ioctl() system call

Un'interfaccia comune per controllare i dispositivi

Angelo Dureghello
angelo.dureghello@timesys.com

Sabato 28 Ottobre 2023

Angelo Dureghello

- grande interesse per l'open-source, elettronica e sistemi embedded
- attivo come programmatore su sistemi embedded dal 2001
- mainline Linux kernel contributor, 52 patch (author), 92 contributi totali (log msg), 4 driver completi, 1 driver-maintainer flag.

  `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/log/?h=v6.0&qt=author&q=dureghello`

- U-boot custodian architetture m68k/ColdFire
- progettista di alcune schede linux-embedded (amcore, stmark2, codice in kernel mainline)
- speaker a ELCE 2016, FOSDEM 2020, UNITS, linux-day nazionali e cittadini
- lavora per Timesys Corporation, vive e lavora a Trieste, Italy

In generale, una "system call" (chiamata di sistema) e' una particolare funzione che consente di richiedere un determinato servizio al "cuore" del sistema operativo.

In particolare, in Linux, eseguire una chiamata di sistema significa trasferire il controllo dallo spazio utente (user-space) al kernel (kernel-space).

Sono moltissime, alcuni esempi generici di chiamate di sistema:

- un programma che accede ad un file, l'accesso viene eseguito dal kernel, le system calls saranno open(), read(), write(), etc
- un demone, che si esegue in background, utilizza fork()
- ...
- comunicare con un dispositivo, tramite apposite chiamate ioctl()

ioctl() system call

Introduzione - categorie

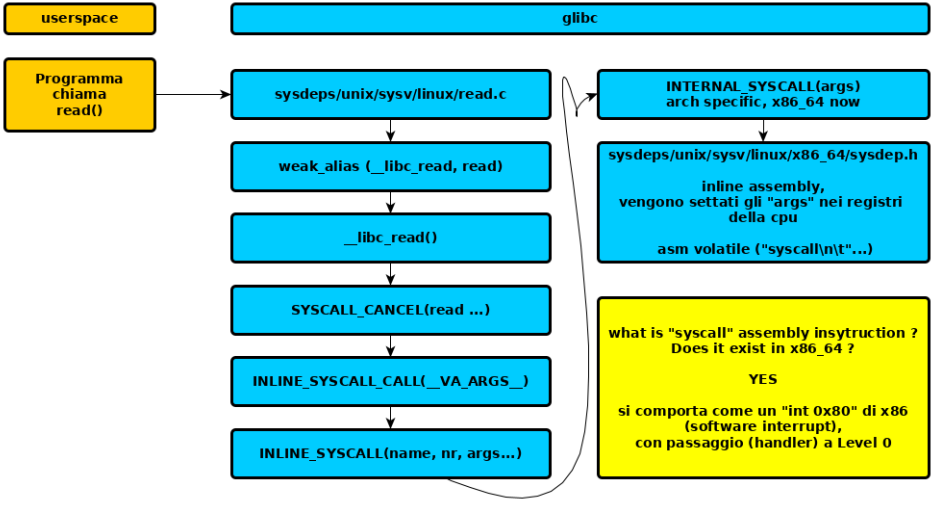
Vengono generalmente invocate tramite una funzione "wrapper" presente nella libc.

Principali categorie:

- Process Control (fork(), exit(), exec(), ...)
- File Management (open(), read(), write(), close(), ...)
- Device Management (ioctl())
- Information Maintenance (getpid(), alarm(), sleep() ...)
- Communication (pipe() , shmget(), mmap() ..)
- ...
- la lista completa e' visibile con "man syscalls" da console, nella lista e' visibile la versione del kernel in cui la chiamata e' stata introdotta

ioctl() system call

Percorso della system call read()



ioctl() system call

Percorso di una system call, glibc, sysdeps/unix/sysv/linux/x86_64/sysdep.h

```
158 /*  The Linux/x86-64 kernel expects the system call parameters in
159     registers according to the following table:
160
161     syscall number rax
162     arg 1       rdi
163     arg 2       rsi
164     arg 3       rdx
165     arg 4       r10
166     arg 5       r8
167     arg 6       r9
168
169     The Linux kernel uses and destroys internally these registers:
170     return address from
171     syscall             rcx
172     eflags from syscall  r11
173 */
```

ioctl() system call

Percorso di una system call, glibc, sysdeps/unix/sysv/linux/x86_64/sysdep.h

Chiamata "extended inline assembly" finale

```
267 #undef internal_syscall2
268 #define internal_syscall2(number, arg1, arg2)                \
269 ({                                                           \
270     unsigned long int resultvar;                             \
271     TYPEFY (arg2, __arg2) = ARGIFY (arg2);                   \
272     TYPEFY (arg1, __arg1) = ARGIFY (arg1);                   \
273     register TYPEFY (arg2, _a2) asm ("rsi") = __arg2;        \
274     register TYPEFY (arg1, _a1) asm ("rdi") = __arg1;        \
275     asm volatile (                                           \
276     "syscall\n\t"                                            \
277     : "=a" (resultvar)                                       \
278     : "0" (number), "r" (_a1), "r" (_a2)                     \
279     : "memory", REGISTERS_CLOBBERED_BY_SYSCALL);             \ /* cc, r11, cx */
280     (long int) resultvar;                                    \
281 })
```

ioctl() system call

Percorso di una system call, glibc, sysdeps/unix/sysv/linux/x86_64/syscall.S, DO_CALL macro chiama syscall()

```
1  /*
2   * long syscall (syscall_number, arg1, arg2, arg3, arg4, arg5, arg6)
3   */
4      .text
5  ENTRY (syscall)
6      movq %rdi, %rax           /* Syscall number -> rax.  */
7      movq %rsi, %rdi           /* shift arg1 - arg5.  */
8      movq %rdx, %rsi
9      movq %rcx, %rdx
10     movq %r8, %r10
11     movq %r9, %r8
12     movq 8(%rsp),%r9          /* arg6 is on the stack.  */
13     syscall                   /* Do the system call.  */
14     cmpq $-4095, %rax         /* Check %rax for error.  */
15     jae SYSCALL_ERROR_LABEL   /* Jump to error handler if error.  */
16     ret                       /* Return to caller.  */
17
18 PSEUDO_END (syscall)
```

ioctl() system call

Passaggio userspace -> kernelspace

ioctl() system call

Percorso di una system call, kernel, arch/x86/entry/entry_64.S

```
297 /*
298  * Registers on entry:
299  * rax  system call number
300  * rcx  return address
301  * r11  saved rflags (note: r11 is callee-clobbered register in C ABI)
302  * rdi  arg0
303  * rsi  arg1
304  * rdx  arg2
305  * r10  arg3 (needs to be moved to rcx to conform to C ABI)
306  * r8   arg4
307  * r9   arg5
308  */
309
310 SYM_CODE_START(entry_SYSCALL_64)
311         UNWIND_HINT_ENTRY
312         ENDBR
313         ...
        call    do_syscall_64
```

ioctl() system call

Percorso di una system call, kernel, arch/x86/entry/common.c

```
99  __visible noinstr void do_syscall_64(struct pt_regs *regs, int nr)
100 {
101         add_random_kstack_offset();
102         nr = syscall_enter_from_user_mode(regs, nr);
103
104         instrumentation_begin();
105
106         if (!do_syscall_x64(regs, nr) && !do_syscall_x32(regs, nr) && nr != -1) {
107                 /* Invalid system call, but still a system call. */
108                 regs->ax = __x64_sys_ni_syscall(regs);
109         }
110
111         instrumentation_end();
112         syscall_exit_to_user_mode(regs);
113 }
```

ioctl() system call

Percorso di una system call, kernel, arch/x86/entry/common.c

```
40 static __always_inline bool do_syscall_x64(struct pt_regs *regs, int nr)
41 {
42         /*
43          * Convert negative numbers to very high and thus out of range
44          * numbers for comparisons.
45          */
46         unsigned int unr = nr;
47
48         if (likely(unr < NR_syscalls)) {
49                 unr = array_index_nospec(unr, NR_syscalls);
50                 regs->ax = sys_call_table[unr](regs);
51                 return true;
52         }
53         return false;
54 }
```

ioctl() system call

Percorso di una system call

- elenco in arch/x86/entry/syscalls/syscall_64.tbl,
- sono sparse nel layer (path) specifico del kernel in uso,
- ad esempio, per read(), SYSCALL_DEFINE3(read, ...) in fs/read_write.c
- ad esempio, per fork(), SYSCALL_DEFINE0(fork) in kernel/fork.c
- ...
- per ioctl(), SYSCALL_DEFINE3(ioctl, ...) in fs/read_write.c

ioctl = Input and Output Control

Serve a comunicare con i "device drivers"

Il maggior utilizzo di questa chiamata e' per eseguire operazioni specifiche quando non e' presente una system call specifica.

ioctl() system call

Interfaccia ioctl()

- ioctl() consente di gestire i dispositivi (device) tramite l'implementazione nel driver,
- ioctl() non e' l'unico modo di comunicare con il driver,
  - accesso a file in memoria (pseudo fs), con open() close() read() write()
  - fops (/dev/xxx,)
  - sysfs (/sys/class/xxxx))
  - procfs (/proc)
  - configfs (mount -t configfs none /config)
  - debugfs (/sys/kernel/debug/xxx)
- ma necessario per effettuare particolari operazioni speficiche diverse da semplici read/write, come ad esempio configurare un codec audio.

Si implementa nel driver per operazioni "speciali" che non possono essere eseguite tramite read/write, come:

- espellere un cdrom
- eseguire una serie di operazioni,
- cambiare il bitrate di un codec audio,
- modificare i paramentri di comunicazione di una porta seriale (baud rate, data, stop bits, ... etc)

ioctl() system call

Interfaccia ioctl()

- La chiamata da user-space, in quanto non strutturata, non e' tra le piu simpatiche agli sviluppatori,
- i comandi (secondo argomento) sono spesso non documentati, diversi per ogni driver, moltissimi,
- spesso necessario capire dal codice stesso del kernel il loro utilizzo,
- e' anche difficle far lavorare alla stessa maniera gli argomenti su architetture diverse, come per un programma userspace 32bit eseguito su cpu 64bit,
- legati ai sorgenti della versione in uso,

- malgrado tutto, rimane spesso una delle scelte piu semplici per eseguire operazioni dirette sui dispositivi.

In Douglas McIlroy's history of Unix, ioctl() e' stata definta
"un armadio pieno di scheletri"
sviluppata principalmente per evitare di creare troppe system calls diverse.

La sua introduzione risale storicamente all'introduzione di stty (v2) in AT&T Unix (System V) anche se i documenti man la attribuiscono a Version 7 AT&T Unix

Nel kernel Linux da sempre /include/linux/sys.h

Sebbene alcune piccole aree del kernel si siano liberate da ioctl(), probailmente vi rimarra' per sempre.

- da userspace ioctl ha il seguente "prototype"

```
int ioctl(int fd, unsigned long cmd, ...);
```

- i tre puntini in questo caso non indicano "piu" argomenti, ma un singolo argomento "opzionale", utilizzati in modo da evitare il type checking.
- tradizionalmente, si tratta di un char *argp
- qualche comando non richiede argomenti, altri un integer, altri un puntatore a un dato.

ioctl() system call

Interfaccia ioctl(), implementazione nei driver

Storicamente, un semplice "character device" driver implementava ioctl tramite la funzione:

```
int (*ioctl) (struct inode *inode, struct file *filp,
              unsigned int cmd, unsigned long arg);
```

Per il problema derivante dal locking tramite BKL, dal kernel 2.6.36 ioctl viene rimosso, e si lascia al driver decidere il tipo di lock: a favore di:

```
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
```

oggidi', kernel 6.5, possiamo utilizzare 2 varianti, o entrambi:

```
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
```

ioctl() system call

Interfaccia ioctl(), implementazione nei driver

In altri "subsystems" e versioni troviamo delle funzioni d'accesso un po' diverse:

```
int (*ioctl) (struct tty_struct *tty, unsigned int cmd,
              unsigned long arg)
int (*ioctl) (struct device *dev, unsigned int cmd,
              unsigned long arg)
int (*ioctl) (struct scsi_device *sdev, unsigned int cmd,
              void __user *arg)
...
```

Resta comunque lo stesso formato, a tre parametri, contesto, comando e valore.

ioctl() system call

I comandi ioctl() li ho, ma dove saranno ?

ioctl() system call

Interfaccia ioctl(), chiamata da userspace

```
0  #include <sys/ioctl.h>
1  #include <fcntl.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4  #include <unistd.h>
5
6  #include "dsc.h"
7
8  int main(int argc, char **argv)
9  {
10     struct dsc_xfer_info inf;
11
12     inf.rcv_words = 12;
13     inf.rx_buff = (uint16_t *)malloc(32 * sizeof(
           uint16_t));
14
15     int fd = open("/dev/dsc", O_RDWR);
16     if (fd == -1)
17          return -1;
18
19     ioctl(fd, DSC_IOCTL_RD_VAR_WORD, &inf);
20
21     close(fd);
22
     return 0;
```

- L'esempio passa al kernel un puntatore a una struttura (&inf),
- talvolta l'argomento puo essere unb semplice intero, facile da gestire lato kernel.
- Dove trovare l'header file con il comando DSC_IOCTL_RD_VAR_WORD ?
- Dove trovare i dettagl della struttura dsc_xfer_info ?

ioctl() system call

Interfaccia ioctl(), chiamata da userspace

- per gli ioctl codes comuni, sono gia installati con libc, li trovate in /usr/include/asm-generic/ioctls.h (#include <linux/ioctl.h>)
- per gli ioctl di un driver specifico, installate i "linux-header" files della versione specifica, li troverete poi in /usr/src/.../include/uapi/linux o include/linux, a seconda della versione del kernel,
- conoscendo la definizione, aiutartsi con grep nella directory **include**,
- si possono scaricare, o clonare, anche gli interi sorgenti del kernel, ma non e' necessario,
- si puo banalmente copiarsi il singolo header file nella directory dove compiliamo,
- in genere i dettagli della struttura da utilizzare si trovano sempre nello stesso header file.

Entriamo nel kernel ...

occhio !

- prima di iniziare ad aggiungere la funzionalita' ioctl() ad un driver e' necessaio scegliere i valori da assegnare al magic code e ai comandi
- istintivamente, molti programmaotri scelgono numeri piccoli partendo da 0, 1 etc
- **ci sono buone ragioni per non farlo, i valori devono essere unici per evitare di inviare un comando giusto a un device sbagliato**
- non e' raro un errore del genere, magari si richiede di cambiare il baud rate a un device che non e' una porta seriale.
- se il valore e' unico, l'applicazione riceve -EINVAL, piuttosto che il driver possa eseguire qualcosa di indesiderato.

ioctl() system call

Interfaccia ioctl(), implementazione nel driver - 1, i comandi

- Inizialmente in Linux i comandi sono costanti a 32bit,
- qualcuno aveva provato a strutturarli,
- per aiutare i programmatori, inizialmente si usavano solo 16bit, con byte meno significativo che era un numero sequenziale, e il byte precedente che identificava il driver, detto anche "magic" byte
- anche se talvolta piu di un byte e' usato per identificare il driver,
- esempio,
  TCGETS ha valore 0x00005401, con 0x54 = 'T' che indica il "terminal" driver,
  CYGETTIMEOUT ha valore 0x00435906, con 0x43 0x59 = 'C' 'Y' indicaano il "cyclades" driver.
- questa struttura si e' evoluta, ma alcuni driver usano ancora questo formato.

Oggigiorno, kernel 6.5, per la scelta dei valori si veda:

- Documentation/userspace-api/ioctl/ioctl-number.rst
- include/uapi/asm-generic/ioctl.h

Scegliendo anche un "magic code" uguale ad un driver esistente, ma un range di valori (Seq#) non occupato.

Oggigiorno, kernel 6.5,

- si definiscono in un header file include/uapi/linux/***.h (generici per subsystem)
- si definiscono in un header file include/uapi/linux/**subsystem**/***.h (drivers del subsystem)
- i.e. include/uapi/linux/misc/mdrv.h par un "misc" driver
- header file name che ricorda il nome del driver
- osservare bene driver simili della versione del kernel in uso, in quanto già approvati, in genere sono ok

Nell'aggiungere nuovi comandi, si dovrebbero utilizzare le macro _IO definite nell'header file <linux/ioctl.h>:

da Documentation/userspace-api/ioctl/ioctl-number.rst

```
======  ==  ============================================
_IO     an ioctl with no parameters
_IOW    an ioctl with write parameters (copy_from_user)
_IOR    an ioctl with read parameters  (copy_to_user)
_IOWR   an ioctl with both write and read parameters
======  ==  ============================================
```

Da Documentation/userspace-api/ioctl/ioctl-number.rst

Code	Seq# (hex)	Include File	Comments
0x00	00-1F	linux/fs.h	conflict!
0x00	00-1F	scsi/scsi_ioctl.h	conflict!
0x00	00-1F	linux/fb.h	conflict!
0x00	00-1F	linux/wavefront.h	conflict!
0x02	all	linux/fd.h	
0x03	all	linux/hdreg.h	
0x04	D2-DC	linux/umsdos_fs.h	Dead since 2.6.11, but don't reuse these.
0x06	all	linux/lp.h	

....

Per un nuovo driver (dsc) Individuo un range libero:

```
0xCA  00-0F  uapi/misc/cxl.h
0xCA  10-2F  uapi/misc/ocxl.h
0xCA  70-7f  uapi/misc/dsc.h   <---
```

Ma, ... occhio 

- custom driver proprietario ? Possibili problemi in caso di aggiornamenti al kernel,
- approvato in mainline, o niente upgrades di versione ? tutto ok

ioctl() system call

Interfaccia ioctl(), implementazione nel driver - 1, i comandi

Creazione dei comandi, in include/uapi/linux/misc/dsc.h:

```
#ifndef _UAPI_MISC_DSC_H
#define _UAPI_MISC_DSC_H

/* ioctl magic */
#define DSC_MAGIC 0xCA
/* ioctl numbers */
#define DSC_IOCTL_RD_VAR_WORD     _IOR(DSC_MAGIC, 0x70, __u32)
#define DSC_IOCTL_RD_VAR_WORDS    _IOR(DSC_MAGIC, 0x71, __u32)
#define DSC_IOCTL_WR_VAR_WORD     _IOW(DSC_MAGIC, 0x72, __u32)
#define DSC_IOCTL_WR_VAR_WORDS    _IOW(DSC_MAGIC, 0x73, __u32)
#define DSC_IOCTL_WR_FLASH_BLK    _IOW(DSC_MAGIC, 0x74, __u32)
#define DSC_IOCTL_SERIAL          _IOW(DSC_MAGIC, 0x75, __u32)

#endif /* _UAPI_MISC_DSC_H */
```

FIOCLEX
Viene settato il "Close on exec" flag, il file in uso viene chiuso se il processo chiamante esegue un nuovo programma.

FIONCLEX
Rimuove il "Close on exec" flag, annullando la richiesta di FIOCLEX

FIOASYNC
Abilita o disabilita le "Asynchronous Notification (SIGIO su dati in arrivo).
Generalmente non utilizzato, si utilizza fcntl() per ottenere la stessa cosa.

FIOQSIZE
Restituisce la dimensione di un file, nel caso di un device driver, restituisce ENOTTY.

FIONBIO
Controlla modalita' blocking / nonblocking, ovvero il flag O_NONBLOCK, anche in questo caso si usa generalmente fcntl(). command.

ioctl() system call

Interfaccia ioctl(), implementazione nel driver - 1, comandi

Prima di ogni operazione all'interno del driver, sara' opportuno verificare il magic byte:

```
if (_IOC_TYPE(cmd) != DSC_MAGIC)
        return -EINVAL;
```

ed eventualemente controllare il comando:

```
if (_IOC_NR(cmd) > DSC_IOC_MAXNR)
        return -EINVAL;
```

Il che puo' essere fatto anche restituendo errore dal "default" sello switch().

Sono possibili altri controlli specifici su read/write:

```
if (_IOC_DIR(cmd) & _IOC_READ)
        ...
if (_IOC_DIR(cmd) & _IOC_WRITE)
        ...
```

Controllare gli argomenti ...

ioctl() system call

Interfaccia ioctl(), implementazione nel driver - 2, l'argomento

- Se si utilizza un intero, caso semplice, esso puo' essere usato direttamente.
- Puntatore:  va verificato che l'indirizzo userspace sia valido. Altrimenti,
  - comportamento errato del driver,
  - kernel oops,
  - corruzzione, problemi di sicurezza

```
int access_ok(const void __user *addr, unsigned long size)
```

- per accedere a singole variabili da userspace

```
#define get_user(x, ptr)
#define put_user(x ,ptr)
```

Settare la funzione ioctl() in "file_operations":

```
1  const struct file_operations dsc_fops = {
2          .owner = THIS_MODULE,
3          .open = dsc_open,
4          .read = dsc_read,
5          .write = dsc_write,
6          .unlocked_ioctl = dsc_ioctl,    /* <====== */
7          .release = dsc_release,
8  };
```

Organizzazione della memoria, per CPU senza MMU

- address space fisico, unico, suddiviso area userspace / kernelspace

Organizzazione della memoria, per CPU con MMU

- userspace: userspace virtual address space (tradotto in fisico da mmu hw)
- userspace: kernelspace virtual address space (vmalloc)
- userspace: kernelspace physical address space

Per copie di memoria, da kernel-space non si opera mai direttamente su indirizzi virtuali userspace,

```
#include <asm/uaccess.h>
...
unsigned long copy_to_user(void __user *to,
      const void *from,
      unsigned long count);
unsigned long copy_from_user(void *to,
      const void __user *from,
      unsigned long count);
```

ioctl() system call

Interfaccia ioctl(), implementazione nel driver - 2, l'argomento

Controlli di sicurezza ! (userspace intoccabile)

```
1  #include <asm/uaccess.h>
2  ...
3  static long dsc_ioctl(struct file *filep, unsigned int cmd, unsigned long arg)
4  {
5          struct dsc_xfer_info inf;
6          ...
7
8          if (!access_ok((void __user *)arg, sizeof(struct dsc_xfer_info))) {
9                  pr_err("invalid userspace access\n");
10                 return -EINVAL;
11         }
12         /* Campi della struttura userspace non sono accessibili, va copiata */
13         if (copy_from_user(&inf, (void __user *)arg,
14                 sizeof(struct dsc_xfer_info))) {
15                 pr_err("invalid copy_from_user access\n");
16                 return -EINVAL;
17         }
18         ...
```

ioctl() system call

Interfaccia ioctl(), implementazione nel driver - 2, l'argomento

Processiamo i comandi, tipicamente, con switch/case:

```
1  static long dsc_ioctl(struct file *filep, unsigned int cmd, unsigned long arg)
2  {
3          ...
4          uint16_t data = *inf.tx_buff;
5
6          /* dopo i vari sanity check */
7          switch (cmd) {
8          case DSC_IOCTL_RD_VAR_WORD:
9                  rx_count = 1;
10                 break;
11         case DSC_IOCTL_RD_VAR_WORDS:
12                 rx_count = data & 0x3FF;
13                 break;
14         default:
15                 return -EINVAL;
           }
           ...
```

ioctl() system call

Interfaccia ioctl(), implementazione nel driver - 3, operazioni del driver

Restituiamo al lato userspace un blocco di dati letto da un dispositivo hardware:

```
1  static long dsc_ioctl(struct file *filep, unsigned int cmd, unsigned long arg)
2  {
3          /* lettura da un hardware di un buffer di dati */
4
5          ptr = (uint16_t *)buffer;
6          count = rx_count;
7
8          while (count--)
9                  *ptr++ = ioread16(addr++);
10
11         copy_to_user(inf.rx_buff, buffer, rx_count);
12         ...
13
14
15         /* Uscita ok */
           return 0;
```

- altro tipico errore e' la scelta del tipo di "lock", e se serva o meno,
- qui e' un discroso complesso, scelta del lock giusto non e' semplice.
- il kernel offre molti lock differenti.
- desidero un contesto atomico ? Supponiamo di si. Non voglio si possano effettuare altre letture/scritture "I/O" fin che non ho finito.
- dallo stesso driver devo gestirte anche degli interrupt ? Supponiamo di si.
- In questo caso scelgo uno spinlock, performante.
- dopo il lock, mi trovo in "contesto atomico", preemption disabilitato, quindi, occhio 
- fondamnetale usare spin_lock_irq() per evitare dealdlocks

ioctl() system call

Interfaccia ioctl(), implementazione nel driver - 4, locking

```
1  static long dsc_ioctl(struct file *filep, unsigned int cmd, unsigned long arg)
2  {
3          /* lettura da un hardware di un buffer di dati */
4
5          ptr = (uint16_t *)buffer;
6          count = rx_count;
7
8          spin_lock_irq(&dsc->lock);
9
10         while (count--)
11                 *ptr++ = ioread16(addr++);
12
13         copy_to_user(inf->rx_buff, buffer, rx_count);
14
15         spin_unlock_irq(&dsc->lock);
16         ...
17
18         /* Uscita ok */
19         return 0;
```

DOMANDE ?

GRAZIE A TUTTI

THANK YOU LUG TRIESTE

Link utili per approfondimenti

Linux kernel - sorgenti mainline

git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

Oppure visibili da web browser, per qualsiasi versione

https://elixir.bootlin.com/linux/v6.6-rc4/source/kernel

glibc

Linux kernel - sorgenti mainline

https://elixir.bootlin.com/glibc/glibc-2.38.9000/