```
// EXTENSION 2017-02-24

/**
 * Checks if the address virtual pages [base, size) are permanent or can be make
 * permanent in the physical memory.
 *
 * @param  base  virtual pages starting address
 * @param  size  virtual pages length
 *
 * @return  an ID which can be used as argument of the nonresident(natl id)
 *          primitive to undo the operation.
 */
extern "C" natl resident(addr base, natq size);

/**
 * Undoes the operations performed by the resident(addr, natq) primitive.
 */
extern "C" void nonresident(natl id);

// EXTENSION 2017-02-24


// EXTENSION 2017-02-24

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

/**
 * CE device descriptor. Each CE PCI device (identified by the vendor ID 0xedce
 * and device ID 0x1234) occupies 32 bytes in the I/O space starting from the
 * address specified in the BAR0 configuration register. Theese PCI devices are
 * capable of transferring data in PCI Bus Mastering.
 */
struct des_ce
{
    // destination buffer virtual address MSB
    ioaddr iVPTRHI;

    // destination buffer virtual address LSB
    ioaddr iVPTRLO;

    // number of bytes to be transferred
    ioaddr iCNT;

    // directory (32 bit) / tab4 (64 bit) physical address to be used to
    // translate the provided destination buffer virtual address
    ioaddr iCR3;

    // status register: reading it is considered as interrupt request ak
    // in case of errors the transfer is interrupted and the status register bit
    // n. 2 is set to 1
    ioaddr iSTS;

    // command regsiter address: writing 1 here starts the transfers
    ioaddr iCMD;

    // synchronization semaphore
    natl sync;

    // mutex semaphore
    natl mutex;

    // last transfer operation result
    bool error;
};

/**
 * Initialized CE devices array.
```

```
 */
des_ce array_ce[MAX_CE];

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

/**
 * Reads the content of the CPU CR3 register.
 */
extern "C" addr readCR3();

// EXTENSION 2017-02-24


// SOLUTION 2017-02-24

/**
 * Transfers from the CE device having the provided id the number of specified
 * bytes into the destination buffer.
 *
 * @param  id      CE device ID;
 * @param  quanti  number of bytes to be transferred;
 * @param  buf     destination buffer virtual address.
 */
extern "C" bool c_cedmaread(natl id, natl quanti, char *buf)
{
    // tranfer result
    bool rv;

    // check if the given CE device id is valid
    if (id >= next_ce)
    {
        // print warning log message
        flog(LOG_WARN, "Invalid CE device: %d", id);

        // abort calling process
        abort_p();
        }

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

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

    // print log message for debugging purposes
    flog(LOG_DEBUG, "virt %p len %d", buf, quanti);

    // reset error flag to false before starting a new transfer
    ce->error = false;

    // check if the provided buffer virtual address is or can be made resident
    // in memory
    natl rid = resident(buf, quanti);
    if (rid == 0xffffffff)
    {
        // if not, the transfer can not be completed
        rv = false;
        goto out;
    }

    // write destination buffer virtual address MSB
    outputl((natq)buf, ce->iVPTRLO);

    // write destination buffer virtual address LSB
    outputl((natq)buf >> 32, ce->iVPTRHI);
```

```
    // write number of bytes to be transferred
    outputl(quanti, ce->iCNT);

    // retrieve CPU CR3 register content and write it in the
    // CE device CR3 register
    outputl((natq)readCR3(), ce->iCR3);

    // clear status register: might contain previous errors
    outputl(0, ce->iSTS);

    // write 1 in the command register: start transfers
    outputl(1, ce->iCMD);

    // wait for the sync sempahore: DMA transfers completed
    sem_wait(ce->sync);

    // check if there was any error during DMA transfers
    rv = !ce->error;

    // undo the previous resident operation
    nonresident(rid);

out:
    // notify mutex semaphore
    sem_signal(ce->mutex);

    // return transfer result
    return rv;
}

/**
 * Called every time the CE device having the given id sends an interrupt
 * request.
 */
extern "C" void estern_ce(int id)
{
    // retrieve CE device descriptor
    des_ce *ce = &array_ce[id];

    // status register buffer byte
    natl b;

    // without the infinite for loop when the wfi() function sends the EOI and
    // schedules a new process, the next time this process is waken the function
    // will end
    for (;;)
    {
        // read from the CE device status register: interrupt request ak
        inputl(ce->iSTS, b);

        // check bit n. 2 of the status register for errors
        ce->error = (b & 2);

        // notify synchronization semaphore
        sem_signal(ce->sync);

        // send EOI and schedule a new process
        wfi();
    }
}

// SOLUTION 2017-02-24

// EXTENSION 2017-02-24

/**
 * Initializes the CE devices available on the PCI bus 0. A maximum number of
 * MAX_CE device can be initializes which descriptors will be place in the
 * array_ce array. All other CE devices will be ignored.
 */
```

```
bool ce_init()
{
    // loop through PCI device on bus 0 having the required vendor and device ID
    for (natb bus = 0, dev = 0, fun = 0;
         pci_find_dev(bus, dev, fun, 0xedce, 0x1234);
         pci_next(bus, dev, fun))
    {
        // check if the maximum number of CE devices is not exceeded
        if (next_ce >= MAX_CE)
        {
            // if so, print a warning log message
            flog(LOG_WARN, "Too many CE devices");

            // just return to the caller
            break;
        }

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

        // retrieve BAR0 register content
        natw base = pci_read_confl(bus, dev, fun, 0x10);

        // retrieve base address
        base &= ~0x1;

        // set VPTRHI register address: base address
        ce->iVPTRHI = base;

        // set VPTRLO register address: base + 4
        ce->iVPTRLO = base + 4;

        // set CNT register address: base + 8
        ce->iCNT = base + 8;

        // set CR3 register address: base + 12
        ce->iCR3 = base + 12;

        // set status register address: base + 16
        ce->iSTS = base + 16;

        // set command register address: base + 20
        ce->iCMD = base + 20;

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

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

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

        // activate external process for CE device interrupt requests
        activate_pe(estern_ce, next_ce, PRIO, LIV, irq);

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

        // increase initialized CE devices counter
        next_ce++;
    }

    // return CE devices initialization successful
    return true;
}

// EXTENSION 2017-02-24
```

```
////////////////////////////////////////////////////////////////////////////
//                  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 2017-02-24

    // initialize CE devices
    if (!ce_init())
    {
        // abort the calling process in case of errors
        abort_p();
    }
// EXTENSION 2017-02-24

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