```
// SOLUTION 2016-07-06

/**
 * CE device descriptor. CE devices are PCI devices and can not work in bus
 * mastering (DMA). Transfers must be handled reading from the RBR register each
 * byte.
 */
struct des_ce
{
    // control register address
    ioaddr iCTL;

    // status register address
    ioaddr iSTS;

    // RBR register address
    ioaddr iRBR;

    // synchronization semaphore
    natl sync;

    // mutex semaphor
    natl mutex;

    // destination buffer virtual address
    char * buf;

    // number of bytes to be transferred
    natl quanti;

    // char used to stop the transfer
    char stop;
};
// SOLUTION 2016-07-06

// EXTENSION 2016-07-06
/**
 * Maximum number of CE devices to be initialized at boot.
 */
static const int MAX_CE = 16;

/**
 * CE devices decriptors array.
 */
des_ce array_ce[MAX_CE];

/**
 * Next CE device id to be initialized.
 */
natl next_ce;

// EXTENSION 2016-07-06

// SOLUTION 2016-07-06
/**
 * Called by the IO_TIPO_CEREAD interrupt handler a_ceread in io/io.s.
 *
 * Retrieves from the RBR register of the given CE device a number of bytes
 * equal to 'quanti' into the destination buffer. If the stop char is retrieved
 * the transfer will be stopped before reaching the bytes limit.
 *
 * @param  id      CE device id;
 * @param  buf     destination buffer address;
 * @param  quanti  number of bytes to retrieve;
 * @param  stop    stop char.
 */
extern "C" void c_ceread(natl id, char * buf, natl& quanti, char stop)
{
    // check if the given CE device id is valid
    if (id >= next_ce)
```

```
    {
        // if not, print a warning log message
        flog(LOG_WARN, "CE Device %d does not exit.");

        // abort current process under execution
        abort_p();
    }

    // retrieve CE device descriptor
    des_ce *c = &array_ce[id];

    // wait for the CE device mutex
    sem_wait(c->mutex);

    // set destination buffer address
    c->buf = buf;

    // set number of bytes to be transferred
    c->quanti = quanti;

    // set stop char
    c->stop = stop;

    // write to the control register: enable interrupt requests
    outputb(1, c->iCTL);

    // wait for the synchronization sempahore: set in estern_ce
    sem_wait(c->sync);

    // set number of bytes actually transferred
    quanti -= c->quanti;

    // signal mutex semaphore
    sem_signal(c->mutex);
}

/**
 * Called everytime the CE device having the given id sends an interrupt
 * request.
 *
 * @param  id  the id of the CE device sending the interrupt request.
 */
extern "C" void estern_ce(int id)
{
    // retrieve CE device descriptor
    des_ce *c = &array_ce[id];

    // RBR register temp destination buffer
    natb b;

    // this infinite for loop is needed because once the wfi() is done sending
    // the EOI to the APIC it will also schedule a new process; when a new
    // interrupt request is received from this ce device this process will wake
    // up again and start from where it was ended: without the for loop the
    // function will just end resulting in a dead lock
    for (;;)
    {
        // stop CE device interrupt requests
        outputb(0, c->iCTL);

        // read RBR register content: interrupt request ak
        inputb(c->iRBR, b);

        // write transferred byte
        *c->buf++ = b;

        // decrease number of bytes to be transferred
        c->quanti--;

        // check if either the number of bytes to be transferred has been
```

```
        // reached or the stop char has been retrieved
        if (c->quanti == 0 || b == c->stop)
        {
            // if so, signal synchronization semaphore
            sem_signal(c->sync);
                }
        else
        {
            // otherwise, enable interrupt requests
            outputb(1, c->iCTL);
        }

        // send End Of Interrupt to APIC
        wfi();
    }
}
//   SOLUZIONE 2016-07-06 )

// EXTENSION 2016-07-06
/**
 * Initializes the CE devices on the PCI bus. Called at the end of the I/O
 * module initialization.
 */
bool ce_init()
{
    // loop through the PCI bus devices
    for (natb bus = 0, dev = 0, fun = 0;
         pci_find_dev(bus, dev, fun, 0xedce, 0x1234);
         pci_next(bus, dev, fun))
    {
        // check the number of retrieved CE devices
        if (next_ce >= MAX_CE)
        {
            // print warning log message
                flog(LOG_WARN, "Too many CE devices.");

            // exit for loop
            break;
        }

        // retrieve pointer to available CE device descriptor
        des_ce *ce = &array_ce[next_ce];

        // retrieve base register content
        natw base = pci_read_confl(bus, dev, fun, 0x10);

        // set bit n.0 to 0: retrieve base register address
        base &= ~0x1;

        // set control register address: base
        ce->iCTL = base;

        // set status register address: base + 4
        ce->iSTS = base + 4;

        // set RBR register address: base + 8
        ce->iRBR = base + 8;

        // initialize synchronization semaphore
        ce->sync = sem_ini(0);

        // initialize mutex sempahore
        ce->mutex = sem_ini(1);

        // retrieve PCI device APIC pin
        natb irq = pci_read_confb(bus, dev, fun, 0x3c);

        // activate external process
        activate_pe(estern_ce, next_ce, PRIO, LIV, irq);
```

```
        // log CE device info
        flog(LOG_INFO, "ce%d %2x:%1x:%1x base=%4x IRQ=%d", next_ce, bus, dev, fun, base,
irq);

        // increase CE devices counter
        next_ce++;
    }

    // return true: initialization successful
    return true;
}
// EXTENSION 2016-07-06

////////////////////////////////////////////////////////////////////////////
//                  INIZIALIZZAZIONE DEL SOTTOSISTEMA DI I/O                  //
////////////////////////////////////////////////////////////////////////////

// inizializza i gate usati per le chiamate di IO
//
extern "C" void fill_io_gates(void);

extern "C" natl end;
// eseguita in fase di inizializzazione
//
extern "C" void cmain(int sem_io)
{

        fill_io_gates();
        mem_mutex = sem_ini(1);
        if (mem_mutex == 0xFFFFFFFF) {
                flog(LOG_ERR, "impossible creare semaforo mem_mutex");
                abort_p();
        }
        unsigned long long end_ = (unsigned long long)&end;
        end_ = (end_ + DIM_PAGINA - 1) & ~(DIM_PAGINA - 1);
        heap_init((void *)end_, DIM_IO_HEAP);
        if (!console_init())
                abort_p();
        if (!com_init())
                abort_p();
        if (!hd_init())
                abort_p();

// EXTENSION 2016-07-06

    // initialize CE devices
    if (!ce_init())
    {
        // abort the current process if the initialization does not succeed
        abort_p();
    }

// EXTENSION 2016-07-06

        sem_signal(sem_io);
        terminate_p();
}
```