```
//EXTENSION 2016-07-27

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

/**
 * The provided buffer for the cedmaread primitive must be aligned to its page,
 * the number of pages to be transferred must be greater than 0 and smaller
 * than 10 pages.
 */
static const int MAX_CE_BUF_DES = 10;

/**
 * The cedmaread primitive will transfer the available bytes to the memory
 * spaces addressed by a vector of transfer descriptors. The address of the
 * vector must be placed in the CE device BMPTR register.
 *
 * Destination buffer descriptor for CE device transfers.
 */
struct ce_buf_des
{
    // memory location physical address
    natl addr;

    // memory location length
    natw len;

    // set to 1 if this is the last descriptor
    natb eod;

    // set to 1 by the CE device if all internal bytes have been transferred
    // till this buffer descriptor
    natb eot;
};

/**
 * CE Device descriptor.
 *
 * Ce devices are capable of working in bus mastering. Each device stores a
 * certain amount if bytes and when the CMD register is set to 1 it will try
 * and move them to the memory space in Bus Mastering (DMA). It is not possible
 * to know the number of stored bytes. The bytes will be transferred to a
 * sequence of memory locations addressed by a vector of transfers descriptors
 * (ce_buf_des) addressed in BMPTR. Each descriptor must provide a starting
 * physical destination address and a length. The device will entirely use all
 * available memory locations until all its internal bytes have been
 * transferred. If the provided transfer descriptors does not provide enough
 * memory locations for all the available bytes, the remaining data will be lost
 * in the transfer. Anyway, the CE device will send an interrupt request the
 * transfer operation is complited (either because there are no more bytes to
 * be transferred or memory locations available). Interrupt requests will always
 * be enabled and reading from the status register will work as interrupt ak.
 */
struct des_ce
{
    // BMPTR register address
        natw iBMPTR;

    // command register address
    natw iCMD;

    // status register address
    natw iSTS;

    // synchronization semaphore
        natl sync;

    // mutex semaphore
```

```
        natl mutex;

    // destination buffers descriptors
        ce_buf_des buf_des[MAX_CE_BUF_DES];

} __attribute__((aligned(128)));

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

/**
 * Number of initialized CE devices.
 */
natl next_ce;

// EXTENSION 2016-07-27

// SOLUTION 2016-07-27

/**
 * Starts the transfers from the CE device having the given ID. The transfers
 * are executed by the PCI device in Bus Mastering (DMA) using the given buffer
 * descriptor array. The CE device will send an interrupt request when the
 * transfers are done. The estern_ce method will handle such interrupt request
 * and set the synchronization semaphore.
 *
 * When the synchronization semaphore is set, al DMA transfers have been
 * completed and we can loop through available buffer descriptors to
 * count the number of bytes actually transferred and check if any of them
 * contains the eot flag.
 *
 * The user will provide a virtual address in 'buf' and a number of bytes to be
 * read. We will have to create the buffer descriptors (ce_buf_des) array
 * manually.
 */
extern "C" bool c_cedmaread(natl id, natl& quanti, char *buf)
{
    // check if the given id is valid
    if (id >= next_ce)
    {
        // print warning log message
        flog(LOG_WARN, "CE device not found: %d", id);

        //
        abort_p();
    }

    // check if the buffer is aligned to the page: to check the alignment we use
    // a bitwise AND wich will return true if at least one of the last 12 least
    // significant bits is not equal to zero in which case the given address is
    // not a multiple of 4096 (the page size)
    if ((natq)buf & 0xfff)
    {
        // print warning log message
        flog(LOG_WARN, "Address %x not aligned to the page.", buf);

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

    // check if the number of bytes to be transferred is greater than zero and
    // smaller than 10 pages
    if (quanti == 0 || quanti > MAX_CE_BUF_DES * 4096)
    {
        // if so, print warning log message
        flog(LOG_WARN, "Invalid value for transfer bytes: %d", quanti);

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

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

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

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

    // last buffer descriptor index
    int i;

    // loop through available buffer descriptors and until the number of bytes
    // to be transferred is reached
    for (i = 0; i < MAX_CE_BUF_DES && quanti; i++)
    {
        // retrieve number of bytes to be transferred
        natw len = quanti;

        // check if len is not bigger than the page size
        if (len > 4096)
        {
            // otherwise decrease it to the page size
            len = 4096;
        }

        // set i-th buffer descriptor physical address
        ce->buf_des[i].addr = (natq)trasforma(buf);

        // set i-th buffer descriptor transfer length (bytes)
        ce->buf_des[i].len = len;

        // set i-th buffer descriptor eot and eod to 0
        ce->buf_des[i].eot = ce->buf_des[i].eod = 0;

        // decrease number of bytes to be transferred
        quanti -= len;

        // increase buffer virtual address by the bytes transferred
        buf += len;

        // print log message for debugging purposes
        flog(LOG_DEBUG, "des[%d] addr %x len %d", i, ce->buf_des[i].addr, ce->buf_des[i].
len);
    }

    // set the last buffer descriptor eod
    ce->buf_des[i - 1].eod = 1;

    // write to the command register: start transfer in BUS Mastering: DMA
    outputl(1, ce->iCMD);

    // wait for the synchronization semaphore: set by estern_ce
    sem_wait(ce->sync);

    // clear bytes to be transferred
    quanti = 0;

    //
    int j;

    // completition flag
    bool complete = false;

    // loop through CE device available buffer descriptors
    for (j = 0; j < i; j++)
    {
        // count transferred bytes for each buffer descriptor to be returned to
        // the caller
        quanti += ce->buf_des[j].len;

        // check if the eot is set (all bytes available transferred)
        if (ce->buf_des[j].eot)
        {
            // set completition flag to be returned
            complete = true;

            // exit for loop
            break;
        }
    }

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

    // return completition flag
    return complete;
}

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

    // input byte buffer
    natl 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 (;;)
    {
        // read CE device status register: interrupt request ak
        inputl(ce->iSTS, b);

        // notify synchronization sempahore: all transfers completed
        sem_signal(ce->sync);

        // send EOI to the APIC and schedule a new process
        wfi();
    }
}
// SOLUTION 2016-07-27

// EXTENSION 2016-07-27

/**
 * Initializes the CE device. Called at the end of the I/O module
 * initialization.
 *
 * Loops through all PCI devices available on bus 0 and looks for those having
 * vendor ID 0xedce and device ID 0x1234. A maximum of MAX_CE devices can be
 * initialized: the remaining ones will simply be ignored.
 */
bool ce_init()
{
    // loop through PCI bus device 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 more CE devices can be initialized
        if (next_ce >= MAX_CE)
        {
            // print warning lo message: maximum number of CE devices exceeded
            flog(LOG_WARN, "Too many CE devices.");

            // exit for loop
            break;
        }

        // retrieve next 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 BMPTR register address: base address
        ce->iBMPTR = base;

        // set command register address: base address + 4
        ce->iCMD = base + 4;

        // set status register address: base address + 8
        ce->iSTS = base + 8;

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

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

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

        // retrieve physical address of the destination buffers descriptors
        addr iff = trasforma(&ce->buf_des[0]);

        // write destination buffers descriptor to the BMPTR register
        outputl(reinterpret_cast<natq>(iff), ce->iBMPTR);

        // activate external process for the APIC pin
        activate_pe(estern_ce, next_ce, PRIO, LIV, irq);

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

        // increase CE device counter
        next_ce++;
    }

    // return true: initialization succeeded
    return true;
}

// EXTENSION 2016-07-27

////////////////////////////////////////////////////////////////////////////////
//                   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 2016-07-27

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

// EXTENSION 2016-07-27

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