Sorbonne-Université Sciences

Architecture Logicielle et Matérielle des Ordinateurs

2019

Gestionnaire Interruptions Exceptions Traps
GIET Code source

licence informatique LU3IN004

Alain Greiner
Quentin Meunier
Franck Wajsbürt
Pirouz Bazargan
Emmanuelle Encrenaz

Code source du **GIET**
(*Gestionnaire d'Interruptions, Exceptions et Trappes*)

U.E. ALMO - LU3IN004

septembre 2019

Table des matières

1 Code du noyau **2**
1.1 giet.s . . . 2
1.2 drivers.h . . . 7
1.3 drivers.c . . . 8
1.4 exc_handler.h . . . 21
1.5 exc_handler.c . . . 21
1.6 irq_handler.h . . . 23
1.7 irq_handler.c . . . 23
1.8 sys_handler.h . . . 27
1.9 sys_handler.c . . . 27
1.10 ctx_handler.h . . . 29
1.11 ctx_handler.c . . . 29
1.12 common.h . . . 31
1.13 common.c . . . 33
1.14 hwr_mapping.h . . . 37

2 Code de la bibliothèque utilisateur **39**
2.1 stdio.h . . . 39
2.2 stdio.c . . . 40

1 Code du noyau

1.1 giet.s

```
/*
 * GIET: Interruption/Exception/Trap Handler for MIPS32 processor
 *
 * The base address of the segment containing this code MUST be 0x80000000, in
 * order to have the entry point at address 0x80000180!!! All messages are
 * printed on the TTY corresponding to the task&processor identifiers.
 *
 * It uses two arrays of functions:
 * - the _cause_vector[16] array defines the 16 causes to enter the GIET
 *   it is initialized in th exc_handler.c file
 * - the _syscall_vector[32] array defines the 32 system calls entry points
 *   it is initialised in the sys_handler.c file
 */

    .section .giet, "ax", @progbits
    .space 0x180

/*
 * GIET Entry point (at address 0x80000180)
 */

    .func   _giet
    .type   _giet, %function

_giet:
    mfc0    $27,    $13                 /* $27 <= Cause register */
    la      $26,    _cause_vector       /* $26 <= _cause_vector */
    andi    $27,    $27,    0x3c        /* $27 <= XCODE*4 */
    addu    $26,    $26,    $27         /* $26 <= &_cause_vector[XCODE] */
    lw      $26,    ($26)               /* $26 <=  _cause_vector[XCODE] */
    jr      $26                         /* Jump indexed by XCODE */

    .endfunc
    .size _giet, .-_giet

/*
 * *** System Call Handler ***
 *
 * A system call is handled as a special function call.
 *  - $2 contains the system call index (< 16).
 *  - $3 is used to store the syscall address
 *  - $4, $5, $6, $7 contain the arguments values.
 *  - The return address (EPC) and the SR are saved in the stack.
 *  - Interrupts are enabled before branching to the syscall.
 *  - All syscalls must return to the syscall handler.
 *  - $2, $3, $4, $5, $6, $7 as well as $26 & $27 can be modified.
 *
 * In case of undefined system call, an error message displays the value of EPC
 * on the TTY corresponding to the processor, and the user program is killed.
 */

    .globl  _sys_handler
    .func   _sys_handler
    .type   _sys_handler, %function

_sys_handler:
    addiu   $29,    $29,    -24     /* 2 slots for SR&EPC, 4 slots for args passing */
    mfc0    $26,    $12             /* load SR */
```

```
    sw      $26,    16($29)         /* save it in the stack */
    mfc0    $27,    $14             /* load EPC */
    addiu   $27,    $27,    4       /* increment EPC for return address */
    sw      $27,    20($29)         /* save it in the stack */

    andi    $26,    $2,     0x1F    /* $26 <= syscall index (i < 32) */
    sll     $26,    $26,    2       /* $26 <= index * 4 */
    la      $27,    _syscall_vector /* $27 <= &_syscall_vector[0] */
    addu    $27,    $27,    $26     /* $27 <= &_syscall_vector[i] */
    lw      $3,     0($27)          /* $3  <= syscall address */

    li      $27,    0xFFFFFFED      /* Mask for UM & EXL bits */
    mfc0    $26,    $12             /* $26 <= SR */
    and     $26,    $26,    $27     /* UM = 0 / EXL = 0 */
    mtc0    $26,    $12             /* interrupt enabled */
    jalr    $3                      /* jump to the proper syscall */
    mtc0    $0,     $12             /* interrupt disbled */

    lw      $26,    16($29)         /* load SR from stack */
    mtc0    $26,    $12             /* restore SR */
    lw      $26,    20($29)         /* load EPC from stack */
    mtc0    $26,    $14             /* restore EPC */
    addiu   $29,    $29,     24     /* restore stack pointer */
    eret                            /* exit GIET */

    .endfunc
    .size _sys_handler, .-_sys_handler

/*
 * *** Interrupt Handler ***
 *
 * This simple interrupt handler cannot be interrupted.
 *
 * All non persistant registers, such as $1 to $15, and $24 to $25, as well as
 * register $31, HI, LO and EPC, are saved in the interrupted program stack, before
 * calling the Interrupt Service Routine. These registers can be used by the
 * ISR code.
 */

    .globl  _int_handler
    .func   _int_handler
    .type   _int_handler, %function

_int_handler:
    addiu   $29,    $29,    -25*4   /* stack space reservation (19 registers to
                                       save and 4 free words to call function) */
    .set noat
    sw      $1,     4*4($29)        /* save $1 */
    .set at
    sw      $2,     5*4($29)        /* save $2 */
    sw      $3,     6*4($29)        /* save $3 */
    sw      $4,     7*4($29)        /* save $4 */
    sw      $5,     8*4($29)        /* save $5 */
    sw      $6,     9*4($29)        /* save $6 */
    sw      $7,     10*4($29)       /* save $7 */
    sw      $8,     11*4($29)       /* save $8 */
    sw      $9,     12*4($29)       /* save $9 */
    sw      $10,    13*4($29)       /* save $10 */
    sw      $11,    14*4($29)       /* save $11 */
    sw      $12,    15*4($29)       /* save $12 */
    sw      $13,    16*4($29)       /* save $13 */
    sw      $14,    17*4($29)       /* save $14 */
    sw      $15,    18*4($29)       /* save $15 */
```



```
    sw      $24,    19*4($29)       /* save $24 */
    sw      $25,    20*4($29)       /* save $25 */
    sw      $31,    21*4($29)       /* save $31 */
    mflo    $26
    sw      $26,    22*4($29)       /* save LO */
    mfhi    $26
    sw      $26,    23*4($29)       /* save HI */
    mfc0    $27,    $14
    sw      $27,    24*4($29)       /* save EPC */

    la      $26,    _int_demux
    jalr    $26                     /* jump to a C function to find the proper ISR */

restore:
    .set noat
    lw      $1,     4*4($29)        /* restore $1 */
    .set at
    lw      $2,     4*5($29)        /* restore $2 */
    lw      $3,     4*6($29)        /* restore $3 */
    lw      $4,     4*7($29)        /* restore $4 */
    lw      $5,     4*8($29)        /* restore $5 */
    lw      $6,     4*9($29)        /* restore $6 */
    lw      $7,     4*10($29)       /* restore $7 */
    lw      $8,     4*11($29)       /* restore $8 */
    lw      $9,     4*12($29)       /* restore $9 */
    lw      $10,    4*13($29)       /* restore $10 */
    lw      $11,    4*14($29)       /* restore $11 */
    lw      $12,    4*15($29)       /* restore $12 */
    lw      $13,    4*16($29)       /* restore $13 */
    lw      $14,    4*17($29)       /* restore $14 */
    lw      $15,    4*18($29)       /* restore $15 */
    lw      $24,    4*19($29)       /* restore $24 */
    lw      $25,    4*20($29)       /* restore $25 */
    lw      $31,    4*21($29)       /* restore $31 */
    lw      $26,    4*22($29)
    mtlo    $26                     /* restore LO */
    lw      $26,    4*23($29)
    mthi    $26                     /* restore HI */
    lw      $27,    4*24($29)       /* return address (EPC) */
    addiu   $29,    $29,    25*4    /* restore stack pointer */
    mtc0    $27,    $14             /* restore EPC */
    eret                            /* exit GIET */

    .endfunc
    .size _int_handler, .-_int_handler

/*
 * *** _task_switch ***
 *
 * A task context is an array of 64 words = 256 bytes. It aims at containing
 * copies of all the processor registers except HI and LO which need not be saved.
 * As much as possible a register is stored at the index defined by its number
 * (for example, $8 is saved in ctx[8]).
 * The exception are :
 * - $0 is not saved since always 0.
 * - $26, $27 are not saved since not used by the task (they are system
 * registers).
 *
 * 0*4(ctx) SR    8*4(ctx) $8    16*4(ctx) $16   24*4(ctx) $24      32*4(ctx) EPC
 * 1*4(ctx) $1    9*4(ctx) $9    17*4(ctx) $17   25*4(ctx) $25      33*4(ctx) CR
 * 2*4(ctx) $2   10*4(ctx) $10   18*4(ctx) $18   26*4(ctx) reserved 34*4(ctx) tty_id + 0x80
 * 3*4(ctx) $3   11*4(ctx) $11   19*4(ctx) $19   27*4(ctx) reserved 35*4(ctx) reserved
 * 4*4(ctx) $4   12*4(ctx) $12   20*4(ctx) $20   28*4(ctx) $28      36*4(ctx) reserved
```

```
 * 5*4(ctx) $5   13*4(ctx) $13   21*4(ctx) $21   29*4(ctx) $29      37*4(ctx) reserved
 * 6*4(ctx) $6   14*4(ctx) $14   22*4(ctx) $22   30*4(ctx) $30      38*4(ctx) reserved
 * 7*4(ctx) $7   15*4(ctx) $15   23*4(ctx) $23   31*4(ctx) $31      39*4(ctx) reserved
 *
 * The return address contained in $31 is saved in the _current task context
 * (in the ctx[31] slot), and the function actually returns to the address
 * contained in the ctx[31] slot of the new task context.
 *
 * This function receives two arguments representing addresses of task
 * contexts, respectively for the current running task to be descheduled and
 * for the next task to be scheduled.
 */

    .globl  _task_switch
    .func   _task_switch
    .type   _task_switch, %function

_task_switch:

    /* save _current task context */

    add     $27,    $4,     $0  /* $27 <= @_task_context_array[current_task_index] */

    mfc0    $26,    $12         /* $26 <= SR */
    sw      $26,    0*4($27)    /* ctx[0] <= SR */
    .set noat
    sw      $1,     1*4($27)    /* ctx[1] <= $1 */
    .set at
    sw      $2,     2*4($27)    /* ctx[2] <= $2 */
    sw      $3,     3*4($27)    /* ctx[3] <= $3 */
    sw      $4,     4*4($27)    /* ctx[4] <= $4 */
    sw      $5,     5*4($27)    /* ctx[5] <= $5 */
    sw      $6,     6*4($27)    /* ctx[6] <= $6 */
    sw      $7,     7*4($27)    /* ctx[7] <= $7 */
    sw      $8,     8*4($27)    /* ctx[8] <= $8 */
    sw      $9,     9*4($27)    /* ctx[9] <= $9 */
    sw      $10,    10*4($27)   /* ctx[10] <= $10 */
    sw      $11,    11*4($27)   /* ctx[11] <= $11 */
    sw      $12,    12*4($27)   /* ctx[12] <= $12 */
    sw      $13,    13*4($27)   /* ctx[13] <= $13 */
    sw      $14,    14*4($27)   /* ctx[14] <= $14 */
    sw      $15,    15*4($27)   /* ctx[15] <= $15 */
    sw      $16,    16*4($27)   /* ctx[16] <= $16 */
    sw      $17,    17*4($27)   /* ctx[17] <= $17 */
    sw      $18,    18*4($27)   /* ctx[18] <= $18 */
    sw      $19,    19*4($27)   /* ctx[19] <= $19 */
    sw      $20,    20*4($27)   /* ctx[20] <= $20 */
    sw      $21,    21*4($27)   /* ctx[21] <= $21 */
    sw      $22,    22*4($27)   /* ctx[22] <= $22 */
    sw      $23,    23*4($27)   /* ctx[23] <= $23 */
    sw      $24,    24*4($27)   /* ctx[24] <= $24 */
    sw      $25,    25*4($27)   /* ctx[25] <= $25 */
    sw      $28,    28*4($27)   /* ctx[28] <= $28 */
    sw      $29,    29*4($27)   /* ctx[29] <= $29 */
    sw      $30,    30*4($27)   /* ctx[30] <= $30 */
    sw      $31,    31*4($27)   /* ctx[31] <= $31 */
    mfc0    $26,    $14
    sw      $26,    32*4($27)   /* ctx[32] <= EPC */
    mfc0    $26,    $13
    sw      $26,    33*4($27)   /* ctx[33] <= CR */

    /* restore next task context */
```



```
    add     $27,    $5,     $0  /* $27<= @_task_context_array[next_task_index] */

    lw      $26,    0*4($27)
    mtc0    $26,    $12         /* restore SR */
    .set noat
    lw      $1,     1*4($27)    /* restore $1 */
    .set at
    lw      $2,     2*4($27)    /* restore $2 */
    lw      $3,     3*4($27)    /* restore $3 */
    lw      $4,     4*4($27)    /* restore $4 */
    lw      $5,     5*4($27)    /* restore $5 */
    lw      $6,     6*4($27)    /* restore $6 */
    lw      $7,     7*4($27)    /* restore $7 */
    lw      $8,     8*4($27)    /* restore $8 */
    lw      $9,     9*4($27)    /* restore $9 */
    lw      $10,    10*4($27)   /* restore $10 */
    lw      $11,    11*4($27)   /* restore $11 */
    lw      $12,    12*4($27)   /* restore $12 */
    lw      $13,    13*4($27)   /* restore $13 */
    lw      $14,    14*4($27)   /* restore $14 */
    lw      $15,    15*4($27)   /* restore $15 */
    lw      $16,    16*4($27)   /* restore $16 */
    lw      $17,    17*4($27)   /* restore $17 */
    lw      $18,    18*4($27)   /* restore $18 */
    lw      $19,    19*4($27)   /* restore $19 */
    lw      $20,    20*4($27)   /* restore $20 */
    lw      $21,    21*4($27)   /* restore $21 */
    lw      $22,    22*4($27)   /* restore $22 */
    lw      $23,    23*4($27)   /* restore $23 */
    lw      $24,    24*4($27)   /* restore $24 */
    lw      $25,    25*4($27)   /* restore $25 */
    lw      $28,    28*4($27)   /* restore $28 */
    lw      $29,    29*4($27)   /* restore $29 */
    lw      $30,    30*4($27)   /* restore $30 */
    lw      $31,    31*4($27)   /* restore $31 */
    lw      $26,    32*4($27)
    mtc0    $26,    $14         /* restore EPC */
    lw      $26,    33*4($27)
    mtc0    $26,    $13         /* restore CR */

    jr      $31                 /* returns to caller */

    .endfunc
    .size _task_switch, .-_task_switch
```

1.2 drivers.h

```
/*
 * Drivers for the following SoCLib hardware components:
 *
 * - mips32
 * - vci_multi_tty
 * - vci_multi_timer
 * - vci_multi_dma
 * - vci_multi_icu
 * - vci_gcd
 * - vci_frame_buffer
 * - vci_block_device
 */

#ifndef _DRIVERS_H_
#define _DRIVERS_H_

/*
 * For retrieving ldscript symbols corresponding to base addresses of
 * peripheral devices.
 */

typedef struct __ldscript_symbol_s __ldscript_symbol_t;

extern __ldscript_symbol_t seg_icu_base;
extern __ldscript_symbol_t seg_timer_base;
extern __ldscript_symbol_t seg_tty_base;
extern __ldscript_symbol_t seg_gcd_base;
extern __ldscript_symbol_t seg_dma_base;
extern __ldscript_symbol_t seg_fb_base;
extern __ldscript_symbol_t seg_ioc_base;

/*
 * Global variables for interaction with ISR
 */

extern volatile unsigned int _dma_status[];
extern volatile unsigned char _dma_busy[];

extern volatile unsigned char _ioc_status;
extern volatile unsigned char _ioc_done;
extern volatile unsigned int _ioc_lock;

extern volatile unsigned char _tty_get_buf[];
extern volatile unsigned char _tty_get_full[];

/*
 * Prototypes of the hardware drivers functions.
 */

unsigned int _procid();
unsigned int _proctime();

unsigned int _timer_write(unsigned int register_index, unsigned int value);
unsigned int _timer_read(unsigned int register_index, unsigned int * buffer);

unsigned int _tty_write(const char * buffer, unsigned int length);
unsigned int _tty_read(char * buffer, unsigned int length);
unsigned int _tty_read_irq(char * buffer, unsigned int length);

unsigned int _ioc_write(unsigned int lba, const void * buffer, unsigned int count);
unsigned int _ioc_read(unsigned int lba, void * buffer, unsigned int count);
```



```
unsigned int _ioc_completed();

unsigned int _icu_write(unsigned int register_index, unsigned int value);
unsigned int _icu_read(unsigned int register_index, unsigned int *buffer);

unsigned int _gcd_write(unsigned int register_index, unsigned int value);
unsigned int _gcd_read(unsigned int register_index, unsigned int *buffer);

unsigned int _fb_sync_write(unsigned int offset, const void * buffer, unsigned int length);
unsigned int _fb_sync_read(unsigned int offset, const void * buffer, unsigned int length);
unsigned int _fb_write(unsigned int offset, const void * buffer, unsigned int length);
unsigned int _fb_read(unsigned int offset, const void * buffer, unsigned int length);
unsigned int _fb_completed();

#endif
```

1.3 drivers.c

```
/*
 * The following global parameters must be defined in a config.h file:
 *
 * - NB_PROCS    : number of processors in the platform
 * - NB_MAXTASKS : max number of tasks per processor
 * - NO_HARD_CC  : No hardware cache coherence
 *
 * The following base addresses must be defined in the ldscript file:
 *
 * - seg_icu_base
 * - seg_timer_base
 * - seg_tty_base
 * - seg_gcd_base
 * - seg_dma_base
 * - seg_fb_base
 * - seg_ioc_base
 */

#include <config.h>
#include <drivers.h>
#include <common.h>
#include <hwr_mapping.h>
#include <ctx_handler.h>

/* Here, we perform some static parameters checking */
#if !defined(NB_PROCS)
# error You must define NB_PROCS in 'config.h' file!
#endif

#if NB_PROCS > 8
# error GIET currently supports a maximum of 8 processors
#endif

#if !defined(NB_MAXTASKS)
# error You must define NB_MAXTASKS in 'config.h' file!
#endif

#if NB_MAXTASKS > 4
# error GIET currently supports a maximum of 4 tasks/processor
#endif

#if !defined(NO_HARD_CC)
# error You must define NO_HARD_CC in 'config.h' file!
#endif
```

```
/*
 * Global (uncachable) variables for interaction with ISR
 *
 * _ioc_lock variable must be integer because 'll/sc' locking mechanism works
 * on integer only.
 */

#define in_unckdata __attribute__((section (".unckdata")))

in_unckdata volatile unsigned int _dma_status[NB_PROCS];
in_unckdata volatile unsigned char _dma_busy[NB_PROCS] = {
    [0 ... NB_PROCS - 1] = 0
};

in_unckdata volatile unsigned char _ioc_status;
in_unckdata volatile unsigned char _ioc_done = 0;
in_unckdata volatile unsigned int _ioc_lock = 0;

in_unckdata volatile unsigned char _tty_get_buf[NB_PROCS * NB_MAXTASKS];
in_unckdata volatile unsigned char _tty_get_full[NB_PROCS * NB_MAXTASKS] = {
    [0 ... NB_PROCS * NB_MAXTASKS - 1] = 0
};

/* *************
 * Mips32 driver
 * *************
 */

/*
 * _procid()
 *
 * Access CP0 and returns current processor's identifier.
 */
unsigned int _procid()
{
    unsigned int ret;
    asm volatile("mfc0 %0, $15, 1" : "=r"(ret));
    return (ret & 0x3FF);
}

/*
 * _proctime()
 *
 * Access CP0 and returns current processor's elapsed clock cycles (since
 * boot-up).
 */
unsigned int _proctime()
{
    unsigned int ret;
    asm volatile("mfc0 %0, $9" : "=r"(ret));
    return ret;
}

/* ********************
 * VciMultiTimer driver
 * ********************
 *
 * - The total number of timers is equal to NB_PROCS. There is one timer per
 *   processor.
 * - These two functions give access in read/write mode any internal
 *   configuration register with 'register_index' of the timer associated to
 *   the running processor.
 */
```



```
/*
 * _timer_write()
 *
 * Write a 32-bit word in a memory mapped register of a timer device. The base
 * address is deduced by the proc_id.
 * - Returns 0 if success, > 0 if error.
 */
unsigned int _timer_write(unsigned int register_index, unsigned int value)
{
    volatile unsigned int * timer_address;
    unsigned int proc_id;

    /* parameters checking */
    if (register_index >= TIMER_SPAN)
        return 1;

    proc_id = _procid();
    timer_address = (unsigned int *) &seg_timer_base + (proc_id * TIMER_SPAN);
    timer_address[register_index] = value; /* write word */

    return 0;
}

/*
 * _timer_read()
 *
 * Read a 32-bit word in a memory mapped register of a timer device. The base
 * address is deduced by the proc_id.
 * - Returns 0 if success, > 0 if error.
 */
unsigned int _timer_read(unsigned int register_index, unsigned int * buffer)
{
    volatile unsigned int * timer_address;
    unsigned int proc_id;

    /* parameters checking */
    if (register_index >= TIMER_SPAN)
        return 1;

    proc_id = _procid();
    timer_address = (unsigned int *) &seg_timer_base + (proc_id * TIMER_SPAN);
    *buffer = timer_address[register_index]; /* read word */

    return 0;
}

/*
 * ******************
 * VciMultiTty driver
 * ******************
 *
 * - The max number of TTYs is equal to NB_PROCS * NB_MAXTASKS.
 *   (one private TTY per task).
 * - For each task, the tty_id is stored in the context of the task (slot 34),
 *   and can be explicitely defined by the system designer in the boot code,
 *   using the _tty_config() function. The actual stored value is (ty_id + 0x80000000)
 *   A 0 value means that the default tty_id must be used. It is computed as :
 *   tty_id = proc_id * NB_MAXTASKS + task_id.
 *
 *   Finally, the TTY address is always computed as : seg_tty_base + tty_id*TTY_SPAN
 */
```

```
/*
 * _tty_config()
 *
 * Initialize the tty_index associated to the task identified by (proc_id, task_id).
 * It returns 1 in case of success and 0 in case of error.
 */
unsigned int _tty_config(unsigned int tty_id, unsigned int proc_id, unsigned int task_id)
{
    if (task_id >= NB_MAXTASKS)
        return 0;
    if (proc_id >= NB_PROCS)
        return 0;
    _task_context_array[(proc_id * NB_MAXTASKS + task_id) * 64 + 34] = tty_id + 0x80000000;
    return 1;
}

/*
 * _tty_write()
 *
 * Write one or several characters directly from a fixed-length user buffer to
 * the TTY_WRITE register of the TTY controler.
 *
 * It doesn't use the TTY_PUT_IRQ interrupt and the associated kernel buffer.
 * This is a non blocking call: it tests the TTY_STATUS register.
 * As soon as the TTY_STATUS[WRITE] bit is set, the transfer stops and the
 * function returns  the number of characters that have been actually written.
 */
unsigned int _tty_write(const char * buffer, unsigned int length)
{
    volatile unsigned int * tty_address;

    unsigned int proc_id;
    unsigned int task_id;
    unsigned int tty_id;

    unsigned int nwritten;

    proc_id = _procid();
    task_id = _current_task_array[proc_id];
    tty_id  = _task_context_array[(proc_id * NB_MAXTASKS + task_id) * 64 + 34];
    if (tty_id == 0)
        tty_id = proc_id * NB_MAXTASKS + task_id;
    else
        tty_id = tty_id - 0x80000000;

    tty_address = (unsigned int *) &seg_tty_base + tty_id*TTY_SPAN;

    for (nwritten = 0; nwritten < length; nwritten++)
    {
        /* check tty's status */
        if ((tty_address[TTY_STATUS] & 0x2) == 0x2)
            break;
        else
            /* write character */
            tty_address[TTY_WRITE] = (unsigned int) buffer[nwritten];
    }

    return nwritten;
}

/*
 * _tty_read_irq()
 *
```



```
 * This non-blocking function uses the TTY_GET_IRQ interrupt and the associated
 * kernel buffer, that has been written by the ISR.
 *
 * It fetches one single character from the _tty_get_buf[tty_index] kernel
 * buffer, writes this character to the user buffer, and resets the
 * _tty_get_full[tty_index] buffer.
 *
 * - Returns 0 if the kernel buffer is empty, 1 if the buffer is full.
 */
unsigned int _tty_read_irq(char * buffer, unsigned int length)
{
    unsigned int proc_id;
    unsigned int task_id;
    unsigned int tty_id;

    proc_id = _procid();
    task_id = _current_task_array[proc_id];
    tty_id  = _task_context_array[(proc_id * NB_MAXTASKS + task_id) * 64 + 34];
    if (tty_id == 0)
        tty_id = proc_id * NB_MAXTASKS + task_id;
    else
        tty_id = tty_id - 0x80000000;

    if (_tty_get_full[tty_id] == 0)
        return 0;

    *buffer = _tty_get_buf[tty_id];
    _tty_get_full[tty_id] = 0;
    return 1;
}

/*
 * _tty_read()
 *
 * This function fetches one character directly from the TTY_READ register of
 * the TTY controler controler, and writes this character to the user buffer.
 *
 * It doesn't use the TTY_GET_IRQ interrupt and the associated kernel buffer.
 * This is a non-blocking call: it tests the TTY_STATUS register.
 *
 * - Returns 0 if the register is empty, 1 if the register is full.
 */
unsigned int _tty_read(char * buffer, unsigned int length)
{
    volatile unsigned int * tty_address;

    unsigned int proc_id;
    unsigned int task_id;
    unsigned int tty_id;

    proc_id = _procid();
    task_id = _current_task_array[proc_id];
    tty_id  = _task_context_array[(proc_id * NB_MAXTASKS + task_id) * 64 + 34];
    if (tty_id == 0)
        tty_id = proc_id * NB_MAXTASKS + task_id;
    else
        tty_id = tty_id - 0x80000000;

    tty_address = (unsigned int *) &seg_tty_base + tty_id * TTY_SPAN;

    if ((tty_address[TTY_STATUS] & 0x1) != 0x1)
        return 0;
```

```
    *buffer = (char) tty_address[TTY_READ];
    return 1;
}

/*
 * ******************
 * VciMultiIcu driver
 * ******************
 *
 * The total number of ICUs is equal to NB_PROCS. There is one ICU per
 * processor.
 */

/*
 * _icu_write()
 *
 * Write a 32-bit word in a memory mapped register of the ICU device. The
 * base address is deduced by the proc_id.
 * - Returns 0 if success, > 0 if error.
 */
unsigned int _icu_write(unsigned int register_index, unsigned int value)
{
    volatile unsigned int * icu_address;
    unsigned int proc_id;

    /* parameters checking */
    if (register_index >= ICU_END)
        return 1;

    proc_id = _procid();
    icu_address = (unsigned int *) &seg_icu_base + (proc_id * ICU_SPAN);
    icu_address[register_index] = value;   /* write word */
    return 0;
}

/*
 * _icu_read()
 *
 * Read a 32-bit word in a memory mapped register of the ICU device. The
 * ICU base address is deduced by the proc_id.
 * - Returns 0 if success, > 0 if error.
 */
unsigned int _icu_read(unsigned int register_index, unsigned int *buffer)
{
    volatile unsigned int * icu_address;
    unsigned int proc_id;

    /* parameters checking */
    if (register_index >= ICU_END)
        return 1;

    proc_id = _procid();
    icu_address = (unsigned int *) &seg_icu_base + (proc_id * ICU_SPAN);
    *buffer = icu_address[register_index]; /* read word */
    return 0;
}

/*
 * *************
 * VciGcd driver
 * *************
 */
```



```
/*
 * _gcd_write()
 *
 * Write a 32-bit word in a memory mapped register of the GCD coprocessor.
 * - Returns 0 if success, > 0 if error.
 */
unsigned int _gcd_write(unsigned int register_index, unsigned int value)
{
    volatile unsigned int * gcd_address;

    /* parameters checking */
    if (register_index >= GCD_END)
        return 1;

    gcd_address = (unsigned int *) &seg_gcd_base;
    gcd_address[register_index] = value; /* write word */
    return 0;
}

/*
 * _gcd_read()
 *
 * Read a 32-bit word in a memory mapped register of the GCD coprocessor.
 * - Returns 0 if success, > 0 if error.
 */
unsigned int _gcd_read(unsigned int register_index, unsigned int * buffer)
{
    volatile unsigned int * gcd_address;

    /* parameters checking */
    if (register_index >= GCD_END)
        return 1;

    gcd_address = (unsigned int *) &seg_gcd_base;
    *buffer = gcd_address[register_index]; /* read word */
    return 0;
}

/*
 * *********************
 * VciBlockDevice driver
 * *********************
 *
 * The three functions below use the three variables _ioc_lock _ioc_done,  and
 * _ioc_status for synchronsation.
 * - As the IOC component can be used by several programs running in parallel,
 *   the _ioc_lock variable guaranties exclusive access to the device.  The
 *   _ioc_read() and _ioc_write() functions use atomic LL/SC to get the lock.
 *   and set _ioc_lock to a non zero value.  The _ioc_write() and _ioc_read()
 *   functions are blocking, polling the _ioc_lock variable until the device is
 *   available.
 * - When the tranfer is completed, the ISR routine activated by the IOC IRQ
 *   set the _ioc_done variable to a non-zero value. Possible address errors
 *   detected by the IOC peripheral are reported by the ISR in the _ioc_status
 *   variable.
 * The _ioc_completed() function is polling the _ioc_done variable, waiting for
 * tranfer conpletion. When the completion is signaled, the _ioc_completed()
 * function reset the _ioc_done variable to zero, and releases the _ioc_lock
 * variable.
 *
 * In a multi-processing environment, this polling policy should be replaced by
 * a descheduling policy for the requesting process.
 */
```

```
/*
 * _ioc_get_lock()
 *
 * This blocking helper is used by '_ioc_read()' and '_ioc_write()' functions
 * to get _ioc_lock using atomic LL/SC.
 */
static inline void _ioc_get_lock()
{
    register unsigned int delay = (_proctime() & 0xF) << 4;
    register unsigned int * plock = (unsigned int *) &_ioc_lock;

    asm volatile (
            "_ioc_llsc:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
            "ll␣␣␣$2,␣␣␣␣0($0)␣␣␣␣␣␣␣\n" /* $2 <= _ioc_lock current value */
            "bnez␣$2,␣␣␣␣_ioc_delay␣\n" /* delay if _ioc_lock already taken */
            "li␣␣␣$3,␣␣␣␣1␣␣␣␣␣␣␣␣␣\n" /* $3 <= argument for sc */
            "sc␣␣␣$3,␣␣␣␣0($0)␣␣␣␣␣␣␣\n" /* try to set _ioc_lock */
            "bnez␣$3,␣␣␣␣_ioc_ok␣␣␣␣\n" /* exit if atomic */
            "_ioc_delay:␣␣␣␣␣␣␣␣␣␣␣␣\n"
            "move␣$4,␣␣␣␣%1␣␣␣␣␣␣␣␣␣\n" /* $4 <= delay */
            "_ioc_loop:␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
            "addi␣$4,␣␣␣␣$4,␣␣␣␣-1␣␣\n" /* $4 <= $4 - 1 */
            "bnez␣$4,␣␣␣␣_ioc_loop␣␣\n" /* test end delay */
            "j␣␣␣␣␣␣␣␣␣␣␣_ioc_llsc␣␣\n" /* retry */
            "_ioc_ok:␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
            :
            :"r"(plock), "r"(delay)
            :"$2", "$3", "$4");
}

/*
 *  _ioc_write()
 *
 * Transfer data from a memory buffer to a file on the block_device. The source
 * memory buffer must be in user address space.
 * - lba    : first block index on the disk.
 * - buffer : base address of the memory buffer.
 * - count  : number of blocks to be transfered.
 *
 * - Returns 0 if success, > 0 if error.
 */
unsigned int _ioc_write(unsigned int lba, const void * buffer, unsigned int count)
{
    volatile unsigned int * ioc_address;

    ioc_address = (unsigned int *) &seg_ioc_base;

    /* parameters checking */
    /* buffer must be in user space */
    unsigned int block_size = ioc_address[BLOCK_DEVICE_BLOCK_SIZE];

    if (((unsigned int) buffer >= 0x80000000)
            || (((unsigned int) buffer + block_size * count) >= 0x80000000))
        return 1;

    /* get the lock on ioc device */
    _ioc_get_lock();

    /* block_device configuration for the write transfer */
    ioc_address[BLOCK_DEVICE_BUFFER] = (unsigned int) buffer;
    ioc_address[BLOCK_DEVICE_COUNT] = count;
    ioc_address[BLOCK_DEVICE_LBA] = lba;
```



```
    ioc_address[BLOCK_DEVICE_IRQ_ENABLE] = 1;
    ioc_address[BLOCK_DEVICE_OP] = BLOCK_DEVICE_WRITE;
    return 0;
}

/*
 * _ioc_read()
 *
 * Transfer data from a file on the block device to a memory buffer. The destination
 * memory buffer must be in user address space.
 * - lba    : first block index on the disk.
 * - buffer : base address of the memory buffer.
 * - count  : number of blocks to be transfered.
 *
 * - Returns 0 if success, > 0 if error.
 *
 * Note: all cache lines corresponding to the the target buffer are invalidated
 * for cache coherence.
 */
unsigned int _ioc_read(unsigned int lba, void * buffer, unsigned int count)
{
    volatile unsigned int * ioc_address;

    ioc_address = (unsigned int *) &seg_ioc_base;

    /* parameters checking */
    /* buffer must be in user space */
    unsigned int block_size = ioc_address[BLOCK_DEVICE_BLOCK_SIZE];

    if (((unsigned int) buffer >= 0x80000000)
            || (((unsigned int) buffer + block_size * count) >= 0x80000000))
        return 1;

    /* get the lock on ioc device */
    _ioc_get_lock();

    /* invalidation of data cache */
    if (NO_HARD_CC) _dcache_buf_invalidate(buffer, block_size * count);

    /* block_device configuration for the read transfer */
    ioc_address[BLOCK_DEVICE_BUFFER] = (unsigned int) buffer;
    ioc_address[BLOCK_DEVICE_COUNT] = count;
    ioc_address[BLOCK_DEVICE_LBA] = lba;
    ioc_address[BLOCK_DEVICE_IRQ_ENABLE] = 1;
    ioc_address[BLOCK_DEVICE_OP] = BLOCK_DEVICE_READ;


    return 0;
}

/*
 * _ioc_completed()
 *
 * This function checks completion of an I/O transfer and reports errors. As it
 * is a blocking call, the processor is stalled until the next interrupt.
 *
 * - Returns 0 if success, > 0 if error.
 */
unsigned int _ioc_completed()
{
    unsigned int ret;

    /* busy waiting */
```

```
    while (_ioc_done == 0)
        asm volatile("nop");

    /* test IOC status */
    if ((_ioc_status != BLOCK_DEVICE_READ_SUCCESS)
            && (_ioc_status != BLOCK_DEVICE_WRITE_SUCCESS))
        ret = 1;        /* error */
    else
        ret = 0;        /* success */

    /* reset synchronization variables */
    _ioc_done = 0;
    _ioc_lock = 0;

    return ret;
}

/*
 * *********************
 * VciFrameBuffer driver
 * *********************
 *
 * The '_fb_sync_write' and '_fb_sync_read' functions use a memcpy strategy to
 * implement the transfer between a data buffer (user space) and the frame
 * buffer (kernel space). They are blocking until completion of the transfer.
 * ---
 * The '_fb_write()', '_fb_read()' and '_fb_completed()' functions use the DMA
 * coprocessor to transfer data between the user buffer and the frame buffer.
 *
 * Quite similarly to the block device, these three functions use a polling
 * policy to test the global variables _dma_busy[i] and detect the transfer
 * completion.  As each processor has its private DMA, there is up to NB_PROCS
 * _dma_busy locks, that are indexed by the proc_id.
 * A _dma_busy variable is reset by the ISR associated to the DMA device IRQ.
 */

/*
 * _fb_sync_write()
 *
 * Transfer data from an memory buffer to the frame_buffer device with a
 * memcpy. The source memory buffer must be in user address space.
 * - offset : offset (in bytes) in the frame buffer.
 * - buffer : base address of the memory buffer.
 * - length : number of bytes to be transfered.
 *
 * - Returns 0 if success, > 0 if error.
 */
unsigned int _fb_sync_write(unsigned offset, const void * buffer, unsigned int length)
{
    volatile unsigned char * fb_address;

    /* parameters checking */
    /* buffer must be in user space */
    if (((unsigned int) buffer >= 0x80000000)
            || (((unsigned int) buffer + length ) >= 0x80000000 ))
        return 1;

    fb_address = (unsigned char *) &seg_fb_base + offset;

    /* buffer copy */
    memcpy((void *) fb_address, (void *) buffer, length);

    return 0;
```



```
}

/*
 * _fb_sync_read()
 *
 * Transfer data from the frame_buffer device to an memory buffer with a
 * memcpy. The destination memory buffer must be in user address space.
 * - offset : offset (in bytes) in the frame buffer.
 * - buffer : base address of the memory buffer.
 * - length : number of bytes to be transfered.
 *
 * - Returns 0 if success, > 0 if error.
 */
unsigned int _fb_sync_read(unsigned int offset, const void * buffer, unsigned int length)
{
    volatile unsigned char * fb_address;

    /* parameters checking */
    /* buffer must be in user space */
    if (((unsigned int) buffer >= 0x80000000)
            || (((unsigned int) buffer + length) >= 0x80000000))
        return 1;

    fb_address = (unsigned char *) &seg_fb_base + offset;

    /* buffer copy */
    memcpy((void *) buffer, (void *) fb_address, length);

    return 0;
}

/*
 * _fb_write()
 *
 * Transfer data from an memory buffer to the frame_buffer device using a DMA.
 * The source memory buffer must be in user address space.
 * - offset : offset (in bytes) in the frame buffer.
 * - buffer : base address of the memory buffer.
 * - length : number of bytes to be transfered.
 *
 * - Returns 0 if success, > 0 if error.
 */
unsigned int _fb_write(unsigned int offset, const void * buffer, unsigned int length)
{
    volatile unsigned char * fb_address;
    volatile unsigned int * dma;

    unsigned int proc_id;

    unsigned int delay;
    unsigned int i;

    /* parameters checking */
    /* buffer must be in user space */
    if (((unsigned int) buffer >= 0x80000000)
            || (((unsigned int) buffer + length) >= 0x80000000))
        return 1;

    proc_id = _procid();
    fb_address = (unsigned char *) &seg_fb_base + offset;
    dma = (unsigned int *) &seg_dma_base + (proc_id * DMA_SPAN);

    /* waiting until DMA device is available */
```

```
    while (_dma_busy[proc_id] != 0)
    {
        /* if the lock failed, busy wait with a pseudo random delay between bus
         * accesses */
        delay = (_proctime() & 0xF) << 4;
        for (i = 0; i < delay; i++)
            asm volatile("nop");
    }
    _dma_busy[proc_id] = 1;

    /* DMA configuration for write transfer */
    dma[DMA_IRQ_DISABLE] = 0;
    dma[DMA_SRC] = (unsigned int) buffer;
    dma[DMA_DST] = (unsigned int) fb_address;
    dma[DMA_LEN] = (unsigned int) length;
    return 0;
}

/*
 * _fb_read()
 *
 * Transfer data from the frame_buffer device to an memory buffer using a DMA.
 * The destination memory buffer must be in user address space.
 * - offset : offset (in bytes) in the frame buffer.
 * - buffer : base address of the memory buffer.
 * - length : number of bytes to be transfered.
 *
 * - Returns 0 if success, > 0 if error.
 *
 * Note: all cache lines corresponding to the the target buffer are invalidated
 * for cache coherence.
 */
unsigned int _fb_read(unsigned int offset, const void * buffer, unsigned int length)
{
    volatile unsigned char * fb_address;
    volatile unsigned int * dma;

    unsigned int proc_id;

    unsigned int delay;
    unsigned int i;

    /* parameters checking */
    /* buffer must be in user space */
    if (((unsigned int) buffer >= 0x80000000)
            || (((unsigned int) buffer + length) >= 0x80000000))
        return 1;

    proc_id = _procid();
    fb_address = (unsigned char *) &seg_fb_base + offset;
    dma = (unsigned int *) &seg_dma_base + (proc_id * DMA_SPAN);

    /* waiting until DMA device is available */
    while (_dma_busy[proc_id] != 0)
    {
        /* if the lock failed, busy wait with a pseudo random delay between bus
         * accesses */
        delay = (_proctime() & 0xF) << 4;
        for (i = 0; i < delay; i++)
            asm volatile("nop");
    }
    _dma_busy[proc_id] = 1;
```



```
    /* DMA configuration for write transfer */
    dma[DMA_IRQ_DISABLE] = 0;
    dma[DMA_SRC] = (unsigned int) fb_address;
    dma[DMA_DST] = (unsigned int) buffer;
    dma[DMA_LEN] = (unsigned int) length;

    /* invalidation of data cache */
    if (NO_HARD_CC) _dcache_buf_invalidate(buffer, length);

    return 0;
}

/*
 * _fb_completed()
 *
 * This function checks completion of a DMA transfer to or fom the frame buffer.
 *
 * As it is a blocking call, the processor is stalled until the next interrupt.
 *
 * - Returns 0 if success, > 0 if error.
 */
unsigned int _fb_completed()
{
    unsigned int proc_id;

    proc_id = _procid();

    while (_dma_busy[proc_id] != 0)
        asm volatile("nop");

    if (_dma_status[proc_id] != 0)
        return 1;

    return 0;
}
```

1.4 exc_handler.h

```
#ifndef _EXCP_HANDLER_H
#define _EXCP_HANDLER_H

/*
 * Exception Vector Table (indexed by cause register)
 *
 * 16 entries corresponding to 16 causes functions addresses
 */

typedef void (*_exc_func_t)(void);
extern const _exc_func_t _cause_vector[16];

#endif
```

1.5 exc_handler.c

```
#include <exc_handler.h>
#include <drivers.h>
#include <common.h>

/*
 * Prototypes of exception handlers.
 */

static void _cause_ukn();
static void _cause_adel();
static void _cause_ades();
static void _cause_ibe();
static void _cause_dbe();
static void _cause_bp();
static void _cause_ri();
static void _cause_cpu();
static void _cause_ovf();

extern void _int_handler();
extern void _sys_handler();


/*
 * Initialize the exception vector according to CR code
 */
const _exc_func_t _cause_vector[16] = {
    &_int_handler,  /* 0000 : external interrupt */
    &_cause_ukn,    /* 0001 : undefined exception */
    &_cause_ukn,    /* 0010 : undefined exception */
    &_cause_ukn,    /* 0011 : undefined exception */
    &_cause_adel,   /* 0100 : illegal address read exception */
    &_cause_ades,   /* 0101 : illegal address write exception */
    &_cause_ibe,    /* 0110 : instruction bus error exception */
    &_cause_dbe,    /* 0111 : data bus error exception */
    &_sys_handler,  /* 1000 : system call */
    &_cause_bp,     /* 1001 : breakpoint exception */
    &_cause_ri,     /* 1010 : illegal codop exception */
    &_cause_cpu,    /* 1011 : illegal coprocessor access */
    &_cause_ovf,    /* 1100 : arithmetic overflow exception */
    &_cause_ukn,    /* 1101 : undefined exception */
    &_cause_ukn,    /* 1110 : undefined exception */
    &_cause_ukn,    /* 1111 : undefined exception */
};

static const char * exc_message_causes[] = {
```



```
    "\n\nException : strange unknown cause\n",
    "\n\nException : illegal read address \n",
    "\n\nException : illegal write address\n",
    "\n\nException : inst bus error       \n",
    "\n\nException : data bus error       \n",
    "\n\nException : breakpoint           \n",
    "\n\nException : reserved instruction \n",
    "\n\nException : illegal coproc access\n"
    "\n\nException : arithmetic overflow  \n",
};

static void _cause(unsigned int msg_cause)
{
    char * buf = "0x00000000";

    /* print the human readable cause */
    _putk(exc_message_causes[msg_cause]);

    /* print EPC value */
    _putk("\nEPC = ");

    unsigned int epc = _get_epc();
    _itoa_hex(epc, buf + 2);
    _putk(buf);

    /* print BAR value */
    _putk("\nBAR = ");

    unsigned int bar = _get_bar();
    _itoa_hex(bar, buf + 2);
    _putk(buf);

    /* print CAUSE value */
    _putk("\nCAUSE = ");

    unsigned int cause = _get_cause();
    _itoa_hex(cause, buf + 2);
    _putk(buf);

    /* exit forever */
    _exit();
}

static void _cause_ukn()  { _cause(0); }
static void _cause_adel() { _cause(1); }
static void _cause_ades() { _cause(2); }
static void _cause_ibe()  { _cause(3); }
static void _cause_dbe()  { _cause(4); }
static void _cause_bp()   { _cause(5); }
static void _cause_ri()   { _cause(6); }
static void _cause_cpu()  { _cause(7); }
static void _cause_ovf()  { _cause(8); }
```

1.6 irq_handler.h

```
#ifndef _IRQ_HANDLER_H
#define _IRQ_HANDLER_H

/*
 * Interrupt Vector Table (indexed by interrupt index)
 *
 * 32 entries corresponding to 32 ISR addresses
 */

typedef void (*_isr_func_t)(void);
extern _isr_func_t _interrupt_vector[32];

/*
 * Prototypes of the Interrupt Service Routines (ISRs) supported by the GIET.
 * - they must be installed in reset.s
 */

void _isr_default();

void _isr_dma();

void _isr_ioc();

void _isr_timer0();
void _isr_timer1();
void _isr_timer2();
void _isr_timer3();

void _isr_tty_get();
void _isr_tty_get_task0();
void _isr_tty_get_task1();
void _isr_tty_get_task2();
void _isr_tty_get_task3();

void _isr_switch();

#endif
```

1.7 irq_handler.c

```
/*
 * These routines must be "installed" by the boot code in the interrupt vector
 * (_interrupt_vector), depending on the system architecture
 */

#include <config.h>
#include <irq_handler.h>
#include <drivers.h>
#include <common.h>
#include <ctx_handler.h>
#include <hwr_mapping.h>

/*
 * Initialize the whole interrupt vector with the default ISR
 */
_isr_func_t _interrupt_vector[32] = { [0 ... 31] = &_isr_default };

/*
 * _int_demux()
 *
 * This functions uses an external ICU component (Interrupt Controler Unit)
```



```
 * that concentrates up to 32 interrupts lines up to (NB_PROCS) IRQ lines that
 * can be connected to any of the (NB_PROCS) MIPS32 IRQ inputs.
 *
 * This component returns the highest priority active interrupt index (smaller
 * indexes have the highest priority) by reading the ICU_IT_VECTOR register.
 * Any value larger than 31 means "no active interrupt", and the default ISR
 * (that does nothing) is executed.
 *
 * The interrupt vector (32 ISR addresses array stored at _interrupt_vector
 * address) is initialised with the default ISR address. The actual ISR
 * addresses are supposed to be written in the interrupt vector array by the
 * boot code.
 */
void _int_demux(void)
{
    int interrupt_index;
    _isr_func_t isr;

    /* retrieves the highest priority active interrupt index */
    if (!_icu_read(ICU_IT_VECTOR, (unsigned int *) &interrupt_index))
    {
        /* no interrupt is active */
        if (interrupt_index > 31)
            return;

        /* call the ISR corresponding to this index */
        isr = _interrupt_vector[interrupt_index];
        isr();
    }
}

/*
 * _isr_default()
 *
 * The default ISR is called when no specific ISR has been installed in the
 * interrupt vector. It simply displays a message on TTY0.
 */
void _isr_default()
{
    _putk("\n\n!!!␣Default␣ISR␣!!!\n");
}

/*
 * _isr_dma
 *
 * This ISR acknowledges the interrupt from the dma controller, depending on
 * the proc_id. It reset the global variable _dma_busy[i] for software
 * signaling, after copying the DMA status into the _dma_status[i] variable.
 */
void _isr_dma()
{
    volatile unsigned int* dma_address;
    unsigned int proc_id;

    proc_id = _procid();
    dma_address = (unsigned int *) &seg_dma_base + (proc_id * DMA_SPAN);

    _dma_status[proc_id] = dma_address[DMA_LEN]; /* save status */
    _dma_busy[proc_id] = 0;                      /* release DMA */
    dma_address[DMA_RESET] = 0;         /* reset IRQ */
}

/*
```

```
 * _isr_ioc
 *
 * There is only one IOC controler shared by all tasks. It acknowledge the IRQ
 * using the ioc base address, save the status, and set the _ioc_done variable
 * to signal completion.
 */
void _isr_ioc()
{
    volatile unsigned int * ioc_address;

    ioc_address = (unsigned int *) &seg_ioc_base;

    _ioc_status = ioc_address[BLOCK_DEVICE_STATUS]; /* save status & reset IRQ */
    _ioc_done   = 1;                                /* signals completion */
}

/*
 * _isr_timer
 *
 * This ISR handles up to 8 IRQs generated by 8 independant timers, and
 * connected to 8 different processors. The behaviour depends on the processor
 * id: It acknowledges the IRQ on TIMER[id] and displays a message on TTY[id]
 */
void _isr_timer()
{
    volatile unsigned int * timer_address;
    unsigned int proc_id;

    proc_id = _procid();
    timer_address = (unsigned int*)&seg_timer_base + (proc_id * TIMER_SPAN);

    timer_address[TIMER_RESETIRQ] = 0; /* reset IRQ */

    _putk("\n\n!!! Interrupt timer received at cycle: ");

    char buf[] = "          ";
    int date = (int)_proctime();
    _itoa_dec(date, buf);

    _putk(buf);

    _putk("\n\n");
}

/*
 * _isr_tty_get_task* (* = 0,1,2,3)
 *
 * A single processor can run up to 4 tasks in pseudo-parallelismr, and each
 * task has is own private terminal.
 *
 * These 4 ISRs handle up to 4 IRQs associate to 4 independant terminals
 * connected to a single processor.
 *
 * It acknowledge the IRQ using the terminal basee address depending on both
 * the proc_id and the task_id (0,1,2,3).
 *
 * There is one communication buffer _tty_get_buf[tty_id] per terminal.
 * protected by a set/reset variable _tty_get_full[tty_id].
 *
 * The _tty_get_full[tty_id] synchronisation variable is set by the ISR, and
 * reset by the OS.
 *
 * To access these buffers, the terminal index is computed as
```



```
 *     tty_id = proc_id*ntasks + task_id
 * A character is lost if the buffer is full when the ISR is executed.
 */
void _isr_tty_get_indexed(unsigned int task_id)
{
    volatile unsigned int * tty_address;
    unsigned int proc_id;

    proc_id = _procid();
    tty_address = (unsigned int*)&seg_tty_base
        + (proc_id * NB_MAXTASKS * TTY_SPAN)
        + (task_id * TTY_SPAN);

    unsigned int tty_id = _procid() * NB_MAXTASKS + task_id;

    /* save character and reset IRQ */
    _tty_get_buf[tty_id] = (unsigned char)tty_address[TTY_READ];

    /* signals character available */
    _tty_get_full[tty_id] = 1;
}

void _isr_tty_get()
{
    _isr_tty_get_indexed(0);
}
void _isr_tty_get_task0()
{
    _isr_tty_get_indexed(0);
}
void _isr_tty_get_task1()
{
    _isr_tty_get_indexed(1);
}
void _isr_tty_get_task2()
{
    _isr_tty_get_indexed(2);
}
void _isr_tty_get_task3()
{
    _isr_tty_get_indexed(3);
}

/*
 * _isr_switch
 *
 * This ISR is in charge of context switch. It handles up to 4 IRQs,
 * corresponding to 4 different processors. If the processor uses several
 * timers, the context switch is driven by the IRQ associated to timer0. It
 * acknowledges the IRQ on TIMER[proc_id] and calls the _ctx_switch() function.
 */
void _isr_switch()
{
    volatile unsigned int *timer_address;
    unsigned int proc_id;

    proc_id = _procid();
    timer_address = (unsigned int *) &seg_timer_base + (proc_id * TIMER_SPAN);

    timer_address[TIMER_RESETIRQ] = 0; /* reset IRQ */
    _ctx_switch();
}
```

1.8 sys_handler.h

```
#ifndef _SYS_HANDLER_H
#define _SYS_HANDLER_H

/*
 * Syscall Vector Table (indexed by syscall index)
 *
 * 32 entries corresponding to 32 syscall handler addresses.
 *
 * No declaration of a special type for function pointer here, because syscall
 * handlers have each different prototypes.
 */

extern const void * _syscall_vector[32];

#endif
```

1.9 sys_handler.c

```
#include <sys_handler.h>
#include <drivers.h>
#include <ctx_handler.h>
#include <common.h>
#include <config.h>

/*
 * Local syscall handlers prototypes
 */
static void _sys_ukn();
static unsigned int _procnumber();

/*
 * Initialize the syscall vector with syscall handlers
 */
const void * _syscall_vector[32] = {
    &_procid,           /* 0x00 */
    &_proctime,         /* 0x01 */
    &_tty_write,        /* 0x02 */
    &_tty_read,         /* 0x03 */
    &_timer_write,      /* 0x04 */
    &_timer_read,       /* 0x05 */
    &_gcd_write,        /* 0x06 */
    &_gcd_read,         /* 0x07 */
    &_sys_ukn,          /* 0x08 */
    &_sys_ukn,          /* 0x09 */
    &_tty_read_irq,     /* 0x0A */
    &_sys_ukn,          /* 0x0B */
    &_sys_ukn,          /* 0x0C */
    &_ctx_switch,       /* 0x0D */
    &_exit,             /* 0x0E */
    &_procnumber,       /* 0x0F */
    &_fb_sync_write,    /* 0x10 */
    &_fb_sync_read,     /* 0x11 */
    &_fb_write,         /* 0x12 */
    &_fb_read,          /* 0x13 */
    &_fb_completed,     /* 0x14 */
    &_ioc_write,        /* 0x15 */
    &_ioc_read,         /* 0x16 */
    &_ioc_completed,    /* 0x17 */
    &_barrier_init,     /* 0x18 */
    &_barrier_wait,     /* 0x19 */
    &_sys_ukn,          /* 0x1A */
    &_sys_ukn,          /* 0x1B */
    &_sys_ukn,          /* 0x1C */
    &_sys_ukn,          /* 0x1D */
    &_sys_ukn,          /* 0x1E */
    &_sys_ukn,          /* 0x1F */
};

static void _sys_ukn()
{
    /* print the human readable cause */
    _putk("\n\n!!! Undefined System Call !!!\n");

    /* print EPC value */
    _putk("\nEPC = ");

    char * buf = "0x00000000";
    unsigned int epc = _get_epc();
    _itoa_hex(epc, buf + 2);
    _putk(buf);

    /* exit forever */
    _exit();
}

static unsigned int _procnumber()
{
    return NB_PROCS;
}
```

1.10 ctx_handler.h

```
#ifndef _TASK_H
#define _TASK_H

/*
 * Current running task index
 * and task context array are
 * used by the TTY driver.
 */

extern unsigned char _current_task_array[];
extern unsigned int  _task_context_array[];


/*
 * Prototype of the context switch function
 */
void _ctx_switch();

#endif
```

1.11 ctx_handler.c

```
#include <config.h>
#include <ctx_handler.h>
#include <drivers.h>

/* Size (in words) of a task context */
#define TASK_CTXT_SIZE 64

/*
 * Table of (NB_PROCS * NB_MAXTASKS) task context.
 */
unsigned int _task_context_array[NB_PROCS * NB_MAXTASKS * TASK_CTXT_SIZE];

/*
 * Current running task index on each processor.
 */
unsigned char _current_task_array[NB_PROCS] = { [0 ... NB_PROCS - 1] = 0 };

/*
 * Number of tasks on each processor.
 */
unsigned char _task_number_array[NB_PROCS] = { [0 ... NB_PROCS - 1] = 1 };

/*
 * _ctx_switch()
 *
 * This function performs a context switch between the current running task and
 * another task.
 * It can be used in a multi-processor architecture, with the assumption that
 * the tasks are statically allocated to processors.
 * The max number of processors is (NB_PROCS), and the max number of tasks is
 * (NB_MAXTASKS).
 * The scheduling policy is round-robin : for each processor, the task index is
 * incremented, modulo the number of tasks allocated to the processor.
 *
 * The function has no argument, and no return value.
 *
 * It uses three global variables:
 * - _current_task_array :  an array of (NB_PROCS) task index:
 *    index of the task actually running on each processor
```



```
 * - _task_number_array : an array of (NB_PROCS) numbers:
 *    the number of tasks allocated to each processor
 * - _task_context_array : an array of (NB_PROCS * NB_MAXTASKS) task contexts:
 *    at most 8 processors / each processor can run up to 4 tasks
 *
 * Caution : This function is intended to be used with periodic interrupts.  It
 * can be directly called by the OS, but interrupts must be disabled before
 * calling.
 */

extern void _task_switch(unsigned int *, unsigned int *);

void _ctx_switch()
{
    unsigned char curr_task_index;
    unsigned char next_task_index;

    unsigned int * curr_task_context;
    unsigned int * next_task_context;

    unsigned int proc_id;

    proc_id = _procid();

    /* first, test if there is more than one task to schedule on the processor.
     * otherwise, let's just return. */
    if (_task_number_array[proc_id] <= 1)
        return;

    /* find the task context of the currently running task */
    curr_task_index = _current_task_array[proc_id];
    curr_task_context = &_task_context_array[(proc_id * NB_MAXTASKS + curr_task_index)
        * TASK_CTXT_SIZE];

    /* find the task context of the next running task (using a round-robin
     * policy) */
    next_task_index = (curr_task_index + 1) % _task_number_array[proc_id];
    next_task_context = &_task_context_array[(proc_id * NB_MAXTASKS + next_task_index)
        * TASK_CTXT_SIZE];

    /* before doing the task switch, update the _current_task_array with the
     * new task index */
    _current_task_array[proc_id] = next_task_index;

    /* now, let's do the task switch */
    _task_switch(curr_task_context, next_task_context);
}
```

1.12 common.h

```
/*
 * Commonly used functions
 */

#ifndef _COMMON_H
#define _COMMON_H

/*
 * Prototypes of common functions
 */
unsigned int _putk(const char * msg);
void _exit() __attribute__((noreturn));
void _dcache_buf_invalidate(const void * buffer, unsigned int size);

void _itoa_dec(unsigned int val, char * buf);
void _itoa_hex(unsigned int val, char * buf);

unsigned int _barrier_init(unsigned int index, unsigned int count);
unsigned int _barrier_wait(unsigned int index);

/*
 * memcpy function
 *
 * This function is likely not to be called directly but GCC can automatically
 * issue call to it during compilation so we must provide it. 'static inline'
 * so the function's code is directly included when used.
 *
 * Code taken from MutekH.
 */
static inline void * memcpy(void * _dst, const void * _src, unsigned int size)
{
    unsigned int * dst = _dst;
    const unsigned int * src = _src;

    /* if source and destination buffer are word-aligned,
     * then copy word-by-word */
    if (!((unsigned int) dst & 3) && !((unsigned int) src & 3))
    {
        while (size > 3)
        {
            *dst++ = *src++;
            size -= 4;
        }
    }

    unsigned char * cdst = (unsigned char *) dst;
    unsigned char * csrc = (unsigned char *) src;

    /* byte-by-byte copy */
    while (size--)
    {
        *cdst++ = *csrc++;
    }
    return _dst;
}

/*
 * ---
 * MIPS32 related helpers
 * ---
 */
```



```
/*
 * _get_epc()
 *
 * Access CP0 and returns EPC register.
 */
static inline unsigned int _get_epc()
{
    unsigned int ret;
    asm volatile("mfc0␣%0,␣$14" : "=r" (ret));
    return ret;
}

/*
 * _get_bar()
 *
 * Access CP0 and returns BAR register.
 */
static inline unsigned int _get_bar()
{
    unsigned int ret;
    asm volatile("mfc0␣%0,␣$8" : "=r" (ret));
    return ret;
}

/*
 * _get_cause()
 *
 * Access CP0 and returns CAUSE register.
 */
static inline unsigned int _get_cause()
{
    unsigned int ret;
    asm volatile("mfc0␣%0,␣$13" : "=r" (ret));
    return ret;
}

#if 0
/*
 *  _it_mask()
 * Access CP0 and mask IRQs
 */

static inline void _it_mask()
{
    asm volatile(
            "mfc0␣␣$2,␣$12␣␣␣␣\n"
            "ori␣␣␣$2,␣$2,␣1␣␣␣\n"
            "mtc0␣␣$2,␣$12␣␣␣␣\n"
            ::: "$2"
            );
}

/*
 *  _it_enable()
 * Access CP0 and enable IRQs
 */
static inline void _it_enable()
{
    asm volatile(
            "mfc0␣␣$2,␣$12␣␣␣␣\n"
            "addi␣␣$2,␣$2,␣-1␣␣\n"
            "mtc0␣␣$2,␣$12␣␣␣␣\n"
```

```
            ::: "$2"
            );
}
#endif

#endif
```

1.13 common.c

```
#include <common.h>
#include <drivers.h>

/*
 * _putk()
 *
 * Print a message with _tty_write after calculating its length.
 */
unsigned int _putk(const char * msg)
{
    unsigned int len = 0;
    const char * tmp = msg;

    while (*tmp++)
        len++;

    return _tty_write(msg, len);
}

/*
 * _exit()
 *
 * Exit (suicide) after printing a death message on a terminal.
 */
void _exit()
{
    char buf[40] = "\n\n!!! Exit Processor 0x___ !!!\n";

    unsigned int proc_id = _procid();

    /* proc_id can be up to 0x3FF so display three digits */
    buf[23] = (char)((proc_id >> 8) & 0xF) + 0x30;
    buf[24] = (char)((proc_id >> 4) & 0xF) + 0x30;
    buf[25] = (char)((proc_id >> 0) & 0xF) + 0x30;

    _putk(buf);

    /* infinite loop */
    while (1)
        asm volatile("nop");
}

/*
 * _dcache_buf_invalidate()
 *
 * Invalidate all data cache lines corresponding to a memory buffer (identified
 * by an address and a size).
 */
void _dcache_buf_invalidate(const void * buffer, unsigned int size)
{
    unsigned int i;
    unsigned int tmp;
    unsigned int line_size;
```



```
    /*
     * compute data cache line size based on config register (bits 12:10)
     */
    asm volatile("mfc0 %0, $16, 1" : "=r" (tmp));
    tmp = ((tmp >> 10) & 0x7);
    line_size = 2 << tmp;

    /* iterate on cache lines to invalidate each one of them */
    for (i = 0; i < size; i += line_size)
    {
        asm volatile(
                " cache %0, %1"
                ::"i" (0x11), "R" (*((unsigned char *) buffer + i))
                );
    }
}

/*
 * _itoa_dec()
 *
 * Convert a 32-bit unsigned integer to a string of ten decimal characters.
 */
void _itoa_dec(unsigned int val, char * buf)
{
    const static char dectab[] = "0123456789";
    unsigned int i;

    for (i = 0; i < 10; i++)
    {
        if ((val != 0) || (i == 0))
            buf[9 - i] = dectab[val % 10];
        else
            buf[9 - i] = 0x20;
        val /= 10;
    }
}

/*
 * _itoa_hex()
 *
 * Convert a 32-bit unsigned integer to a string of height hexadecimal
 * characters.
 */
void _itoa_hex(unsigned int val, char *buf)
{
    const static char hexatab[] = "0123456789ABCD";
    unsigned int i;

    for (i = 0; i < 8; i++)
    {
        buf[7 - i] = hexatab[val % 16];
        val /= 16;
    }
}

/*
 * Barrier related uncachable variables
 */

#define in_unckdata __attribute__((section (".unckdata")))

#define MAX_BARRIER_COUNT 8
```

```
in_unckdata unsigned int volatile _barrier_initial_value[MAX_BARRIER_COUNT] = {
    [0 ... MAX_BARRIER_COUNT - 1] = 0
};
in_unckdata unsigned int volatile _barrier_count[MAX_BARRIER_COUNT] = {
    [0 ... MAX_BARRIER_COUNT - 1] = 0
};

/*
 * _barrier_init()
 *
 * This function makes a cooperative initialisation of the barrier: several
 * tasks can try to initialize the barrier, but the initialisation is done by
 * only one task, using LL/SC instructions.
 */

unsigned int _barrier_init(unsigned int index, unsigned int value)
{
    /* check the index */
    if (index >= MAX_BARRIER_COUNT)
        return 1;

    unsigned int * pinit = (unsigned int *) &_barrier_initial_value[index];
    unsigned int * pcount = (unsigned int *) &_barrier_count[index];

    /* parallel initialisation using atomic instructions LL/SC */
    asm volatile ("_barrier_init_test:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
                  "ll␣␣␣$2,␣␣␣␣␣0(%0)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n" /* read initial value */
                  "bnez␣$2,␣␣␣␣␣_barrier_init_done␣␣␣␣␣␣\n"
                  "move␣$3,␣␣␣␣␣%2␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
                  "sc␣␣␣$3,␣␣␣␣␣0(%0)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n" /* try to write initial value *
                  "beqz␣$3,␣␣␣␣␣_barrier_init_test␣␣␣␣␣␣\n"
                  "move␣$3,␣␣␣␣␣%2␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
                  "sw␣␣␣$3,␣␣␣␣␣0(%1)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n" /* write count */
                  "_barrier_init_done:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
                  :: "r" (pinit), "r" (pcount), "r" (value)
                  : "$2", "$3");
    return 0 ;
}

/*
 * _barrier_wait()
 *
 * This blocking function decrements a barrier's counter and then uses a
 * busy_wait mechanism for synchronization, because the GIET does not support
 * dynamic scheduling/descheduling of tasks.
 *
 * There is at most MAX_BARRIER_COUNT independant barriers, and an error is
 * returned if the barrier index is larger than MAX_BARRIER_COUNT.
 */
unsigned int _barrier_wait(unsigned int index)
{
    if (index >= MAX_BARRIER_COUNT)
        return 1;

    unsigned int * pcount = (unsigned int *) &_barrier_count[index];
    unsigned int maxcount = _barrier_initial_value[index];
    unsigned int count;

    /* parallel decrement barrier counter using atomic instructions LL/SC
     * - input : pointer on the barrier counter
     * - output : counter value
     */
    asm volatile ("_barrier_decrement:␣␣␣␣␣␣␣␣␣␣␣\n"
```



```
                  "ll␣␣␣%0,␣0(%1)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
                  "addi␣$3,␣%0,␣␣␣␣␣-1␣␣␣␣␣␣␣␣␣␣\n"
                  "sc␣␣␣$3,␣0(%1)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
                  "beqz␣$3,␣_barrier_decrement␣␣\n"
                  : "=&r" (count)
                  : "r" (pcount)
                  : "$2", "$3");

    /* the last task re-initializes the barrier counter to the max value,
     * waking up all other waiting tasks
     */

    if (count == 1)
        /* last task */
        *pcount = maxcount;
    else
        /* other tasks busy-wait for the re-initialization */
        while (*pcount != maxcount);

    return 0;
}
```

1.14 hwr_mapping.h

```
#ifndef _HWR_MAPPING_H
#define _HWR_MAPPING_H

/*
 * Registers mapping for the different peripherals
 */

/* IOC (block device) */
enum IOC_registers {
    BLOCK_DEVICE_BUFFER,
    BLOCK_DEVICE_LBA,
    BLOCK_DEVICE_COUNT,
    BLOCK_DEVICE_OP,
    BLOCK_DEVICE_STATUS,
    BLOCK_DEVICE_IRQ_ENABLE,
    BLOCK_DEVICE_SIZE,
    BLOCK_DEVICE_BLOCK_SIZE,
};
enum IOC_operations {
    BLOCK_DEVICE_NOOP,
    BLOCK_DEVICE_READ,
    BLOCK_DEVICE_WRITE,
};
enum IOC_status{
    BLOCK_DEVICE_IDLE,
    BLOCK_DEVICE_BUSY,
    BLOCK_DEVICE_READ_SUCCESS,
    BLOCK_DEVICE_WRITE_SUCCESS,
    BLOCK_DEVICE_READ_ERROR,
    BLOCK_DEVICE_WRITE_ERROR,
    BLOCK_DEVICE_ERROR,
};

/* DMA */
enum DMA_registers {
    DMA_SRC         = 0,
    DMA_DST         = 1,
    DMA_LEN         = 2,
    DMA_RESET       = 3,
    DMA_IRQ_DISABLE = 4,
    /**/
    DMA_END         = 5,
    DMA_SPAN        = 8,
};

/* GCD */
enum GCD_registers {
    GCD_OPA     = 0,
    GCD_OPB     = 1,
    GCD_START   = 2,
    GCD_STATUS  = 3,
    /**/
    GCD_END     = 4,
};

/* ICU */
enum ICU_registers {
    ICU_INT         = 0,
    ICU_MASK        = 1,
    ICU_MASK_SET    = 2,
    ICU_MASK_CLEAR  = 3,
```



```
    ICU_IT_VECTOR   = 4,
    /**/
    ICU_END         = 5,
    ICU_SPAN        = 8,
};

/* TIMER */
enum TIMER_registers {
    TIMER_VALUE     = 0,
    TIMER_MODE      = 1,
    TIMER_PERIOD    = 2,
    TIMER_RESETIRQ  = 3,
    /**/
    TIMER_SPAN      = 4,
};

/* TTY */
enum TTY_registers {
    TTY_WRITE   = 0,
    TTY_STATUS  = 1,
    TTY_READ    = 2,
    TTY_CONFIG  = 3,
    /**/
    TTY_SPAN    = 4,
};

#endif
```

2 Code de la bibliothèque utilisateur

2.1 stdio.h

```
#ifndef _STDIO_H
#define _STDIO_H

/*
 * These functions implements a minimal C library
 */

/* MIPS32 related functions */
unsigned int procid();
unsigned int proctime();
unsigned int procnumber();

/* TTY device related functions */
unsigned int tty_putc(char byte);
unsigned int tty_puts(char *buf);
unsigned int tty_putw(unsigned int val);
unsigned int tty_getc(char *byte);
unsigned int tty_getc_irq(char *byte);
unsigned int tty_gets_irq(char *buf, unsigned int bufsize);
unsigned int tty_getw_irq(unsigned int *val);
unsigned int tty_printf(char *format,...);

/* Timer device related functions */
unsigned int timer_set_mode(unsigned int mode);
unsigned int timer_set_period(unsigned int period);
unsigned int timer_reset_irq();
unsigned int timer_get_time(unsigned int *time);

/* GCD coprocessor related functions */
unsigned int gcd_set_opa(unsigned int val);
unsigned int gcd_set_opb(unsigned int val);
unsigned int gcd_start();
unsigned int gcd_get_result(unsigned int *val);
unsigned int gcd_get_status(unsigned int *val);

/* Block device related functions */
unsigned int ioc_read(unsigned int lba, void *buffer, unsigned int count);
unsigned int ioc_write(unsigned int lba, void *buffer, unsigned int count);
unsigned int ioc_completed();

/* Frame buffer device related functions */
unsigned int fb_sync_read(unsigned int offset, void *buffer, unsigned int length);
unsigned int fb_sync_write(unsigned int offset, void *buffer, unsigned int length);
unsigned int fb_read(unsigned int offset, void *buffer, unsigned int length);
unsigned int fb_write(unsigned int offset, void *buffer, unsigned int length);
unsigned int fb_completed();

/* Software barrier related functions */
unsigned int barrier_init(unsigned int index, unsigned int count);
unsigned int barrier_wait(unsigned int index);

/* Misc */
void exit();
unsigned int rand();
unsigned int ctx_switch();

/*
 * memcpy function
```



```
 *
 * This function is likely not to be called directly but GCC can automatically
 * issue call to it during compilation so we must provide it. 'static inline'
 * so the function's code is directly included when used.
 *
 * Code taken from MutekH.
 */
static inline void *memcpy(void *_dst, const void *_src, unsigned int size)
{
    unsigned int *dst = _dst;
    const unsigned int *src = _src;

    /* if source and destination buffer are word-aligned,
     * then copy word-by-word */
    if (!((unsigned int)dst & 3) && !((unsigned int)src & 3))
        while (size > 3) {
            *dst++ = *src++;
            size -= 4;
        }

    unsigned char *cdst = (unsigned char*)dst;
    unsigned char *csrc = (unsigned char*)src;

    /* byte-by-byte copy */
    while (size--) {
        *cdst++ = *csrc++;
    }
    return _dst;
}

#endif
```

2.2 stdio.c

```
#include <stdarg.h>
#include <stdio.h>

#define SYSCALL_PROCID          0x00
#define SYSCALL_PROCTIME        0x01
#define SYSCALL_TTY_WRITE       0x02
#define SYSCALL_TTY_READ        0x03
#define SYSCALL_TIMER_WRITE     0x04
#define SYSCALL_TIMER_READ      0x05
#define SYSCALL_GCD_WRITE       0x06
#define SYSCALL_GCD_READ        0x07
#define SYSCALL_TTY_READ_IRQ    0x0A
#define SYSCALL_TTY_WRITE_IRQ   0x0B
#define SYSCALL_CTX_SWITCH      0x0D
#define SYSCALL_EXIT            0x0E
#define SYSCALL_PROCNUMBER      0x0F
#define SYSCALL_FB_SYNC_WRITE   0x10
#define SYSCALL_FB_SYNC_READ    0x11
#define SYSCALL_FB_WRITE        0x12
#define SYSCALL_FB_READ         0x13
#define SYSCALL_FB_COMPLETED    0x14
#define SYSCALL_IOC_WRITE       0x15
#define SYSCALL_IOC_READ        0x16
#define SYSCALL_IOC_COMPLETED   0x17
#define SYSCALL_BARRIER_INIT    0x18
#define SYSCALL_BARRIER_WAIT    0x19

/*
 * sys_call()
```

```
 *
 * This generic C function is used to implement all system calls.
 */
static inline unsigned int sys_call(unsigned int call_no,
        unsigned int arg_0, unsigned int arg_1, unsigned int arg_2, unsigned int arg_3)
{
    register unsigned int reg_no_and_output asm("v0") = call_no;
    register unsigned int reg_a0 asm("a0") = arg_0;
    register unsigned int reg_a1 asm("a1") = arg_1;
    register unsigned int reg_a2 asm("a2") = arg_2;
    register unsigned int reg_a3 asm("a3") = arg_3;

    asm volatile(
            "syscall"
            : "=r" (reg_no_and_output)  /* output argument */
            : "r" (reg_a0),             /* input arguments */
            "r" (reg_a1),
            "r" (reg_a2),
            "r" (reg_a3),
            "r" (reg_no_and_output)
            : "memory",
            /* These persistant registers will be saved on the stack by the
             * compiler only if they contain relevant data. */
            "at",
            "v1",
            "ra",
            "t0",
            "t1",
            "t2",
            "t3",
            "t4",
            "t5",
            "t6",
            "t7",
            "t8",
            "t9"
               );
    return reg_no_and_output;
}

/*
 * ******************************
 * MIPS32 related system calls
 * ******************************
 */

/*
 * procid()
 *
 * This function returns the processor identifier.
 */
unsigned int procid()
{
    return sys_call(SYSCALL_PROCID, 0, 0, 0, 0);
}

/*
 * proctime()
 *
 * This function returns the local processor time (elapsed clock cycles since
 * bootup).
 */
unsigned int proctime()
```



```
{
    return sys_call(SYSCALL_PROCTIME, 0, 0, 0, 0);
}

/*
 * procnumber()
 *
 * This function returns the number of processors controlled by the system.
 */
unsigned int procnumber()
{
    return sys_call(SYSCALL_PROCNUMBER, 0, 0, 0, 0);
}

/*
 * ******************************
 * TTY device related system calls
 * ******************************
 */

/*
 * tty_putc()
 *
 * This function displays a single ascii character on a terminal.
 * - The terminal index is implicitely defined by the processor identifier (and
 *   by the task ID in case of multi-tasking).
 * - It doesn't use the TTY_PUT_IRQ interrupt, and the associated kernel
 *   buffer.
 * - Returns 1 if the character has been written, 0 otherwise.
 */
unsigned int tty_putc(char byte)
{
    return sys_call(SYSCALL_TTY_WRITE,
            (unsigned int) &byte,
            1,
            0, 0);
}

/*
 * tty_puts()
 *
 * This function displays a string on a terminal.
 * - The terminal index is implicitely defined by the processor identifer (and
 *   by the task ID in case of multi-tasking).
 * - The string must be terminated by a NUL character.
 * - It doesn't use the TTY_PUT_IRQ interrupt, and the associated kernel
 *   buffer.
 * - Returns the number of written characters.
 */
unsigned int tty_puts(char * buf)
{
    unsigned int length = 0;
    while (buf[length] != 0)
    {
        length++;
    }
    return sys_call(SYSCALL_TTY_WRITE,
            (unsigned int) buf,
            length,
            0, 0);
}

/*
```

```
 * tty_putw()
 *
 * This function displays the value of a 32-bit word with decimal characters.
 * - The terminal index is implicitely defined by the processor identifier (and
 *   by pthe task ID in case of multi-tasking).
 * - It doesn't use the TTY_PUT_IRQ interrupt, and the associated kernel
 *   buffer.
 * - Returns the number of written characters (should be equal to ten).
 */
unsigned int tty_putw(unsigned int val)
{
    char buf[10];
    unsigned int i;
    for (i = 0; i < 10; i++)
    {
        buf[9 - i] = (val % 10) + 0x30;
        val = val / 10;
    }
    return sys_call(SYSCALL_TTY_WRITE,
            (unsigned int) buf,
            10,
            0,0);
}

/*
 * tty_getc()
 *
 * This blocking function fetches a single ascii character from a terminal.
 * - The terminal index is implicitely defined by the processor identifier (and
 *   by the task ID in case of multi-tasking)
 * - It doesn't use the IRQ_GET interrupt, and the associated kernel buffer.
 * - Returns necessarily 0 when completed.
 */
unsigned int tty_getc(char * byte)
{
    unsigned int ret = 0;
    while (ret == 0)
    {
        ret = sys_call(SYSCALL_TTY_READ,
                (unsigned int) byte,
                1,
                0, 0);
    }
    return 0;
}

/*
 * tty_getc_irq()
 *
 * This blocking function fetches a single ascii character from a terminal.
 * - The terminal index is implicitely defined by the processor identifier (and
 *   by the task ID in case of multi-tasking).
 * - It uses the IRQ_GET interrupt, and the associated kernel buffer.
 * - Returns necessarily 0 when completed.
 */
unsigned int tty_getc_irq(char * byte)
{
    unsigned int ret = 0;
    while (ret == 0)
    {
        ret = sys_call(SYSCALL_TTY_READ_IRQ,
                (unsigned int) byte,
                1,
```



```
                0, 0);
    }
    return 0;
}

/*
 * tty_gets_irq()
 *
 * This blocking function fetches a string from a terminal to a bounded length
 * buffer.
 * - The terminal index is implicitely defined by the processor identifier (and
 *   by the task ID in case of multi-tasking)
 * - It uses the TTY_GET_IRQ interrupt, anf the associated kernel buffer.
 * - Returns necessarily 0 when completed.
 *
 * - Up to (bufsize - 1) characters (including the non printable characters)
 *   will be copied into buffer, and the string is always completed by a NUL
 *   character.
 * - The <LF> character is interpreted, as the function close the string with a
 *   NUL character if <LF> is read.
 * - The <DEL> character is interpreted, and the corresponding character(s) are
 *   removed from the target buffer.
 */
unsigned int tty_gets_irq(char * buf, unsigned int bufsize)
{
    unsigned int ret;
    unsigned char byte;
    unsigned int index = 0;

    while (index < (bufsize - 1))
    {
        do {
            ret = sys_call(SYSCALL_TTY_READ_IRQ,
                    (unsigned int) &byte,
                    1,
                    0, 0);
        } while (ret != 1);

        if (byte == 0x0A)
            break; /* LF */
        else if ((byte == 0x7F) && (index > 0))
            index--; /* DEL */
        else
        {
            buf[index] = byte;
            index++;
        }
    }
    buf[index] = 0;
    return 0;
}

/*
 * tty_getw_irq()
 *
 * This blocking function fetches a string of decimal characters (most
 * significant digit first) to build a 32-bit unsigned integer.
 * - The terminal index is implicitely defined by the processor identifier (and
 *   by the task ID in case of multi-tasking).
 * - It uses the TTY_GET_IRQ interrupt, anf the associated kernel buffer.
 * - Returns necessarily 0 when completed.
 *
 * - The non-blocking system function _tty_read_irq is called several times,
```

```
 *   and the decimal characters are written in a 32 characters buffer until a
 *   <LF> character is read.
 * - The <DEL> character is interpreted, and previous characters can be
 *   cancelled. All others characters are ignored.
 * - When the <LF> character is received, the string is converted to an
 *   unsigned int value. If the number of decimal digit is too large for the 32
 *   bits range, the zero value is returned.
 */
unsigned int tty_getw_irq(unsigned int * val)
{
    unsigned char buf[32];
    unsigned char byte;
    unsigned int save = 0;
    unsigned int dec = 0;
    unsigned int done = 0;
    unsigned int overflow = 0;
    unsigned int max = 0;
    unsigned int i;
    unsigned int ret;

    while (done == 0)
    {
        do {
            ret = sys_call(SYSCALL_TTY_READ_IRQ,
                    (unsigned int) &byte,
                    1,
                    0, 0);
        } while (ret != 1);

        if ((byte > 0x2F) && (byte < 0x3A)) /* decimal character */
        {
            buf[max] = byte;
            max++;
            tty_putc(byte);
        }
        else if ((byte == 0x0A) || (byte == 0x0D)) /* LF or CR character */
        {
            done = 1;
        }
        else if (byte == 0x7F) /* DEL character */
        {
            if (max > 0)
            {
                max--; /* cancel the character */
                tty_putc(0x08);
                tty_putc(0x20);
                tty_putc(0x08);
            }
        }
        if (max == 32) /* decimal string overflow */
        {
            for (i = 0; i < max; i++) /* cancel the string */
            {
                tty_putc(0x08);
                tty_putc(0x20);
                tty_putc(0x08);
            }
            tty_putc(0x30);
            *val = 0; /* return 0 value */
            return 0;
        }
    }
```



```
    /* string conversion */
    for (i = 0; i < max; i++)
    {
        dec = dec * 10 + (buf[i] - 0x30);
        if (dec < save)
            overflow = 1;
        save = dec;
    }

    /* check overflow */
    if (overflow == 0)
    {
        *val = dec; /* return decimal value */
    }
    else
    {
        for (i = 0; i < max; i++) /* cancel the string */
        {
            tty_putc(0x08);
            tty_putc(0x20);
            tty_putc(0x08);
        }
        tty_putc(0x30);
        *val = 0; /* return 0 value */
    }
    return 0;
}

/*
 * tty_printf()
 *
 * This function is a simplified version of the mutek_printf() function.
 * - The terminal index is implicitely defined by the processor identifier (and
 *   by the task ID in case of multi-tasking).
 * - It doesn't use the IRQ_PUT interrupt, anf the associated kernel buffer.
 * - Only a limited number of formats are supported:
 *  - %d : signed decimal
 *  - %u : unsigned decimal
 *  - %x : hexadecimal
 *  - %c : char
 *  - %s : string
 *
 * - Returns 0 if success, > 0 if error.
 */
unsigned int tty_printf(char * format, ...)
{
    va_list ap;
    va_start(ap, format);
    unsigned int ret;

printf_text:

    while (*format)
    {
        unsigned int i;
        for (i = 0; format[i] && format[i] != '%'; i++)
            ;
        if (i)
        {
            ret = sys_call(SYSCALL_TTY_WRITE,
                    (unsigned int) format,
                    i,
                    0, 0);
```

```
            if (ret != i)
                return 1; /* return error */
            format += i;
        }
        if (*format == '%')
        {
            format++;
            goto printf_arguments;
        }
    }

    va_end(ap);
    return 0;

printf_arguments:

    {
        int val = va_arg(ap, long);
        char buf[20];
        char * pbuf;
        unsigned int len = 0;
        unsigned int i;
        static const char HexaTab[] = "0123456789ABCDEF";

        switch (*format++)
        {
            case ('c'):             /* char conversion */
                len = 1;
                buf[0] = val;
                pbuf = buf;
                break;
            case ('d'):             /* decimal signed integer */
                if (val < 0)
                {
                    val = -val;
                    ret = sys_call(SYSCALL_TTY_WRITE,
                            (unsigned int) "-",
                            1,
                            0, 0);
                    if (ret != 1)
                        return 1; /* return error */
                }
            case ('u'):             /* decimal unsigned integer */
                for (i = 0; i < 10 ; i++)
                {
                    buf[9 - i] = HexaTab[val % 10];
                    if (!(val /= 10))
                        break;
                }
                len =  i + 1;
                pbuf = &buf[9 - i];
                break;
            case ('x'):             /* hexadecimal integer */
                ret = sys_call(SYSCALL_TTY_WRITE,
                        (unsigned int) "0x",
                        2,
                        0, 0);
                if (ret != 2)
                    return 1; /* return error */
                for (i = 0; i < 8; i++)
                {
                    buf[7 - i] = HexaTab[val % 16U];
                    if (!(val /= 16U))
```



```
                        break;
                }
                len =  i + 1;
                pbuf = &buf[7 - i];
                break;
            case ('s'):             /* string */
                {
                    char * str = (char *) val;
                    while (str[len])
                        len++;
                    pbuf = (char *) val;
                }
                break;
            default:
                goto printf_text;
        }

        ret = sys_call(SYSCALL_TTY_WRITE,
                (unsigned int) pbuf,
                len,
                0, 0);
        if (ret != len)
            return 1;
        goto printf_text;
    }
}

/*
 * ***********************************
 * Timer device related system calls
 * ***********************************
 */

#define TIMER_VALUE     0
#define TIMER_MODE      1
#define TIMER_PERIOD    2
#define TIMER_RESETIRQ  3

/*
 * timer_set_mode()
 *
 * This function defines the operation mode of a timer. The possible values for
 * this mode are:
 * - 0x0 : Timer not activated
 * - 0x1 : Timer activated, but no interrupt is generated
 * - 0x3 : Timer activarted and periodic interrupts generated
 *
 * - Returns 0 if success, > 0 if error.
 */
unsigned int timer_set_mode(unsigned int val)
{
    return sys_call(SYSCALL_TIMER_WRITE,
            TIMER_MODE,
            val,
            0, 0);
}

/*
 * timer_set_period()
 *
 * This function defines the period value of a timer to enable a periodic
 * interrupt.
 * - Returns 0 if success, > 0 if error.
```

```
 */
unsigned int timer_set_period(unsigned int val)
{
    return sys_call(SYSCALL_TIMER_WRITE,
            TIMER_PERIOD,
            val,
            0, 0);
}

/*
 * timer_reset_irq()
 *
 * This function resets the interrupt signal issued by a timer.
 * - Returns 0 if success, > 0 if error.
 */
unsigned int timer_reset_irq()
{
    return sys_call(SYSCALL_TIMER_WRITE,
            TIMER_RESETIRQ,
            0, 0, 0);
}

/*
 * timer_get_time()
 *
 * This function returns the current timing value of a timer.
 * - Returns 0 if success, > 0 if error.
 */
unsigned int timer_get_time(unsigned int * time)
{
    return sys_call(SYSCALL_TIMER_READ,
            TIMER_VALUE,
            (unsigned int)time,
            0, 0);
}

/*
 * ************************************************************
 * GCD (Greatest Common Divisor) device related system calls
 * ************************************************************
 */

#define GCD_OPA     0
#define GCD_OPB     1
#define GCD_START   2
#define GCD_STATUS  3

/*
 * gcd_set_opa()
 *
 * This function sets the operand A in the GCD coprocessor.
 * - Returns 0 if success, > 0 if error.
 */
unsigned int gcd_set_opa(unsigned int val)
{
    return sys_call(SYSCALL_GCD_WRITE,
            GCD_OPA,
            val,
            0, 0);
}

/*
 * gcd_set_opb()
```



```
 *
 * This function sets operand B in the GCD coprocessor.
 * - Returns 0 if success, > 0 if error.
 */
unsigned int gcd_set_opb(unsigned int val)
{
    return sys_call(SYSCALL_GCD_WRITE,
            GCD_OPB,
            val,
            0, 0);
}

/*
 * gcd_start()
 *
 * This function starts the computation in the GCD coprocessor.
 * - Returns 0 if success, > 0 if error.
 */
unsigned int gcd_start()
{
    return sys_call(SYSCALL_GCD_WRITE,
            GCD_START,
            0, 0, 0);
}

/*
 * gcd_get_status()
 *
 * This function gets the status fromn the GCD coprocessor.
 * - The value is equal to 0 when the coprocessor is idle (computation
 *   completed).
 */
unsigned int gcd_get_status(unsigned int * val)
{
    return sys_call(SYSCALL_GCD_READ,
            GCD_STATUS,
            (unsigned int) val,
            0, 0);
}

/*
 * gcd_get_result()
 *
 * This function gets the result of the computation from the GCD coprocessor.
 */
unsigned int gcd_get_result(unsigned int * val)
{
    return sys_call(SYSCALL_GCD_READ,
            GCD_OPA,
            (unsigned int) val,
            0, 0);
}

/*
 * ***********************************
 * Block device related system calls
 * ***********************************
 */

/*
 * ioc_write()
 *
 * Transfer data from a memory buffer to a file on the block_device.
```

```
 * - lba    : Logical Block Address (first block index)
 * - buffer : base address of the memory buffer
 * - count  : number of blocks to be transfered
 *
 * - Returns 0 if success, > 0 if error (e.g. memory buffer not in user space).
 */
unsigned int ioc_write(unsigned int lba, void * buffer, unsigned int count)
{
    return sys_call(SYSCALL_IOC_WRITE,
            lba,
            (unsigned int) buffer,
            count,
            0);
}

/*
 * ioc_read()
 *
 * Transfer data from a file on the block_device to a memory buffer.
 * - lba    : Logical Block Address (first block index)
 * - buffer : base address of the memory buffer
 * - count  : number of blocks to be transfered
 *
 * - Returns 0 if success, > 0 if error (e.g. memory buffer not in user space).
 */
unsigned int ioc_read(unsigned int lba, void * buffer, unsigned int count)
{
    return sys_call(SYSCALL_IOC_READ,
            lba,
            (unsigned int) buffer,
            count,
            0);
}

/*
 * ioc_completed()
 *
 * This blocking function returns 0 when the I/O transfer is
 * successfully completed, and returns 1 if an address error
 * has been detected.
 */
unsigned int ioc_completed()
{
    return sys_call(SYSCALL_IOC_COMPLETED,
            0, 0, 0, 0);
}

/*
 * ****************************************
 * Frame buffer device related system calls
 * ****************************************
 */

/*
 * fb_sync_write()
 *
 * This blocking function use a memory copy strategy to transfer data from a
 * user buffer to the frame buffer device in kernel space.
 * - offset : offset (in bytes) in the frame buffer
 * - buffer : base address of the memory buffer
 * - length : number of bytes to be transfered
 *
 * - Returns 0 if success, > 0 if error (e.g. memory buffer not in user space).
```



```
 */
unsigned int fb_sync_write(unsigned int offset, void * buffer, unsigned int length)
{
    return sys_call(SYSCALL_FB_SYNC_WRITE,
            offset,
            (unsigned int) buffer,
            length,
            0);
}

/*
 * fb_sync_read()
 *
 * This blocking function use a memory copy strategy to transfer data from the
 * frame buffer device in kernel space to an user buffer.
 * - offset : offset (in bytes) in the frame buffer
 * - buffer : base address of the user buffer
 * - length : number of bytes to be transfered
 *
 * - Returns 0 if success, > 0 if error (e.g. memory buffer not in user space).
 */
unsigned int fb_sync_read(unsigned int offset, void * buffer, unsigned int length)
{
    return sys_call(SYSCALL_FB_SYNC_READ,
            offset,
            (unsigned int) buffer,
            length,
            0);
}

/*
 * fb_write()
 *
 * This non-blocking function use the DMA coprocessor to transfer data from a
 * user buffer to the frame buffer device in kernel space.
 * - offset : offset (in bytes) in the frame buffer
 * - buffer : base address of the user buffer
 * - length : number of bytes to be transfered
 *
 * - Returns 0 if success, > 0 if error (e.g. memory buffer not in user space).
 *
 * The transfer completion is signaled by an IRQ, and must be tested by the
 * fb_completed() function.
 */
unsigned int fb_write(unsigned int offset, void * buffer, unsigned int length)
{
    return sys_call(SYSCALL_FB_WRITE,
            offset,
            (unsigned int) buffer,
            length,
            0);
}

/*
 * fb_read()
 *
 * This non-blocking function use the DMA coprocessor to transfer data from the
 * frame buffer device in kernel space to an user buffer.
 * - offset : offset (in bytes) in the frame buffer
 * - buffer : base address of the memory buffer
 * - length : number of bytes to be transfered
 *
 * - Returns 0 if success, > 0 if error (e.g. memory buffer not in user space).
```

```
 *
 * The transfer completion is signaled by an IRQ, and must be tested by the
 * fb_completed() function.
 */
unsigned int fb_read(unsigned int offset, void * buffer, unsigned int length)
{
    return sys_call(SYSCALL_FB_READ,
            offset,
            (unsigned int) buffer,
            length,
            0);
}

/*
 * fb_completed()
 *
 * This blocking function returns when the transfer is completed.
 * - Returns 0 if success, > 0 if error.
 */
unsigned int fb_completed()
{
    return sys_call(SYSCALL_FB_COMPLETED,
            0, 0, 0, 0);
}

/*
 * ***************************************
 * Software barrier related system calls
 * ***************************************
 */

/*
 * barrier_init()
 *
 * This function initializes the counter for barrier[index].
 * - index  : index of the barrier (between 0 & 7)
 * - count  : number of tasks to be synchronized.
 * The GIET supports up to 8 independant barriers.
 *
 * - Returns 0 if success, > 0 if error (e.g. index >= 8).
 */
unsigned int barrier_init(unsigned int index, unsigned int count)
{
    return sys_call(SYSCALL_BARRIER_INIT,
            index,
            count,
            0, 0);
}

/*
 * barrier_wait()
 *
 * This blocking function use a busy waiting policy, and returns only when all
 * synchonized asks have reached the barrier.
 * - index  : index of the barrier (between 0 & 7)
 * The GIET supports up to 8 independant barriers.
 *
 * - Returns 0 if success, > 0 if error (e.g. index >= 8).
 */
unsigned int barrier_wait(unsigned int index)
{
    return sys_call(SYSCALL_BARRIER_WAIT,
            index,
```



```
            0, 0, 0);
}

/*
 * ****************************
 * Miscellaneous system calls
 * ****************************
 */

/*
 * exit()
 *
 * This function exits the program with a TTY message,
 * and enter an infinite loop.
 * The task is blocked until the next RESET,
 * but it still consume processor cycles ...
 */
void exit()
{
    unsigned int proc_index = procid();
    sys_call(SYSCALL_EXIT, proc_index, 0, 0, 0);
}

/*
 * rand()
 *
 * This function returns a pseudo-random value derived from the processor cycle
 * count. This value is comprised between 0 & 65535.
 */
unsigned int rand()
{
    unsigned int x = sys_call(SYSCALL_PROCTIME, 0, 0, 0, 0);
    if ((x & 0xF) > 7)
        return (x * x & 0xFFFF);
    else
        return (x * x * x & 0xFFFF);
}

/*
 * ctx_switch()
 *
 * The user task calling this function is descheduled and
 * the processor is allocated to another task.
*/
unsigned int ctx_switch()
{
    return sys_call(SYSCALL_CTX_SWITCH, 0, 0, 0, 0);
}
```