Sistemsko programiranje – hardware communication

Jernej Vičič

Overview

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

- hardware level,
- there is no conceptual difference between:
  - of the memory region,
  - region I/O.
- Both are accessed by sending electrical signals to:
  - address bus,
  - control bus.

I/O ports and I/O memory

- 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.

I/O ports and I/O memory, other processors

- 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.

I/O ports and I/O memory, memory mapped I/O

- 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.

I/O registers and normal memory

- 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).

/O registers and normal memory

- 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.

/O registers and normal memory

- 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

```
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.

I/O registers and normal memory

```
writel(dev->registers.addr, io_destination_address);
writel(dev->registers.size, io_size);
writel(dev->registers.operation, DEV_READ);
wmb( );
writel(dev->registers.control, DEV_GO);
```

- correctly sets all the registers,
- the order is important,
- barrier *wmb* provides this.
- barriers affect performance, use with caution.

I/O port usage

- I/O pora are a tool for communication between drivers and devices,
- different functions for using I/O ports,
- portability problem.

I/O port allocation

```
#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/ioports*.

I/O port allocation

```
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 enabloed by functions:

```
#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).

I/O port handling

```
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* – definwd as *unsigned long* or *unsigned int*,
- not enabled for platform S390 (only byte I/O).

I/O port access from user space

- 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 writted or readfrom/to one port.

String instructions

```
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.

Platform dependency

- 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*: *unsigned short*, others *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 ** 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.

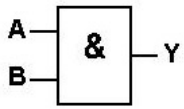
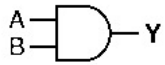
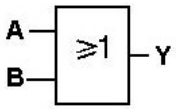
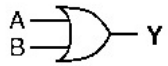
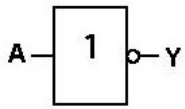
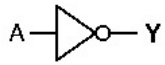
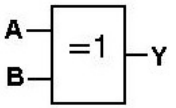
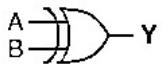
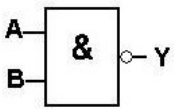
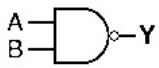
Parallel port

- 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

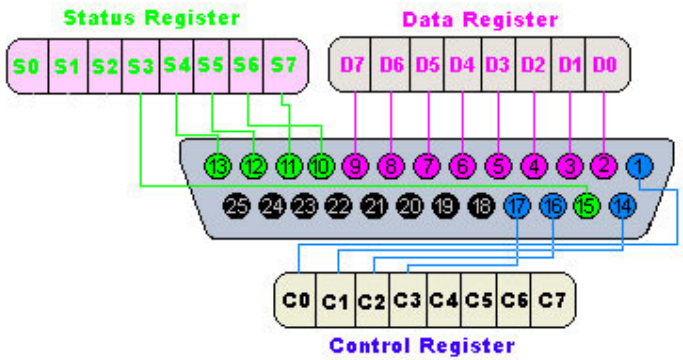
- signal levels are standard TTL,
- TTL – Transistor-Transistor Logic:
  - 0 volt – low,
  - 5 volt – high,
  - change at 1.2 volt
- parallel ports are not protected,
- directly connected the logic gates,
- bad circuit can damage the controller.

Parallel port

ime vrat	IEC simbol vrat	ameriški simbol	opis	pravilnostna tabela
IN, AND			Y = A **AND** B Izhod je 1, če sta **oba vhoda** 1.	VHOD A B / IZHOD A AND B 0 0 0 0 1 0 1 0 0 1 1 1
ALI, OR			Y = A **OR** A Izhod je 1, če je **vsaj eden** od vhodov 1.	VHOD A B / IZHOD A OR B 0 0 0 0 1 1 1 0 1 1 1 1
NE, NOT, negator			Y = **NOT** A (ali: Y = / A) Izhod je = 1, če je vhod = 0 in obratno. Izhod je **negirana vrednost** vhoda.	VHOD A / IZHOD NOT A 0 1 1 0
izključujoči ALI, XOR			Y = A **XOR** B Izhod je 1, če je **natanko eden** izmed vhodov =1.	VHOD A B / IZHOD A XOR B 0 0 0 0 1 1 1 0 1 1 1 0
NE-IN, NAND			Y = **NOT** (A **AND** B) Izhod je **0** (!), če sta **oba vhoda** 1. (Ali: Izhod je 1, če je vsaj eden od vhodov 0.)	VHOD A B / IZHOD A NAND B 0 0 1 0 1 1 1 0 1 1 1 0
NE-ALI,			Y = **NOT** (A **OR** B) Izhod je **0** (!), če je **vsaj eden** od vhodov 1.	VHOD A B / IZHOD A NOR B 0 0 1

Parallel port

Parallel port



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

Figure: Pin usage.

Parallel port

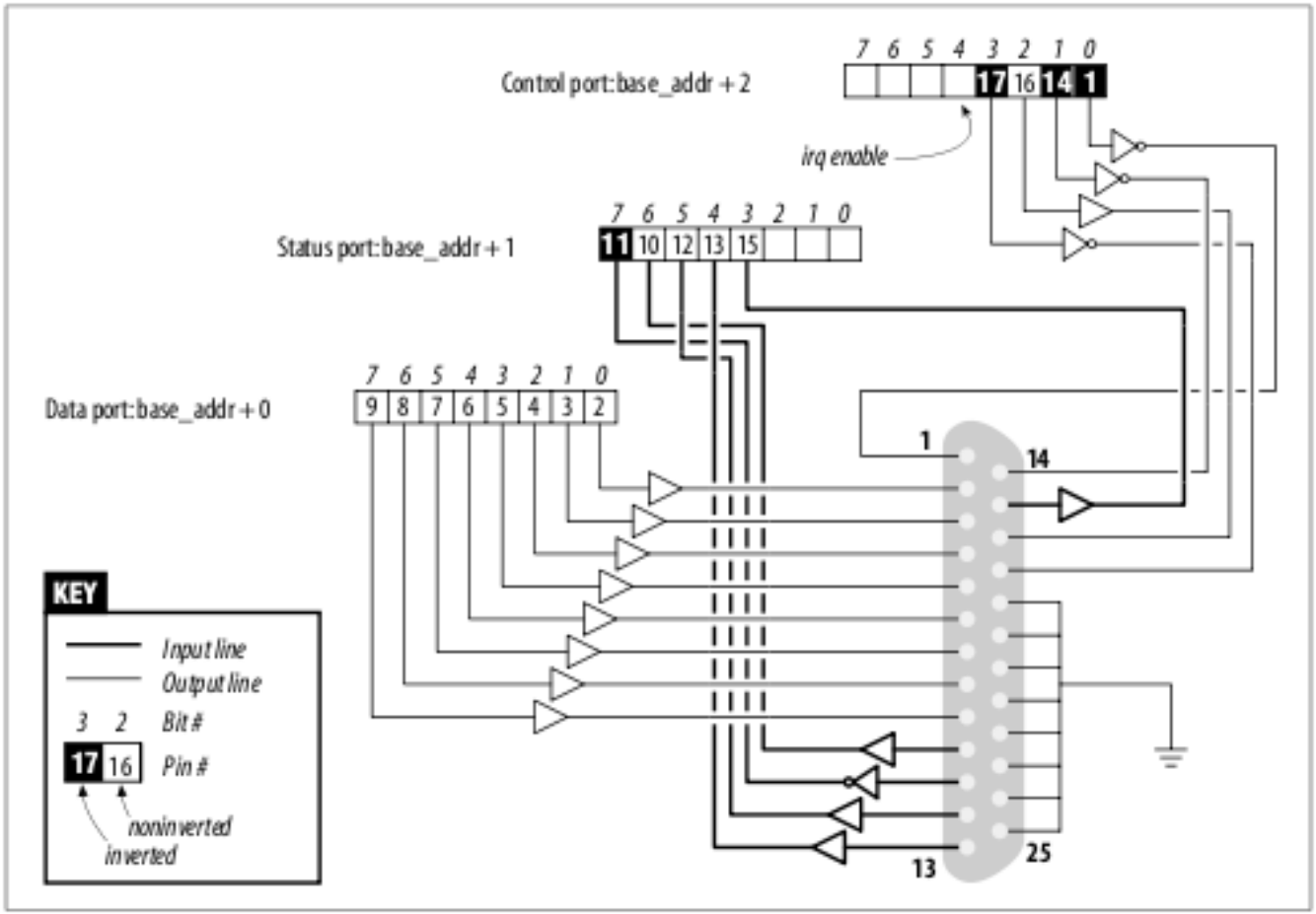


Figure: Pin usage.

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*.

Example driver

- 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,

Example driver

```
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,

Example driver

```
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

```
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)

Example driver

```
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,

Example driver, end

- 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

- 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.

Allocation of I/O memory

```
<linux/ioport.h>

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

- allocates a size of memory,
- *len* – size of allocated memory,
- *start* – start location.
- all locations are presented in:

```
/proc/iomem
```

Allocation of I/O memory

```
void release_mem_region(unsigned long start, unsigned long
```

- 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 lon
void iounmap(void * addr);
```

I/O memory access

```
#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.

```
#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 cou
void ioread32_rep(void *addr, void *buf, unsigned long cou
void iowrite8_rep(void *addr, const void *buf, unsigned lo
void iowrite16_rep(void *addr, const void *buf, unsigned l
void iowrite32_rep(void *addr, const void *buf, unsigned l
```

- 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 meemory region:

```
mips.root# ./short_load use_mem=1 base=0xb7ffffc0
mips.root# echo -n 7 > /dev/short0
```

- module writes to memory with this loop:

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