttt{wmb} provides this.
barriers affect performance, use with caution.
I/O ports are a tool for communication between drivers and devices,
different functions for using I/O ports,
portability problem.
#include <linux/ioport.h>
struct resource *request_region(unsigned long first, unsigned long n, const char *name);

- we want full control of "our access" (exclusive access),
- function request_region allows this
- first – start port,
- n – number of requested ports,
- name – device name.
- Successful port allocations are written in /proc/iports.
void release_region(unsigned long start, unsigned long n);

- port is "returned" after usage (module unload),

int check_region(unsigned long first, unsigned long n);

- we look whether a certain port has already been seized,
I/O port handling

- most HW distinguishes between 8-bit, 16-bit, and 32-bit,
- this is enabled by functions:

```c
#include <asm/io.h>)
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
```

- read and write to 8-bit (byte) ports (eight bits wide),
- `port` – port number, the type depends on the architecture (unsigned or long).
I/O port handling

unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);

- read and write to 16-bit (2 byte) ports (one word wide),
- not enabled for platform S390 (only byte I/O).
unsigned inl(unsigned port);
void outl(unsigned longword, unsigned port);

- read and write to 32-bit (2 byte) ports (longword wide),
- not enabled for platform S390 (only byte I/O).
- longword – defined as unsigned long or unsigned int,
- not enabled for platform S390 (only byte I/O).
copies of these functions also exist for user space,
are defined in `<sys/io.h>`,
specifics:
- code is translated with `-O` (force expansion of inline functions),
- `ioperm` or `iopl` allow you to perform operations on the I/O ports,
- program must be started with `root`. 
String instructions

- *string instructions*,
- one instruction carries many parts of information to a port,
String instructions

- `void insb(unsigned port, void *addr, unsigned long count);`
- `void outsb(unsigned port, void *addr, unsigned long count);`

- read and write `count` bytes on address `addr`,
- data is written or read from/to one port.
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);

Read or write 16-bit values to a single 16-bit port.

- same as before, only 16 bit port.
String instructions

void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
Read or write 32-bit values to a single 32-bit port.

- same as before, only 32 bit port.
I/O commands are inherently dependent on the processor,
it is difficult to hide differences between systems,
a large part of the I/O code is "platform-dependent",
we mentioned the definition of the port number x86: \textit{unsigned short}, others \textit{unsigned long}. 
Platform dependency

- **IA-32 (x86) and x86_64** – support all the features that we have presented, the port numbers are *unsigned short* type.
- **IA-64 (Itanium)** – supports all the functions that we have presented, the port numbers are *unsigned long* and are mapped to memory, the string functions are implemented.
- **Alpha** – supports all the features that we have presented, the port numbers are *unsigned long*, the implementation depends on the peripheral chipset.
- **ARM** – supports all the functions that we have presented, the port numbers are *unsigned int* and are mapped to memory, the string functions are implemented.
- **Cris** – does not support I/O functions, functions are defined and do nothing.
Platform dependency

- **M68k and M68k-nommux** – support all the functions that we have presented, the port numbers are `unsigned char *` type and mapped to memory, the string functions are implemented.

- **MIPS and MIPS64** – support all the functions that we have presented, the port numbers are type `unsigned long` and mapped to memory, the string functions are implemented tight assembly loops.

- **PA-RISC** – supports all the features that we have presented, the port numbers are `int` type on the PCI and `unsigned short` systems on EISA systems, the string functions are implemented.
Platform dependency

- **PowerPC and PowerPC64** – support all the features that we have presented, the port numbers are *unsigned char* * type on 32 bit systems and *unsigned long* on 64 bit systems.
- **S390** - supports only bite-wide functions, no string operations, port numbers are type *char* * type and mapped to memory.
- **Super-H** - supports all the functions that we have presented, the port numbers are *unsigned int*, mapped to memory.
- **SPARC and SPARC64** - support all the features that we have presented, and the port numbers are *unsigned long*, mapped to memory.
Example: I/O port access

- general digital I/O ports,
- we have on most computer systems,
- the most basic incarnation: the width of the byte mapped to memory or to the gate,
- write the value to the gate, the electrical signal on the output ports of the door changes according to the contents of the written byte,
- when reading we always get the values on pins.
Parallel port

- 3 8-bit port,
- start of the first port is on 0x378,
- start of the first port is on 0x278,
- first port:
  - two-way data register,
  - directly connected to pins 2 – 9,
- second port:
  - read-only register,
  - the printer displays status messages,
  - online, out of paper, busy, ...
- third port:
  - output-only control register,
  - example usage: whether the interrupts are enabled.
Parallel port

- signal levels are standard TTL,
- TTL – Transistor-Transistor Logic:
  - 0 volt – low,
  - 5 volt – high,
  - change at 1.2 volt
- parallel ports are not protected,
- directly connected the logic gates,
- bad circuit can damage the controller.
<table>
<thead>
<tr>
<th>Name</th>
<th>IEC Symbol</th>
<th>American Symbol</th>
<th>Description</th>
<th>Truth Table</th>
</tr>
</thead>
</table>
| IN, AND       | \[A \& B\] | \[A \cdot B\]  | \(Y = A \text{ AND } B\)  
Izhod je 1, če sta oba vhoda 1. | VHOD | IZHOD  |
|               |            |                 |                                                                             | A | B | A \text{ AND } B |
|               |            |                 |                                                                             | 0 | 0 | 0  |
|               |            |                 |                                                                             | 0 | 1 | 0  |
|               |            |                 |                                                                             | 1 | 0 | 0  |
|               |            |                 |                                                                             | 1 | 1 | 1  |
| OR, ORI      | \[A \oplus B\] | \[A \lor B\]   | \(Y = A \text{ OR } B\)  
Izhod je 1, če je vsajeden od vhodov 1. | VHOD | IZHOD  |
|               |            |                 |                                                                             | A | B | A \text{ OR } B |
|               |            |                 |                                                                             | 0 | 0 | 0  |
|               |            |                 |                                                                             | 0 | 1 | 1  |
|               |            |                 |                                                                             | 1 | 0 | 1  |
|               |            |                 |                                                                             | 1 | 1 | 1  |
| NOT, negator | \[\bar{A}\] | \[\neg A\]     | \(Y = \text{ NOT } A\)  
Izhod je 1, če je vhod = 0 in obratno.  
Izhod je negirana vrednost vhoda. | VHOD | IZHOD  |
|               |            |                 |                                                                             | A |  | \text{ NOT } A |
|               |            |                 |                                                                             | 0 | 1 | 1  |
|               |            |                 |                                                                             | 1 | 0 | 0  |
| XOR, ORX     | \[A \oplus B\] | \[A \oplus B\] | \(Y = A \text{ XOR } B\)  
Izhod je 1, če je  
\[\text{ natanke eden izmed vhodov 1.}\] | VHOD | IZHOD  |
|               |            |                 |                                                                             | A | B | A \text{ XOR } B |
|               |            |                 |                                                                             | 0 | 0 | 0  |
|               |            |                 |                                                                             | 0 | 1 | 1  |
|               |            |                 |                                                                             | 1 | 0 | 1  |
|               |            |                 |                                                                             | 1 | 1 | 0  |
| NAND, NE IN  | \[A \& B\] | \[A \cdot B\]  | \(Y = \text{ NOT } (A \text{ AND } B)\)  
Izhod je 0 (1),  
če sta oba vhoda 1.  
(\(\text{ ALL Izhod je 1, če je vsajeden od vhodov 0.}\) | VHOD | IZHOD  |
|               |            |                 |                                                                             | A | B | A \text{ AND } B |
|               |            |                 |                                                                             | 0 | 0 | 1  |
|               |            |                 |                                                                             | 0 | 1 | 1  |
|               |            |                 |                                                                             | 1 | 0 | 1  |
|               |            |                 |                                                                             | 1 | 1 | 0  |
| NOR, NE ALL  | \[A \oplus B\] | \[A \oplus B\] | \(Y = \text{ NOT } (A \text{ OR } B)\)  
Izhod je 0 (1), če je vsajeden od vhodov 1. | VHOD | IZHOD  |
|               |            |                 |                                                                             | A | B | A \text{ OR } B |
|               |            |                 |                                                                             | 0 | 0 | 1  |
|               |            |                 |                                                                             | 0 | 1 | 1  |
|               |            |                 |                                                                             | 1 | 0 | 1  |
|               |            |                 |                                                                             | 1 | 1 | 0  |
Parallel port

Figure: Parallel port.

Jernej Vičič
Sistemsko programiranje – hardware communication
**Parallel port**

**Figure:** Pin usage.

<table>
<thead>
<tr>
<th>Pin No (DB25)</th>
<th>Signal name</th>
<th>Direction</th>
<th>Register bit</th>
<th>Inverted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>nStrobe</td>
<td>Out</td>
<td>Control-0</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Data0</td>
<td>In/Out</td>
<td>Data-0</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>Data1</td>
<td>In/Out</td>
<td>Data-1</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>Data2</td>
<td>In/Out</td>
<td>Data-2</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>Data3</td>
<td>In/Out</td>
<td>Data-3</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>Data4</td>
<td>In/Out</td>
<td>Data-4</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>Data5</td>
<td>In/Out</td>
<td>Data-5</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>Data6</td>
<td>In/Out</td>
<td>Data-6</td>
<td>No</td>
</tr>
<tr>
<td>9</td>
<td>Data7</td>
<td>In/Out</td>
<td>Data-7</td>
<td>No</td>
</tr>
<tr>
<td>10</td>
<td>nAck</td>
<td>In</td>
<td>Status-6</td>
<td>No</td>
</tr>
<tr>
<td>11</td>
<td>Busy</td>
<td>In</td>
<td>Status-7</td>
<td>Yes</td>
</tr>
<tr>
<td>12</td>
<td>Paper-Out</td>
<td>In</td>
<td>Status-5</td>
<td>No</td>
</tr>
<tr>
<td>13</td>
<td>Select</td>
<td>In</td>
<td>Status-4</td>
<td>No</td>
</tr>
<tr>
<td>14</td>
<td>Linefeed</td>
<td>Out</td>
<td>Control-1</td>
<td>Yes</td>
</tr>
<tr>
<td>15</td>
<td>nError</td>
<td>In</td>
<td>Status-3</td>
<td>No</td>
</tr>
<tr>
<td>16</td>
<td>nInitialize</td>
<td>Out</td>
<td>Control-2</td>
<td>No</td>
</tr>
<tr>
<td>17</td>
<td>nSelect-Printer</td>
<td>Out</td>
<td>Control-3</td>
<td>Yes</td>
</tr>
<tr>
<td>18-25</td>
<td>Ground</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Parallel port

Figure: Pin usage.
Parallel port

- 12 output bits,
- 5 input bits,
- bit 4, port 2 (third), irq enable
- some bits are negated.
Example driver

- SHORT,
- Simple Hardware Operations and Raw Tests,
- reads and writes to the 8-bit port,
- starts with the load time,
- by default uses the parallel port ports on a PC,
- each device (has its own minor number) accesses its own port,
- nothing useful :(.
Example driver

- needs free access to a parallel interface,
- no other driver can use this port,
- message: "can not get I/O address" → another driver uses the parallel interface,
- find the device in `/proc/ioports`. 
we cannot connect the printer,
printers require a handshake procedure,
we also need interrupts (not easy),
to see some action, we need to install LEDs on the parallel port,
CAUTION: each LED must have a serially connected $1K\Omega$ resistor that goes into a grounded pin (ground),
we can connect the output pin to the input to read our output.
Example driver

- `/dev/short0`, reads and writes on 8-bit port on address: \(0x378 = base\),
- this address can be changed at startup,
- `/dev/short1`, reads and writes on 8-bit port on address: \(base + 1\),
- this address can be changed at startup,
while (count--) {
    outb(*(ptr++), port);
    wmb();
}

item writing to the gate is in the loop. Item each record is followed by a barrier to overcome the optimization,
echo -n "any string" > /dev/short0

- driver usage (writing),
- if we have LEDs on pins, they will light up,
- basically, the display of all characters except the last for the human eye will be invisible,
- the last character remains lit,
- `echo -n` - so there will be no additional newline at the end,
**Example driver**

```c
while (count--) {
    inb(*(ptr++), port);
    rmb();
}
```

```
dd if=/dev/short0 bs=1 count=1 | od -t x1
```

- driver usage (read)
- we need a device that sends a signal,
- otherwise we always read the same byte,
- if we read the output port, we will get the value that we wrote earlier (very likely)
dd if=/dev/short0 bs=1 count=1 | od -t x1

- if we read the output port, we will get the value that we wrote earlier (very likely)
- this command does exactly that,
the smallest possible control of the hardware,
shows how I/O commands are used,
”real drivers” are complicated.
I/O memory

- I/O ports are simple and popular (only in the x86 world),
- The main mechanism for communicating with devices is through registers mapped in memory and memory of devices,
- I/O memory
I/O memory is part of the RAM that the device provides for the processor,

- provided via bus,

- usage:
  - storage of video data,
  - storage of Ethernet packets,
  - implementation of device registers that act as I/O ports,
  - access mode depends on the architecture,
  - we will present only PCI and ISA.
<linux/ioport.h>

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

- allocates a size of memory,
- \textit{len} – size of allocated memory,
- \textit{start} – start location.
- all locations are presented in:

/proc/iomem
void release_mem_region(unsigned long start, unsigned long len);

- allocated memory can be freed,
- before using, we still need to define virtual addresses for I/O memory regions:

#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);
I/O memory access

```c
#include <asm/io.h>

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

- I/O memory reading functions,
- access to 8, 16 and 32 bits,
- `addr` - the location that was set by `ioremap`.
- return value is the contents of read address.


```c
#include <asm/io.h>

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

- I/O memory write functions,
- write 8, 16 and 32 bit,
- `addr` – location that was set by `ioremap`.
I/O memory access

#include <asm/io.h>;

void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);
void ioread32_rep(void *addr, void *buf, unsigned long count);
void iowrite8_rep(void *addr, const void *buf, unsigned long count);
void iowrite16_rep(void *addr, const void *buf, unsigned long count);
void iowrite32_rep(void *addr, const void *buf, unsigned long count);

- reading/writing a string of values (successive locations),
- writing 8, 16 and 32 bits,
- count - number of read/write values,
- buf - a buffer where we store the read values or write these values,
Ports as I/O memory

```
void *ioport_map(unsigned long port, unsigned int count);
```

- function maps I/O ports and presents them as memory I/O,
- mapping "is removed" with:

```
void ioport_unmap(void *addr);
```
Reuse driver "short" for I/O memory

- module short can be used to access the I/O memory,
- at the start, the module is given the address of the addressed memory region:

```bash
mips.root# ./short_load use_mem=1 base=0xb7fffffff0
mips.root# echo -n 7 > /dev/short0
```

- module writes to memory with this loop:

```c
while (count--) {
  iowrite8(*ptr++, address);
  wmb();
}
```