S.Ha.R.K. User Manual

Scuola Superiore di Studi e Perfezionamento S. Anna
ReTiS Lab

Volume II

PROGRAMMING LIBRARIES

Written by
Paolo Gai (pj@sssup.it)

and based on the Hartik's Manual written by
Giorgio Buttazzo (giorgio@sssup.it)
Luigi Palopoli (luigi@gandalf.sssup.it)
Luca Abeni (luca@gandalf.sssup.it)
Giuseppe Lipari (lipari@gandalf.sssup.it)
Gerardo Lamastra (lamastra@sssup.it)
Antonino Casile (casile@sssup.it)
Massimiliano Giorgi (massy@gandalf.sssup.it)

RETIS Lab.
Scuola Superiore S. Anna
Via Carducci, 40 - 56100 Pisa

16th December 2004

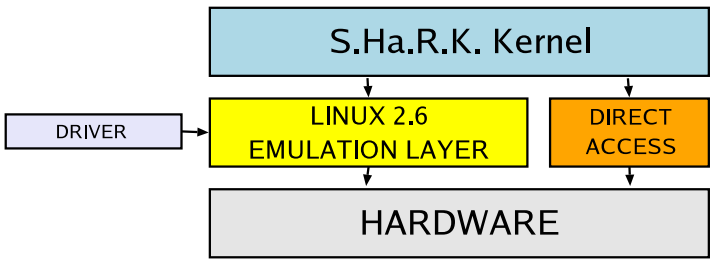
Contents

Chapter 1

Introduction

Device drivers are a critical part of Real-Time Systems. Trying to fit an IRQ and a timer handler, coming from a device, inside a task context, it is a priority for OS like S.Ha.R.K. If we design these drivers considering possible preemptions, exectution times and other Real-Time contraints, a schedulability test can guarantie our system. Buf if we must reuse a source code from third-party drivers makers, without having no knowledge about the driver timing behaviour, with possible non-preemptable critical section, it is very difficult to impose contraints for a schedulability analisys.

The S.Ha.R.K. drivers are mainly ported from Linux 2.6. We projected and tested a Linux 2.6 Emulation Layer. This Layer gives an indipendent environement, where it is possible to compile the original drivers without conflicts with S.Ha.R.K. and OSlib. An interface glue-code allows to access the drivers API.



The Emulation Layer needs a specific kernel module to run. This module has two important objectives:

- To create an high priority execution context for the drivers IRQ and timer handlers.
- To maintain the drivers behaviour inside specific Real-Time constraints, avoiding that a possible malfuncition or resource abuse cause a system failure.

Both of these points are guaranteed through the INTDRIVE module.

As described inside the module documentation, INTDRIVE is an high priority module specifically designed to handle the drivers requests. Usign the hierarchial capabilities of S.Ha.R.K., the INTDRIVE module is indepented from the main system scheduler, so EDF, RM and all the other modules can be used to schedule the application tasks.

To load the Emulation Layer, INTDRIVE must be present inside the system.

```
TIME      __kernel_register_levels__(void *arg)
```

```
{
INTDRIVE_register_level(INTDRIVE_Q,INTDRIVE_T,INTDRIVE_FLAG);
EDF_register_level(EDF_ENABLE_ALL);
}
```

After the INTDRIVE registration, the initialization procedure can begin.

1.1 Driver Initialization

The initialization of a drivers should be done inside the `initfile`, before entering inside the real-time application. During initialization the devices can lock the system for an unpredictable amount of time so any used device should be ready before the real-time tasks are scheduled.

S.Ha.R.K. standard `initfile` suggests a possible implementation:

```
TASK      __init__(void *arg) {
          struct multiboot_info *mb = (struct multiboot_info *)arg;

          set_shutdown_task();

          device_drivers_init();

          sys_atrunlevel(call_shutdown_task, NULL, RUNLEVEL_SHUTDOWN);

          __call_main__(mb);

          return NULL;
}

int       device_drivers_init() {
          KEYB_PARMS kparms = BASE_KEYB;

          LINUXC26_register_module();
          PCI26_init();
          INPUT26_init();
          KEYB26_init(&kparms);
          return 0;
}
```

The **device_drivers_init()** calls all the initialization functions. LINUXC26 is the Linux Emulation Layer and it must be loaded as first. The initialization order follows the driver dependence tree of figure 1.1.If this order is not respected, the initialization procedure fails.

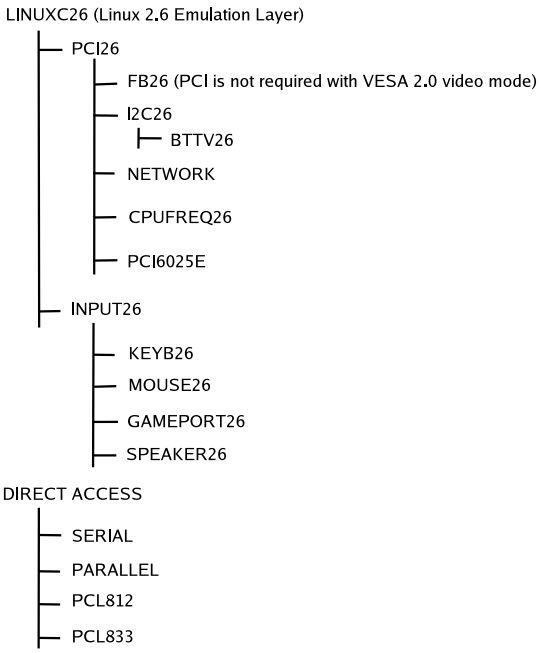


Figure 1.1: Drivers dependeces tree

1.2 Driver Shutdown

Another critical point is the shutdown sequence. When a `sys_end()` or `exit()`is called the system should close the device drivers as soon as possible. Unfortunately the driver shutdown must be executed when the kernel is still in multitasking mode.

```
int       device_drivers_close() {
          KEYB26_close();
          INPUT26_close();
          return 0;

}
```

The RUNLEVELS of S.Ha.R.K. gives the possibility to implement a safe and trasparent procedure to shutdown the system without compromise the drivers stability. As the initiliazation, the shutdown sequence must follow the dependence order.

The flow chart of figure 1.2 shows the steps to reach a safe exit. Anyway if the system is overloaded during the exit procedure, the shutdown task cannot be scheduled. A possible solution is to give high priority to the shutdown task, or to design an application that doesn't reach the CPU saturation. If the shutdown task doesn't execute a timeout (default 3 seconds) force the system exit.

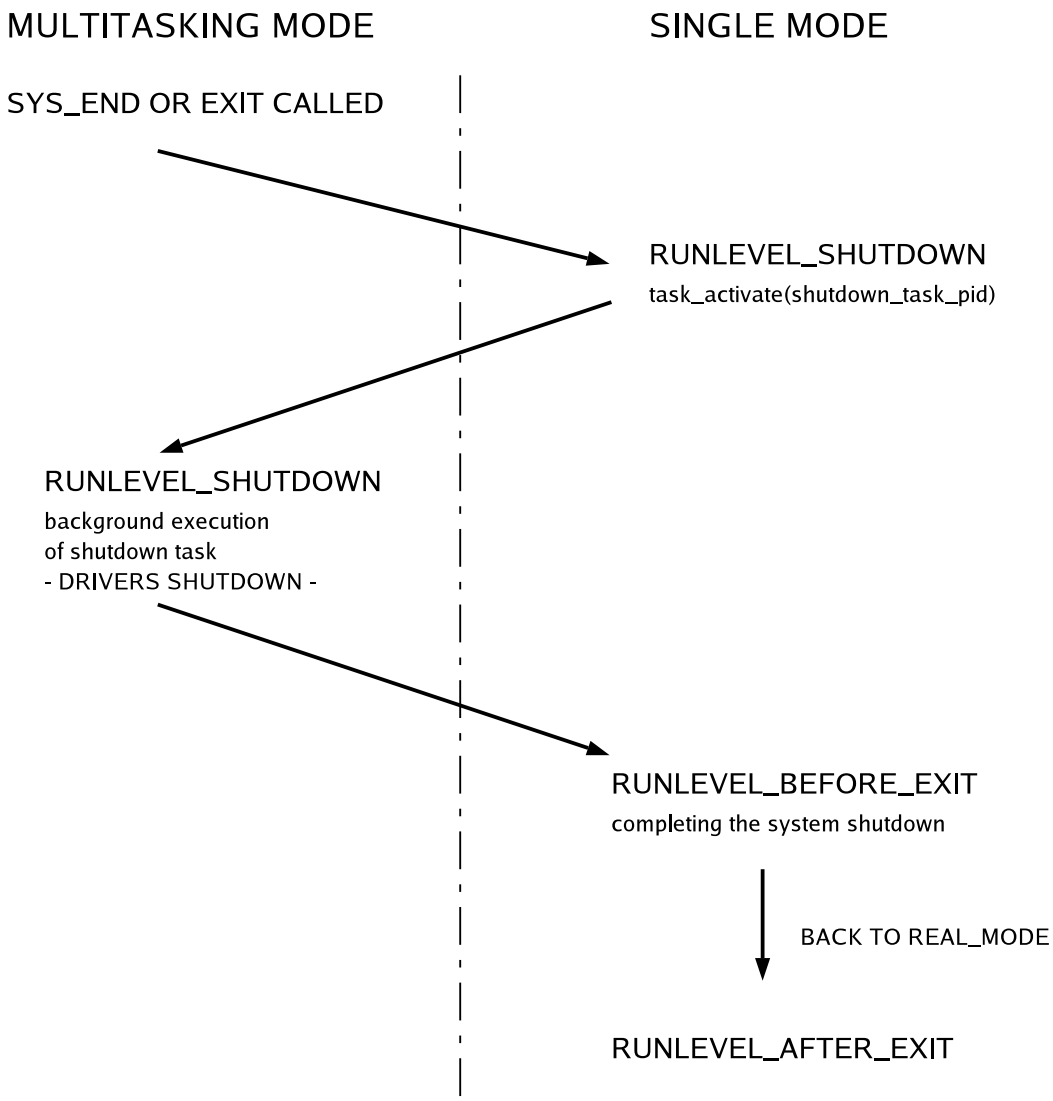


Figure 1.2: Shutdown sequence

Chapter 2

The Linux Compatibility Layer

Is the glue code used for the interaction between the kernel and the linux code. Is mandatory to use drivers ported from the linux 2.6 kernel tree.

LINUXC26_INIT

```
void LINUXC26_init(void);
```

Description: It initializes the compatibility layer interface and the library's internal data structures.

Example

```
int device_drivers_init()
{
        int res;
        KEYB_PARMS kparms = BASE_KEYB;

        LINUXC26_register_module();

        PCI26_init();

        INPUT26_init();

        keyb_def_ctrlC(kparms, NULL);
        KEYB26_init(&kparms);

        FB26_init();
        res = FB26_open(FRAME_BUFFER_DEVICE);
        if (res) {
                cprintf("Error: Cannot open graphical mode\n");
                KEYB26_close();
                INPUT26_close();
                sys_end();
        }
        FB26_use_grx(FRAME_BUFFER_DEVICE);
        FB26_setmode(FRAME_BUFFER_DEVICE,"640x480-16");

        return 0;
}
```

Chapter 3

The Input Library

This library allow the user to interact with applications. Is composed by a lower lever which must be initialized at the beginning. On top of this layer we can find different peripherals:

- Keyboard;
- Mouse;
- Joystick;
- Speaker;
- Event debuger.

Each one can work independently from the others. In order to use the low lever functions, the files `drivers/shark_input26.h` must be included. It contains the prototypes of the declared functions. First of all, the keyboard needs be initialized by the calling `KEYB26_init` primitive into the `__init__` function in the initialization file or in any other point of the application code.

INPUT26_INIT

```
int INPUT26_init(void);
```

Description: It initializes the low lever input interface and the library's internal data structures.

Return value: 0 if the operation is performed successfully; a value less than 0, otherwise.

The following code shows an example of input drivers initialization for the system:

Example:

```
int res;
KEYB_PARMS kparms = BASE_KEYB;
MOUSE_PARMS mparms = BASE_MOUSE;

LINUXC26_register_module();

INPUT26_init();
```

```
keyb_def_map(kparms, KEYMAP_IT);
keyb_def_ctrlC(kparms, NULL);
KEYB26_init(&kparms);

mouse_def_threshold(mparms, 5);
mouse_def_xmin(mparms, 0);
mouse_def_ymin(mparms, 0);
mouse_def_xmax(mparms, 639);
mouse_def_ymax(mparms, 479);
MOUSE26_init(&mparms);

SPEAK26_init();

JOY26_init();
...
```

The event debugger is used to have an output of the raw data coming from an input device for which a driver is not present. It can be stanter, stopper and is possible to control the actual status.

EVBUG26_INIT

```
int KEYB26_init(void);
```

Description: It initializes the event debugger interface and the library's internal data structures.

Return value: 0 if the operation is performed successfully; a value less than 0, otherwise.

EVBUG26_CLOSE

```
int EVBUG26_close(void);
```

Description: It close the event debugger interface.

Return value: 0 if the operation is performed successfully; -1 if the interface in not installed.

EVBUG26_INSTALLED

```
int EVBUG26_installed(void);
```

Description: Return if the event debugger is actually installed.

Return value: 0 if the module is installed; 1 otherwise.

3.1 The keyboard library

In order to use the keyboard handling functions, the `drivers/shark_keyb26.h` header file, containing the interface functions' prototypes, has to be included in the application program.

First of all, the keyboard needs be initialized by the calling `KEYB26_init` primitive into the `__init__` function in the initialization file or in any other point of the application code and the input low level driver must be already installed. A programmer can either initialize the keyboard using the default settings or define his own parameters which are encapsulated into a structure having

KEY_PARMS type. The strucuture can be initialized with the default set of values by setting it equal to BASE_KEYB; the keyb_def_... macros can be used before calling KEYB26_init to modify each setting. Afterwards, it is possible to read the ASCII code of any stroken key by calling the keyb_getch() function. This function requires a parameter which determines whether the task should block until a key is hit or not. In the latter case, if no key has been hit, the function returns 0 (behaving like kbhit()).

If we are interested in the key's scan code, we can call the keyb_getcode() function, which returns a struct containing either the scan code or the ascii code and a byte containing information on the ALT, SHIFT or CTRL key being pressed.

The following code shows an example of usage for the function:

Example:

```
KEY_EVT k;
...
if (keyb_getcode(&k, NON_BLOCK) {
    if (isRightCtrl(k) || isLeftCtrl(k)) &&
         (k.ascii == 'x')) {
         /* Ctrl + 'x' has been pressed */
         ...
    }
}
...
```

Finally, it is possible to define a function to be automatically called every time a specific key (or a specific combination of keys) is pressed. This is done by calling the keyb_hook() function, which receives as arguments a data structure containing the required combination of keys and the function to be called.

KEYB26_INIT

```
int KEYB26_init(KEYB_PARMS *parms);
```

Description: It initializes the keyboard interface and the library's internal data structures. It can be called using NULL as parms to initialize the keyboard interface to default values.
Note that to be proper initialized, you need also to initialize the HARTPORT modules.

Return value: 0 if the operation is performed successfully; a value less than 0, otherwise.

KEYB26_CLOSE

```
int KEYB26_close(void);
```

Description: It close the keyboard interface.

Return value: 0 if the operation is performed successfully; -1 if the keyboard in not installed.

KEYB26_INSTALLED

```
int KEYB26_installed(void);
```

Description: Return if the keyboard driver is actually installed.

Return value: 0 if the keyboard is installed; 1 otherwise.

KEYB_DEFAULT_PARM

```
void keyb_default_parm(KEYB_PARMS parms);
```

Description: It changes the values of parms to the same vales as the BASE_KEYB default initializer.

KEYB_DEF_MAP

```
void keyb_def_map(KEYB_PARMS parms, unsigned char map);
```

Description: It changes the default map used for the keyboard: `map` can be `KEYMAP_US` for an english keyboard or `KEYMAP_IT` for an italian keyboard. The default is english keyboard layout.

KEYB_DEF_CTRLC

```
void keyb_def_ctrlC(KEYB_PARMS parms, void (*ctrlcfunc)(KEY_EVT *k));
```

Description: It enables the execution of a function when the "ctrlC" combination is pressed. By default, if this macro is not used, the `ctrlC` combination results in calling `sys_end()`. Note that a small message is printed also on the console. The message is only visible if the system is in text mode. If you are running a graphic application, remember to redefine the Ctrl-C Handler!

KEYB_DEF_TASK

```
void keyb_def_task(KEYB_PARMS parms, TASK_MODEL * m);
```

Description: It specifies the parameters of the keyboard server. The `TASK_MODEL *` should be a valid pointer to a Task Model, or `KEYB_DEFAULT` if you want to specify the default behaviour.
This macro should be used every time the default server Task Model does not adapt well to the configuration of the scheduling modules registered in the system.
The default server Task Model is equivalent to this initialization:

```
soft_task_default_model(base_m);
soft_task_def_wcet(base_m,2000);
soft_task_def_met(base_m,800);
soft_task_def_period(base_m,25000);
soft_task_def_system(base_m);
soft_task_def_nokill(base_m);
soft_task_def_aperiodic(base_m);
```

KEYB_GETCH, KEYB_GETCHAR

```
int keyb_getch(BYTE wait);
```

```
int keyb_getchar(void); (macro)
```

Description: If the keyboard queue is not empty, `keyb_getch()` returns the ASCII code of the pressed key. If the queue is empty, the function's behaviour depends on the value of the `wait` parameter: if it is `BLOCK`, then the calling task is blocked until a key is pressed; if it is `NON_BLOCK`, the function returns 0. `keyb_getchar()` is a macro for `keyb_getch(BLOCK)`.

Return value: the ASCII code of the pressed key, if the buffer is not empty; 0 otherwise.

KEYB_GETCODE

```
int keyb_getcode(KEY_EVT *k, BYTE wait);
```

Description: It fetches the `KEY_EVT` from the keyboard's queue and copies it into the structure pointed by `k`. If the queue is empty, the function behaves as `key_getch()`.

Return value: 1 if a key was pressed, 0 otherwise.

KEYB_HOOK

```
void keyb_hook(KEY_EVT key, void (*hook)(KEY_EVT *keypressed), unsigned char lock);
```

Description: Whenever the key combination specified in `key` is pressed, the function `hook()` is invoked, getting `key` as input parameter. If lock is set to `FALSE` after executing the function `hook()`the key will be lost, otherwise it will be inserted in the queue.

Example:

```
#include <drivers/shark_keyb26.h>

void hook_func(KEY_EVT *keypressed)
{
  switch (keypressed->ascii) {
  case 'w':  /* if 'CTRL-w' is pressed...  */
    ...
    break;
  case 'x':
    if (isLeftAlt(keypressed)||isRightAlt(keypressed))
    { /* if 'ALT-x' is pressed...  */
      ...
    } else {
      /* if 'x' is pressed...  */
      ...
   }
  }
}

int main(int argc,char *argv[])
{
  KEY_EVT key;
  /* keyboard initialization */
  /* to hook 'CTRL-w' key */
  key.ascii = 'w';
  key.scan = KEY_W;
  key.status = KEY_PRESSED;
  key.flag = CNTR_BIT;
  keyb_hook(key,hook_func,FALSE);
  /* to hook key 'x' */
  key.ascii = 'x';
  key.scan = KEY_X;
  key.status = KEY_PRESSED;
  key.flag = 0;
  keyb_hook(key,hook_func,FALSE);
  /* to hook 'ALT-x' key */
  key.flag=ALTL_BIT|ALTR_BIT;
```

```
    keyb_hook(key,hook_func,FALSE);
    ...
  }
```

KEYB_DISABLE

```
void keyb_disable(void);
```

Description: Trow away the data arriving from the hardware instead of processing them inside the driver.

KEYB_ENABLE

```
void keyb_enable(void);
```

Description: Allow the driver to receive data from the hardware.

KEYB_SET_MAP

```
int keyb_set_map(unsigned char map);
```

Description: It changes the default map used for the keyboard: `map` can be `KEYMAP_US` for an english keyboard or `KEYMAP_IT` for an italian keyboard.

Return value: the keyboard map identifier effectively applied.

KEYB_GET_MAP

```
int keyb_get_map(void);
```

Description: Return the identifier of the keyboard map actually in use.

KEY_EVT

Description: it is a data structure containing the following fields:

ascii: ascii code of the key;

scan: scan code of the key;

status: the key can be pressed, repeated or released. When used to set an hook more than one status can be selected.
The `status` field can be accessed by one of the following macros, whose usage is self-explaining:

- isPressed(&k)
- isRepeated(&k)
- isReleased(&k)

flag: codes of the ALT, SHIFT and CTRL keys.
The `flag` field can be accessed by one of the following macros, whose usage is self-explaining:

- isScanCode(&k), decides whether the hit key has an ASCII code or not;
- isLeftShift(&k)
- isRightShift(&k)
- isLeftAlt(&k)
- isRightAlt(&k)
- isLeftCtrl(&k)
- isRightCtrl(&k)

3.2 The mouse library

To use the mouse into an application program, the user must call the `MOUSE26_init` function. Then, all mouse functions are available until a call to the `MOUSE26_close` function. The initialization of the mouse library is performed by `MOUSE26_init`, which requires a parameter of `MOUSE_PARMS` type to initialize the mouse. The following example shows a possible mouse initialization:

```
int main(int argc,char *argv[])
{
 int result;
  MOUSE_PARMS params = BASE_MOUSE;

  result=MOUSE26_init(&params);
  if (result!=0) {
    // the mouse can't be initialized
  }
  // other mouse functions
  MOUSE26_close();
}
```

The `MOUSE26_close` function is not required but can be used to release all the hardware resources that the library acquires. The NULL constant can be passed to the `MOUSE26_init()` function for a default initialization. The `params` variable can be used to change the default setting of the initialization procedure using some macros, whose names start with `mouse_def_` . All the mouse functions can be found in the include file `drivers/shark_mouse26.h`.

MOUSE26_INIT

```
int MOUSE26_init(KEYB_PARMS *parms);
```

Description: It initializes the mouse interface and the library's internal data structures. It can be called using NULL as `parms` to initialize the mouse interface to default values.
Note that to be proper initialized, you need also to initialize the HARTPORT modules.

Return value: 0 if the operation is performed successfully; a value less than 0, otherwise.

MOUSE26_CLOSE

```
int MOUSE26_close(void);
```

Description: It close the mouse interface.

Return value: 0 if the operation is performed successfully; -1 if the mouse in not installed.

MOUSE26_INSTALLED

```
int MOUSE26_installed(void);
```

Description: Return if the mouse driver is actually installed.

Return value: 0 if the mouse is installed; 1 otherwise.

MOUSE_DEFAULT_PARM

```
void mouse_default_parm(KEYB_PARMS parms);
```

Description: It changes the values of parms to the same vales as the BASE_MOUSE default initializer.

MOUSE_DEF_THRESHOLD

```
void mouse_def_threshold(MOUSE_PARMS parms, int thr);
```

Description: It changes the default threshold (i.e., the mouse sensitivity) value used in the mouse driver. Is a scaling factor between the hardware position increment and the logical increment in the mouse position. The higher the value, the lower the sensitivity. The default is 10.

MOUSE_DEF_X0, MOUSE_DEF_Y0, MOUSE_DEF_Z0

```
void mouse_def_x0(MOUSE_PARMS parms, int xvalue);

void mouse_def_y0(MOUSE_PARMS parms, int yvalue);

void mouse_def_z0(MOUSE_PARMS parms, int zvalue);
```

Description: These functions change the initial value of the mouse position. The default vilue for each parameter is 0 if permitted by mouse position bounds.

MOUSE_DEF_XMIN, MOUSE_DEF_XMAX

```
void mouse_def_xmin(MOUSE_PARMS parms, int minvalue);

void mouse_def_xmax(MOUSE_PARMS parms, int maxvalue);
```

Description: It changes the minimum and maximum allowed value for the x coordinate.

MOUSE_DEF_YMIN, MOUSE_DEF_YMAX

```
void mouse_def_ymin(MOUSE_PARMS parms, int minvalue);

void mouse_def_ymax(MOUSE_PARMS parms, int maxvalue);
```

Description: It changes the minimum and maximum allowed value for the y coordinate.

MOUSE_DEF_TASK

```
void mouse_def_task(MOUSE_PARMS parms, TASK_MODEL * m);
```

Description: This macro defines the parameters for the mouse handling task. The `TASK_MODEL *` should be a valid pointer to a Task Model, or `MOUSE_DEFAULT` if you want to specify the default behaviour.
This macro should be used every time the default Task Model does not adapt well to the configuration of the scheduling modules registered in the system.
The default Task Model is equivalent to this initialization:

```
soft_task_default_model(base_m);
soft_task_def_wcet(base_m,2000);
soft_task_def_met(base_m,500);
soft_task_def_period(base_m,8000);
soft_task_def_system(base_m);
soft_task_def_nokill(base_m);
soft_task_def_aperiodic(base_m);
```

MOUSE_ENABLE

```
void mouse_enable(void);
```

Description: Allow the driver to receive data from the hardware.

MOUSE_DISABLE

```
void mouse_disable(void);
```

Description: This function disable the mouse; the driver stop to respond to the data arriving from the hardware.

MOUSE_SETPOSITION

```
void mouse_setposition(int x, int y, int z);
```

Description: Set values for axes and wheel. Values for x and y axis are compared against bounds for the allowed zone.

MOUSE_GETPOSITION

```
void mouse_getposition(int *x, int *y, int *z, unsigned long *buttons);
```

Description: Get values for axes, wheel and buttons. In the `buttons` variable each bit rappresent the status of a button.

MOUSE_SETLIMITS

```
void mouse_setlimits(int xmin, int ymin, int xmax, int ymax);
```

Description: It changes the minimum and maximum allowed value for x and y coordinate.

MOUSE_GETLIMITS

```
void mouse_getlimits(int *xmin, int *ymin, int *xmax, int *ymax);
```

Description: Allow to obtain the minimum and maximum permited value for x and y coordinate.

MOUSE_SETTHRESHOLD

```
void mouse_setthreshold(int th);
```

Description: It changes the threshold value used in the mouse driver. Is a scaling factor between the hardware position increment and the logical increment in the mouse position. The default is 10.

MOUSE_GETTHRESHOLD

```
void mouse_getthreshold(int th);
```

Description: Return the threshold value used in the mouse driver. Is a scaling factor between the hardware position increment and the logical increment in the mouse position. The default is 10.

MOUSE_HOOK

`void mouse_hook(MOUSE_HANDLER hook);`

Description: Whenever the driver receive data from the hardware the function `hook()` is invoked with a `MOUSE_EVT` as parameter. To detach a previously defined hook with NULL as parameter.

MOUSE_EVT

Description: it is a data structure containing the following fields:

x,y,z: actual mouse position;

dx,dy,dz: increments from the last event;

buttons: each bit in the parameter rappresent the status of one button of the mouse.

The following code shows an example of usage for the function `mouse_hook` and the structure `MOUSE_EVT`:

Example:

```
#include <drivers/shark_mouse26.h>

void hook_func(MOUSE_EVT *mevt)
{
  if (mevt->button&0x1){
    /* The 1st button is pressed...  */
    ...
  }
  if ( (mevt->dx > 10) || (mevt->dx > 10) ) {
    /* Increment on x or y axis major than 10...  */
    ...
  }
  if (mevt->dz > 0) {
    /* Wheel position changed...  */
    ...
  }
}

int main(int argc,char *argv[])
{
  /* mouse hook initialization */
  mouse_hook(hook_func);
  ...
}
```

ISLEFTBUTTON, ISCENTRALBUTTON, ISRIGHTBUTTON

`isLeftButton(int button)`

`isCentralButton(int button)`

`isRightButton(int button)`

Description: These macros are used to test whether a specific mouse button is pressed.

3.2.1 The mouse graphics functions

The system provide a set of function to quickly draw the mouse cursor both in text and graphic mode. For each mode we can find 2 functions. The first used to set the cursor shape and the second to enable/disable the cursor visualization.

ATTENTION: In text mode the mouse position is taken considering characters, while in graphics values are pixels as mining.

MOUSE_TXTCURSOR,MOUSE_GRXCURSOR

```
void mouse_txtcursor(int cmd);

void mouse_grxcursor(int cmd, int bpp);
```

Description: This function enables/disables the textual/grafical autocursor features of the library; `cmd` can be: `DISABLE` (disable cursor), `ENABLE` (enable a cursor). The graphic version need and other parameter `bpp` which is depth of the screen in bytes. These commands can be composite with two flags:

- `AUTOOFF`: if a user mouse handler is called, then during this call the mouse is off (if you use a `mouse_off()` you can hang the mouse task).
- `WITHOUTSEM`: the autocursor mouse functions are not protected by a semaphore (so tasks cannot be blocked, but garbage can be displayed).

MOUSE_TXTSHAPE

```
void mouse_txtshape(DWORD shape);
```

Description: This function defines the shape of the text cursor; "shape" is a DWORD that is used to display the cursor: the text character and attributes are taken from the text memory, these values are and-ed with the low word of `shape` and the result is xor-ed with the high word of `shape`, the result is written into the text memory. For examples, the default mouse cursor shape is 0x7700ffff; the character and attribute that are taken from the the memory are and-ed with 0xffff, so they do not change, and then are xor-ed with 0x7700: the character is xor-ed with 0x00 and the attribute byte is xor-ed with 0x77; the character remains the same but the attribute byte is inverted; so the mouse cursor is displayed inverting the colors of the character (the attribute byte contains the character foreground and background color and is coded in the usually EGA/VGA mode; you can read a book about VGA display adapters to find more informations).

MOUSE_GRXSHAPE

```
void mouse_grxshape(BYTE *img, BYTE *mask, int bpp);
```

Description: This function defines the shape of the graphical cursor; `img` is a pointer to an image of 16x16 that can be used with the `grx_putimage()` function, or can be `NULL` (in this case a default cursor is used); before putting the image into the screen's memory, the image is and-ed with `mask`. The parameter `bpp` is depth of the screen in bytes.

Example:
```
/*
 * the resolution is 640x480 with 2 BYTE for pixel
 */
```

```
/* WHITE,RED,GREEN and MAGENTA are defined into <cons.h>*/

#define W rgb16(255,255,255)
#define R rgb16(255,  0,  0)
#define G rgb16(  0,255,  0)
#define M rbg16(255,  0,255)

/* mouse shape */

WORD my_mouse_cursor[16*16]= {
   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
   0,W,W,W,W,0,0,0,0,0,0,W,W,W,W,0,
   0,W,M,0,0,0,0,0,0,0,0,0,0,M,W,0,
   0,W,0,M,0,0,0,0,0,0,0,0,M,0,W,0,
   0,W,0,0,M,0,0,0,0,0,0,M,0,0,W,0,
   0,0,0,0,0,M,0,0,0,0,M,0,0,0,0,0,
   0,0,0,0,0,0,G,G,G,G,0,0,0,0,0,0,
   0,0,0,0,0,0,G,0,0,G,0,0,0,0,0,0,
   0,0,0,0,0,0,G,0,0,G,0,0,0,0,0,0,
   0,0,0,0,0,0,G,0,0,G,0,0,0,0,0,0,
   0,0,0,0,0,0,G,G,G,G,0,0,0,0,0,0,
   0,0,0,0,0,0,M,M,M,M,0,0,0,0,0,0,
   0,0,0,0,0,0,M,M,M,M,0,0,0,0,0,0,
   0,0,0,0,0,M,M,M,M,M,M,0,0,0,0,0,
   0,0,0,0,M,M,M,M,M,M,M,M,0,0,0,0,
   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
};

/* mask for pixels (2 bytes in this case) */
/* 0xffff means that the background image is used */
/* 0x0000 means tha the background image is cleared */
/*        prior to draw the mouse cursor */

#define F 0xffff
#define B 0x0000

/* mouse mask */
WORD my_mouse_mask[16*16]= {
   B,B,B,B,B,B,F,F,F,F,B,B,B,B,B,B,
   B,0,0,0,0,B,F,F,F,F,B,0,0,0,0,B,
   B,0,0,B,B,F,F,F,F,F,B,B,B,0,0,B,
   B,0,B,0,B,F,F,F,F,F,F,B,0,B,0,B,
   B,0,B,B,0,B,F,F,F,F,B,0,B,B,0,B,
   B,B,B,F,B,0,B,B,B,B,0,B,F,B,B,B,
   F,F,F,F,F,B,0,0,0,0,B,F,F,F,F,F,
   F,F,F,F,F,B,0,B,B,0,B,F,F,F,F,F,
   F,F,F,F,F,B,0,B,B,0,B,F,F,F,F,F,
   F,F,F,F,F,B,0,B,B,0,B,F,F,F,F,F,
   F,F,F,F,F,B,0,0,0,0,B,F,F,F,F,F,
   F,F,F,F,F,B,0,0,0,0,B,F,F,F,F,F,
   F,F,F,F,F,B,0,0,0,0,B,F,F,F,F,F,
```

```
    F,F,F,F,B,0,0,0,0,0,0,B,F,F,F,F,
    F,F,F,B,0,0,0,0,0,0,0,0,B,F,F,F,
    F,F,F,B,B,B,B,B,B,B,B,B,F,F,F,
};

int main(int argc, char *argv[])
{
  MOUSE_PARMS mouse = BASE_MOUSE;

  /*
   * mouse initialization
   */
  MOUSE26_init(&mouse);
  /* a resolution of 640x480 is used */
  mouse_setlimit(XMINLIMIT(640,480),
      YMINLIMIT(640,480),
      XMAXLIMIT(640,480),
      YMAXLIMIT(640,480));
  /* initial position */
  mouse_setposition(320,280);
  /* mouse threshold */
  mouse_setthreshold(2);
  /* my new mouse shape */
  mouse_grxshape(my_mouse_cursor,my_mouse_mask, 2);
  /* automatic graphics mouse cursor enabled */
  mouse_grxcursor(ENABLE, 2);
  /* mouse displayed into the screen */
  mouse_on();

  ...
}
```

MOUSE_OFF, MOUSE_ON

```
void mouse_off(void);

void mouse_on(void);
```

Description: This functions disables and enables the display of the mouse cursor on the screen (the mouse events are handled).

Warning: These functions must be called before/after calling any graphic function that modifies the screen.

3.3 The joystick library

In order to use the gameport handling functions, the `drivers/shark_joy26.h` header file, containing the interface functions' prototypes, has to be included in the application program.

First of all, the joystick needs be initialized by the calling `JOY26_init` primitive into the `__init__` function in the initialization file or in any other point of the application code.

ATTENTION: The driver only work with gameport that emulate the "old SoundBlaster" gameport interface. Other kind of gameports will not be identified.

JOY26_INIT

```
int JOY26_init(void);
```

Description: It initializes the joystick interface and the library's internal data structures.

Return value: 0 if the operation is performed successfully; a value less than 0, otherwise.

JOY26_CLOSE

```
int MOUSE26_close(void);
```

Description: It close the joystick interface.

Return value: 0 if the operation is performed successfully; -1 if the joystick in not installed.

JOY26_INSTALLED

```
int JOY26_installed(void);
```

Description: Return if the joystick driver is actually installed.

Return value: 0 if the joystick is installed; 1 otherwise.

JOY_ENABLE

```
void joy_enable(void);
```

Description: Allow the driver to receive data from the hardware.

JOY_DISABLE

```
void joy_disable(void);
```

Description: Trow away the data arriving from the hardware instead of processing them inside the driver.

JOY_SETSTATUS

```
void joy_setstatus(int axe0, int axe1, int axe2, int axe3, int buttons);
```

Description: Set values for axes and buttons. Usually `axe0` and `axe1` are x and y axis for the first joystick while `axe2` and `axe3` are x and y axis for the second one. In the `buttons` parameter each bit rappresent the status of one button.

JOY_GETSTATUS

```
void joy_getstatus(int *axe0, int *axe1, int *axe2, int *axe3, int *buttons);
```

Description: Get values for axes and buttons. Usually `axe0` and `axe1` are x and y axis for the first joystick while `axe2` and `axe3` are x and y axis for the second one. In the `buttons` parameter each bit rappresent the status of one button.

3.4 The speaker library

The system provide a driver to use the internal PC speaker. It can play a note at a given frequency for a predefined period or in a endless loop. In order to use the speaker handling functions, the `drivers/shark_spk26.h` header file, containing the interface functions' prototypes, has to be included in the application program.

First of all, the mouse needs be initialized by the calling `SPEAK26_init` primitive into the `__init__` function in the initialization file or in any other point of the application code and the input low level driver must be already installed..

SPEAK26_INIT

```
int SPEAK26_init(void);
```

Description: It initializes the speaker interface and the library's internal data structures.

Return value: 0 if the operation is performed successfully; a value less than 0, otherwise.

SPEAK26_CLOSE

```
int SPEAK26_close(void);
```

Description: It close the speaker interface.

Return value: 0 if the operation is performed successfully; -1 if the speaker in not installed.

SPEAK26_INSTALLED

```
int SPEAK26_installed(void);
```

Description: Return if the speaker driver is actually installed.

Return value: 0 if the speaker is installed; 1 otherwise.

SPEAKER_SOUND

```
void speaker_sound(unsigned int hz, unsigned int ticks);
```

Description: Generate a note at a frequency given using the parameter `hz`. The parameter `ticks` is the duration of the note; if set to 0 the note is repeated in an endless loop.

SPEAKER_MUTE

```
int speaker_mute(void);
```

Description: Reset the internal speaker making it silent.

Chapter 4

The Frame Buffer Library

The S.Ha.R.K. system provides support for all modern SVGA cards through the Linux Frame Buffer driver. Using the `grx` graphic library upon it is possible to draw points, lines, rectangles, boxes, circles, and text on 16 bit per plane (bpp) SVGA graphic modes.

In order to use graphics, a program must include the `drivers/shark_fb26.h` header file. Then, it must initialize the Frame Buffer using `FB26_init()`. At this point the drawing library must be connected to the frame buffer with the function `FB26_use_grx()`. Now a graphic mode can be opened using `FB26_setmode()`, and then the drawing functions can be used. The *num number*, needed as a parameter indicate which frame buffer is used. At the end, the program can switch back to text mode through `FB26_close()`.

FB26_INIT

```
int FB26_init(void);
```

Description: It initializes the rame buffer and internal data structures to access the hardware. The function returns -1 on error, 0 otherwise. In order to use the graphic primitives, a program must call this function.

FB26_OPEN

```
int FB26_open(int num);
```

Description: Open the frame buffer number `num`. The function returns -1 on error, 0 otherwise. The frame buffer must be already initialized with `FB26_init`.

FB26_SETMODE

```
int grx_setmode(int num, unsigned chat *mode);
```

Description: It opens the graphic mode identified by the `mode` parameter. The mode number can be obtained using `grx_getmode()`. The parameter `mode` is a string in the format "`widthxheight-bpp`" (ex. "`640x480-16`"). If the mode is supported and can be opened, the function returns 1, otherwise it returns -1.

FB26_CLOSE

```
int FB26_close(int num);
```

Description: It closes the frame buffer `num` returning to text mode.

4.1 The Frame Buffer graphics functions

The GRX library allows to use graphics with 16 bpp; the number of bits per pixel, the graphic depth, determines the number of colors that can be simultaneously displayed on a single screen. In 16 bpp modes, each pixel is represented by two bytes. Since 16 is not divisible by 3, a component (the green one) is described by 6 bits, whereas the other two are described by 5 bits. The `RGB16()` macros help to code RGB values in a pixel value for all these graph functions.

RGB16

```
WORD rgb16(WORD r, WORD g, WORD b);
```

Description: It returns the color value defined by the 3 parameters (`red`, `green` and `blue`) in the format required by drawing function.

GRX_CLEAR

```
void grx_clear(DWORD color);
```

Description: It clears the graphic screen by filling it with the color specified in the parameter `color`.

GRX_PLOT

```
void grx_plot(WORD x, WORD y, DWORD col);
```

Description: It draws a pixel of color `c` at coordinates (`x`,`y`) on the screen. For efficiency reasons no checks are performed on `x` and `y`. Only the `bpp` less significative bits of `col` are used (where `bpp` is the number of bits per plane in the current graphic mode).

GRX_GETPIXEL

```
DWORD grx_getpixel(WORD x, WORD y);
```

Description: It returns the color of pixel at coordinates (`x`,`y`) on the screen. For efficiency reasons no checks are performed on `x` and `y`. Only the `bpp` less significative bits of the returned value are used (where `bpp` is the number of bits per plane in the current graphic mode).

GRX_PUTIMAGE

```
void grx_putimage(WORD x1, WORD y1, WORD x2, WORD y2, BYTE *img);
```

Description: It writes a rectangular bitmap from system memory to video memory. (`x1`, `y1`) is the top left corner, while (`x2`,`y2`) is the right bottom corner. It fills the specified box with the data in the buffer pointed by `*img`. The memory buffer must contain the pixels in the same representation used in the video memory, starting at the top left corner, from left to right, and then, line by line, from up to down, without any gaps and interline spaces.

See also: `grx_getimage()`.

Example

```
BYTE videobuff[200][200];
...
void * videotask(void *arg) {
      int done = 0;
      ...
      while (!done) {
            done = decodeframe(videobuff, 200, 200);
            grx_put(X, Y, X+200, Y+200, videobuff);
            task_endcycle();
      }
}
```

GRX_GETIMAGE

```
void grx_getimage(WORD x1, WORD y1, WORD x2, WORD y2, BYTE *img);
```

Description: It reads a rectangular bitmap from video memory to system memory. (`x1`, `y1`) is the top left corner, while (`x2`,`y2`) is the right bottom corner. It fills the specified buffer pointed by `*img` with the data contained in the selected video box. The memory buffer must be large enough to contain the box (in general, the correct buffer dimension is $(y2 - y1 + 1) * (x2 - x1 + 1) * bpp$).

See also: `grx_putimage()`.

GRX_RECT

```
int grx_rect(WORD x1, WORD y1, WORD x2, WORD y2, DWORD col);
```

Description: It draws an empty rectangle with top left corner at (`x1`,`y1`) and bottom right corner at (`x2`,`y2`). The rectangle is drawn with color `col`.

GRX_BOX

```
int grx_box(WORD x1, WORD y1, WORD x2, WORD y2, DWORD col);
```

Description: It draws a filled rectangle with top left corner at (`x1`,`y1`) and bottom right corner at (`x2`,`y2`). The box is drawn with color `col`.

GRX_LINE

```
void grx_line(WORD x1, WORD y1, WORD x2, WORD y2, DWORD col);
```

Description: It draws a line from (`x1`, `y1`) to (`x2`,`y2`) using color `col`.

GRX_TEXT

```
void grx_text(char *text, WORD x, WORD y, DWORD fg, DWORD bg);
```

Description: It writes a 0 terminated text string in graphic mode at position (`x`,`y`). The string is pointed by `text`, `fg` is the foreground color, and `bg` is the background color.

GRX_CIRCLE

```
void grx_circle(WORD x, WORD y, WORD r, DWORD col);
```

Description: It draws a circle of radius `r` and color `col`, centered at (`x`, `y`).

GRX_DISC

```
void grx_disc(WORD x, WORD y, WORD r, DWORD col);
```

Description: It draws a filled circle of radius `r` and color `col`, centered at (`x`, `y`).

Chapter 5

The Frame Grabber Library

TODO

Chapter 6

The CPU frequency scaling library

This driver allow the application to change the CPU speed in order to reduce power consumption. After the driver initialization is possible to know the list of supported frequencies, the minimin and maximun allowed frequncy. Is possible to get and set the current frequecy and obtaind the deoretical value of the transition duration.

6.1 CPU Information utility

These functions allow the application to know informations about the CPU like manufacturer, model, capabilities, etc.

CPU26_INIT

```
int CPU26_init(void);
```

Description: Initialize the driver and all internal structures. The function returns 0 if the procedure was succesfully, -1 otherwise.

CPU26_CLOSE

```
int CPU26_close(void);
```

Description: It close the CPU driver.

Return value: 0 if the operation is performed successfully; -1 if the keyboard in not installed.

CPU26_INSTALLED

```
int CPU26_installed(void);
```

Description: Return if the event debugger is actually installed.

Return value: 0 if the module is installed; 1 otherwise.

CPU26_SHOWINFO

```
void CPU26_installed(void);
```

Description: Print the CPU informations retrived by the driver.

6.2 CPU scaling functions

These functions allow to get/set parameter about the CPU frequency and its behavior. The low level must be initialized with the `CPU26_init()` function before using scaling funtionalities.

CPU26_DVS_INIT

```
int CPU26_DVS_init(void);
```

Description: Initialize the driver and all internal structures. The function returns the CPU identifyer if that present DVS capabilities, -1 otherwise.

CPU26_DVS_CLOSE

```
int CPU26_DVS_close(void);
```

Description: It close the CPU driver.

Return value: 0 if the operation is performed successfully; -1 if the driver in not installed.

CPU26_DVS_INSTALLED

```
int CPU26_DVS_installed(void);
```

Description: Return if the DVS driver is actually installed.

Return value: The function returns the CPU identifyer if that present DVS capabilities, -1 otherwise.

CPU26_GET_LATENCY

```
int CPU26_get\_latency(void);
```

Description: Return the value of the latency time needed to change between two frequencies.

CPU26_GET_MIN_FREQUENCY, CPU26_GET_MAX_FREQUENCY

```
int CPU26_get_min_frequency(void);

int CPU26_get_max_frequency(void);
```

Description: Are used to obtain minumum and maximum allowed frequencies.

CPU26_GET_CUR_FREQUENCY

```
int CPU26_get_cur_frequency(void);
```

Description: Return the value of the actual frequency of the processor.

CPU26_SET_FREQUENCY

```
int CPU26_set_frequency(int target, unsigned int relation);
```

Description: Return

CPU26_GET_FREQUENCIES

```
int CPU26_set_frequency(int *freqs);
```

Description: Return

CPU26_SHOW_FREQUENCIES

```
int CPU26_show_frequency(char *buff);
```

Description: Return

Chapter 7

The Network Library

To allow communication among different computers, S.Ha.R.K. provides a Network Library implementing the UDP/IP stack on an Ethernet network. The library is organized in three layers:

- low-level driver: this layer is hardware dependent, since it interacts with the network card;
- ethernet layer: this layer allows the upper layer to send and receive ethernet frames. It is not intended to be used by a user program, but only by the code implementing the network protocol;
- high-level layer: this layer implements the network (IP) and the transport (UDP) protocols. It provides the interface used by a user program to access the network through the UDP protocol.

The low-level driver is implemented in order to respect the system real-time requirements (avoiding unpredictable delays in sending/receiving frames). This result is achieved by solving two different problems: the interrupt handling (in the receive phase) and the mutual exclusion needed for accessing the network card (in the transmission phase).

The first problem is solved by using a SOFT task to handle the network card interrupts: on a frame arrival, a task handling the reception is activated. Such a task is guaranteed along with all the other tasks in the system, thus it cannot jeopardize their schedulability. Since a minimum interarrival time for the frames cannot be predicted, the receiving task cannot be a sporadic (HARD) task; therefore the task uses a SOFT_TASK_MODEL, and we have used a Constant Bandwidth Server (CBS) to serve it.

The second problem can be solved using two different methods. The first method adopts a shared memory programming paradigm: a task willing to transmit is allowed to access the network card; mutually exclusive accesses are guaranteed by semaphores. This solution is very simple and the introduced overhead is very low. The second soultion is based on the utilization of a server task devoted to send frames on the network on behalf of other tasks. Each task posts its frames in a mailbox, whence the sender task picks them up. At the moment, only the first solution is implemented, but in order to provide a good degree of flexibility, both approaches will be supported as soon as possible.

The most diffused higher level protocols have been implemented upon the ethernet level. In order to use them within a S.Ha.R.K. application, the `drivers/udpip.h` file, containing the functions prototypes and the data structures, has to be included in the application program.

As a first step, the network drivers have to be initialized. This is done by the `net_init()` primitive that requires the machine's IP address. After initialization, the program has to bind itself to a port by a socket (in UNIX's fashion) by the `udp_bind()` primitive. Afterwards, it is possible eihter to receive packets by using `udp_recvfrom()`, or to send them by using `udp_sendto()`.

NET_INIT

```
void net_init(NET_MODEL *m)
```

Description: It is an interface function used for calling the different layers initializing functions. The `m` parameter specifies the protocols that are going to be activated along with the parameters to be passed to their initializing fucntions. The predefined `net_base` value, if used as `net_init` parameter, causes the ethernet level to be solely intialized by using a mutex semaphore for enforcing mutual exclusion. Moreover, the `net_setudpip(m, addr)` macro is defined to select the UDP/IP protocols stack with local IP address `addr`, expressed in string format. The task used to handle network card interrupts has a SOFT_TASK_MODEL obtained initializing such a model with these arguments:

```
soft_task_default_model(m);
soft_task_def_wcet(m, 1000);
soft_task_def_period(m,20000);
soft_task_def_met(m, 1000);
soft_task_def_aperiodic(m);
soft_task_def_system(m);
soft_task_def_nokill(m);
```

Example

```
int main(int argc, char **argvvoid)
{
    NET_MODEL m = net_base;
    char talk_myipaddr[50];
    ...
    strcpy(talk_myipaddr, "193.205.82.47");
    net_setudpip(m, talk_myipaddr);
    net_init(&m);
    ...
```

IP_STR2ADDR

```
int ip_srt2addr(char *str, IP_ADDR *ip)
```

Description: It converts the IP address, contained in the `str` string parameter, into `IP_ADDR` format. The result is returned in the variable pointed by `ip`. The function returns `TRUE` if the operation has been succesful, `FALSE` otherwise.

UDP_BIND

```
int udp_bind(UDP_ADDR *a, IP_ADDR *bindlist);
```

Description: It binds the receiving program on the specified UDP port. A socket is created and its identifier is returned. Moreover the host addresses specified through the `bindlist` parameter are loaded into the ARP table. The port is identified by the `a` parameter which is composed of the fields named `s_addr`, having `IP_ADDR` type, and `s_port`, having `WORD` type. The `s_port` parameter is the most meaningful since it specifies the port the function binds to. Further details can be found in any UNIX manual. The possibility of specifying the hosts that will be accessed (through the `bindlist` parameter), permits to add ARP table entries in the network initialization phase. In this way, the timing impredictability introduced by ARP can be reduced. This possibility can be discarded by chosing a `NULL` value for the `bindlist` parameter. Otherwise such a parameter has to be a pointer to a `NULL` terminated `IP_ADDR` array. As we said earlier, the returned socket identifier can be fed into the `udp_sendto()` primitive.

Example

```
void * txsessiontask(void *arg) {
      UDP_ADDR local;
      IP_ADDR bl[5];
      int sock;
      /* The periodic txsessiontask, upon its creation,*/
      /* creates a socket befor entering               */
      /* the infinite cycle typical of all             */
      /* periodic tasks                                */
      local.s_port = 1030; /* local port */
      /* It loads the eth address of the hosts         */
      /* the task will communicate with                */
      /* into the ARP table                            */
      ip_str2addr("193.205.82.47", bl);
      /* terminates bind list by NULL                  */
      *((DWORD *)&bl[1]) = NULL;
      sock = udp_bind(&local, bl);
      ...
}
```

UDP_SENDTO

```
int udp_sendto(int s, char *buf, int nbytes, UDP_ADDR *to);
```

Description: It sends an UDP packet, with size `nbytes`, whose body is pointed by `buf` to an address specified by `to`. The last parameter has `UDP_ADDR` type and identifier either the destination IP address or the port (see `udp_bind` for further information on `UDP_ADDR`). In order to send a packet a local socket has to be created, by calling the `udp_bind()` primitive; its identifier shall be passed by the `s` parameter.

Example

```
void * txsessiontask(void *arg) {
      int sock;
      UDP_ADDR local, to;
      ...
      /* socket creation */
      ...
      for (;;) {
               ...
               /* prepares the destination address */
               to.s_port = 1030;
               ip_str2Addr("127.0.0.1",&(to.s_addr));
               udp_sendto(sock, msg, msglen, &to);
               ...
               task_endcycle();
      }
}
```

UDP_RECVFROM

```
int udp_recvfrom(int s, char *buf, UDP_ADDR *from);
```

Description: It receives a UDP packet from the socket identified by `s`; the packet body is copied into the buffer pointed by `buf`; the sender address (composed of the pair port-IPaddress) is copied into the variable pointed by `from`. The primitive returns the number of bytes composing the packet.

Example

```
TASK rxsessiontask() {
    int sock;
    UDP_ADDR local, from;
    /* The non real-time rxsessiontask           */
    /* creates a receiving socket and            */
    /* enters an infinite loop waiting */
    /* for the incoming packets        */
    ...
    /* During initialization the       */
    /* socket is created               */
    ...
    for (;;) {
             ...
             udp_recvfrom(sock, inmsg, &from);
             /* from contains the sender address */
             ...
             task_endcycle();
    }
}
```

UDP_NOTIFY

```
int udp_notify(int s, int(*f)(int len, BYTE *buf, void *p))
```

Desription: the notifying function `f` is associated with the `s` socket. When a packet addressed to the port the socket is bound to arrives, such a function is invoked. Upon its invocation, the function receives as arguments a pointer to the received packet (`buf`), the packet size (`len`), and a pointer specified along with the `udp_notify()` call (`p`). The notifying function is executed within the context of the receiving task; therefore, it should not consume too much time.

Example:

```
int  hrtp_recvfun(int len, BYTE *b, void *p)
{
    struct HRTP_SESSION *s;
    struct HRTP_HDR *h;
    ...
    /* Notifying function used for receiving */
    /* packets from a session level protocol */
    ...
    /* p is a pointer to the descriptor of   */
    /* the session involved in the transmission */
    s = p;
    /* the received packet is copied into a private */
    /* buffer belonging to the session       */
    h = (struct HRTP_HDR *)&(s->b[s->recvd * instance_dim]);
    memcpy(h, b, len);
    ...
    return 0;
}
...
void  hrtp_recv(struct HRTP_SESSION *s, void (*f)(void))
{
     udp_notify(s->sock, hrtp_recvfun, (void *)s);
     s->notifyparm = f;
     s->active = HRTP_RCVIN;
}
...
```

A programmer that wants to implement a new transport/network level stack different from UDP/IP needs to directly access the Ethernet services. This can be done using the Ethernet layer, accessible through the "eth.h" header file.

In order to receive Ethernet frames, a callback function has to be associated to a frame type (the frame type is a field of a frame): each time that a frame of the specified type will arrive, the callback function will be called. A callback can be bound to a frame type using the `eth_setHeader` library call.

In order to transmit Ethernet frames, a transmission buffer must be allocated and filled: the header can be filled using `eth_setHeader()`, while the body must be explicitly filled after obtaining a pointer to it through `eth_getFDB()`. At this point the frame can be sent using `eth_sendPKT()`.

Transmission buffers can be allocated using the `netbuff` module: this module (usable including the "netbuff.h" header file) permits to manage pools of pre-allocated buffers, in order to minimize the unpredictability due to dynamic allocation.

NETBUFF_INIT

```
void netbuff_init(NETBUFF *netb, BYTE nbuffs, WORD buffdim);
```

Description: It initializes a pool composed of `nbuffs` buffers, each of them have `buffdim` size. The pool is identified by the `netb` descriptor.

NETBUFF_GET

```
void *netbuff_get(NETBUFF *netb, BYTE flag);
```

Description: It returns a pointer to the first free buffer in the pool identified by `netb`. The `flag` parameter, which can assume values `BLOCK` or `NON_BLOCK`, indicates whether the operation is a blocking or non-blocking allocation. If a non-blocking `netbuff_get()` is performed when the pool does not contain any free buffer, a NULL pointer is returned.

NETBUFF_RELEASE

```
void netbuff_release(NETBUFF *netb, void *buff);
```

Description: It marks the buffer pointed by `buff` as free in the `netb` pool.

ETH_INIT

```
void eth_init(int mode, TASK_MODEL *m);
```

Description: It initializes the Ethernet layer, in order to transmit or receive ethernet frames. If the Ethernet layer has already been initialized, it does nothing.

Each network level protocol that needs to access the network card must pass through the Ethernet layer, which must be initialized before receiving or sending anything. The Ethernet layer will search for a supported network card, enable it and initalize some internal structures.

At the moment, the `mode` parameter is not used, in the future it will be used to select an operating mode (mutual exclusion on send operation through a dedicated server task or through mutexes).

This library function is called by `net_init()` in the standard UDP/IP configuration.

The task model used for the task that handles the network card interrupts can be passed with the m parameter. If it is NULL, a SOFT_TASK_MODEL obtained initializing such a model with these arguments will be used:

```
soft_task_default_model(m);
soft_task_def_wcet(m, 1000);
soft_task_def_period(m,20000);
soft_task_def_met(m, 1000);
soft_task_def_aperiodic(m);
soft_task_def_system(m);
soft_task_def_nokill(m);
```

HTONS & NTOHS

```
WORD htons(WORD host);

WORD ntohs(WORD net);
```

Description: These two utility functions convert a WORD from the host format to the net format (`htons`) and vice-versa (`ntohs`).

ETH_GETADDRESS

```
void eth_getAddress(ETH_ADDR *eth);
```

Description: It returns the net card Ethernet address in the `ETH_ADDR` structure pointed by `eth`.

ETH_STR2ADDR

```
void eth_str2addr(char *add, struct ETH_ADDR *ds);
```

Description: It converts an Ethernet address from the string format ("xx:xx:xx:xx:xx:xx") to the byte (`ETH_ADDR`) format. The `add` string contains the address in text format, while `ds` is a pointer to the `ETH_ADDR` structure where the output is placed.

ETH_SETPROTOCOL

```
int eth_setProtocol(WORD type, void (*recv)(void *frame))
```

Description: It is used to specify the callback function `recv` to be called when a frame of type `type` is received. It returns `TRUE` in the case of success, `FALSE` otherwise. When the Ethernet layer will call the `recv` callback, it will pass to the function a pointer to the received frame.

The callback function runs in the context of the driver task, served by a CBS.

Each high-level protocol must use this library call to register itself in order to process incoming packets.

Example

```
void  ip_server_recv(void *pkt)
{
     IP_HEADER *iphd;
     ...

     /* This callback is invoked when an IP packet is received */
     iphd = (IP_HEADER *)eth_getFDB(pkt);
     ...
}
...
void ip_init(char *localAddr)
{
     int i;

     /* Initializes IP */
     ...
     /* Registers the protocol to the ethernet layer */
     eth_setProtocol(ETH_IP_TYPE,ip_server_recv);
     ...
}
```

ETH_SETHEADER

```
void *eth_setHeader(void *b, ETH_ADDR dest, WORD type);
```

Description: It fills the header of the frame pointed by `b` with the destination address `dest` and the frame type `type`. It also returns a pointer to the body of the ethernet frame.

ETH_GETFDB

```
void *eth_getFDB(void *p)
```

Description: Get First Data Byte. It returns a pointer to the body of the frame pointed by the `p` parameter.

ETH_SENDPKT

```
int eth_sendPkt(void *p, int len);
```

Description: It transmits the Ethernet frame pointed by `p`, having lenght `len`. The destination and the frame type must be previously specified using `eth_setHeader`.

Example

```
void  arp_sendRequest(int i)
{
     ARP_PKT *pkt;
     ETH_ADDR broadcast,nulladdr;

     eth_str2Addr("FF:FF:FF:FF:FF:FF",&broadcast);
     eth_str2Addr("00:00:00:00:00:00",&nulladdr);
     eth_setHeader(arpBuff,broadcast,ETH_ARP_TYPE);
     pkt = (ARP_PKT *)eth_getFDB(arpBuff);
     pkt->htype = htons(ARP_ETH_TYPE);
     pkt->ptype = htons(ARP_IP_TYPE);
     pkt->hlen = sizeof(ETH_ADDR);
     pkt->plen = sizeof(IP_ADDR);
     pkt->operation = htons(ARP_REQUEST);
     setEthAddr(pkt->sha,myEthAddr);
     setEthAddr(pkt->tha,nulladdr);
     setIpAddr(pkt->sip,myIpAddr);
     setIpAddr(pkt->tip,arpTable[i].ip);
     eth_sendPkt(arpBuff,sizeof(ARP_PKT));
}
```

ETH_CLOSE

```
int eth_close(void);
```

Description: It closes the Ethernet protocol. If it is not explicitly called by the user, it is automatically executed through `sys_atexit`.

Chapter 8

The CMOS real-time clock

In all PCs there is a real-time clock (with a resolution of 1 second) that can be read or written. This clock usually has a drift of about some seconds in 24 hours. The following functions (that can be found into `rtc.h`) can be used to set/get values; all values are passed using a `struct rtc_time` that contains variables for seconds, minutes, hours, days, months and years.

GET_RTC_TIME

```
int get_rtc_time(struct rtc_time *time);
```

Description: The actual time is read from the CMOS real-time clock and written into the structure pointed by `time`. The function returns zero on success, other values on error.

SET_RTC_TIME

```
int set_rtc_time(struct rtc_time *time);
```

Description: The values contained in the structure pointed by `time` are converted in seconds and written into the CMOS real-time clock; the function returns zero on success, other values on error.

RTC_TIME

Description: it is a data structure containing the following fields:
tm_sec : seconds, from 0 to 59;
tm_min: minutes, from 0 to 59;
tm_hour: hours, from 0 to 23;
tm_mday : day, from 1 to 31;
tm_mon : month, from 1 to 12;
tm_year : year, is an integer number (no Y2K problem);
tm_wday : day of the week, from 1 to 7;
tm_yday : day of the year, from 1 to 365;
tm_isdst : 0 if legal hour, 1 if solar hour (not used yet);

Chapter 9

The Sound Library

If a SoundBlaster16 sound card is available, S.Ha.R.K. allows to sample and play sounds by using the functions provided by the sound library[1]. The library currently supports either program or DMA controlled sampling an playing, according to 4 possible operating modes:

- PIO mode;
- DMA-Raw mode;
- DMA-Double-buffering mode;
- DMA-Self-buffering mode.

Working under PIO mode, sounds can be sampled and played only with 8 bit PCM. The frequence depends on the hardware speeds but cannot in any case overcome 10 Khz. This mode is reserved for the pure classical hard real-time approach which refuses the usage of DMA controlled I/O.

The DMA-Raw mode uses DMA controller to sample and play directly on a memory buffer. Owing to technical problems related to the structure of the PC DMA controller, the buffer's size can be no bigger than 64K. This mode is the one that minimizes the DMA operations' impact on CPU.

The DMA Double-Buffering mode uses an internal buffer in order to overcome the 64k limitation. The internal buffer is split into two parts: while the DMA tranfers data to one half, an ad-hoc task moves data between the second half and a user-provided external memory region. In this way, it is possible for a user to work on samples much bigger than 64K, paying the fee of a higher CPU load[2].

The DMA-Self-Buffering mode allows the user to directly handle the internal buffer. The user specifies a function to be activated every time the DMA controller has finished transferring data on one half of the internal buyffer. In this mode, the user can obtain the data while they are being sampled; the time lag between sampling and data delivery is thus reduced. Such a feature makes this working mode interesting for real-time applications.

Independently of the chosen working mode, an operation can be either synchronous or asynchronous. A synchronous operation provides the task invoking the operation with a synchronizing point located at its ending.

In order to use the sound library functions, the files `drivers/sound.h` and `drivers/dma.h` must be included. The former contains the prototypes of the declared functions, the latter is necessary because the sound library uses DMA.

The first step to be performed is initializing the audio drivers by the `sound_init` function. Then, if one wishes to work in DMA-Raw mode, it is necessary to allocate a memory buffer and align it by

[1] Currently only the sound blaster 16 is supported; the code of the library is directly inherited from the Hartik 3.3.0 Kernel...

[2] This is possible only if the protected mode is used.

calling `dma_getpage()` (in the remaining modes no particular alignment is required for the buffer). If the DMA-Self-buffering mode is chosen, the programmer has to properly set the functions to be called every time the DMA finishes working on one half of the internal buffer; this can be done by calling the `sound_setfun()` primitive. As soon as these operations have been performed, sampling or playing can be made through `sound_sample()` and `sound_play()`, respectively.

SOUND_INIT

```
void sound_init(WORD rawbufsize, WORD tick);
```

Description: It initializes the audio driver by allocating the internal buffer for the DMA-Double-buffering and DMA-Self-Buffering modes. The `rawbufsize` parameter contains the dimension of this buffer. Higher values reduce the CPU load and are thus advised when using the DMA-Double-buffering mode. Lower values, on the contrary, can be used to shorten the latency between sampling and data delivering (particularly when using DMA-Self-buffering). The `tick` parameter contains the value of the system tick; its correctness is fundamental for the PIO mode.

SOUND_INFO

```
void sound_info(void);
```

Description: It outputs on the screen some information concerning the soundcard and the drivers.

SOUND_SETFUN

```
void sound_setfun(int (*infun)(BYTE *rawbuff),
     int (*outfun)(BYTE *rawbuff));
```

Description: It specifies the functions to be called when the DMA finishes working on one of the two internal buffer's halves when using DMA-Self-Buffering mode. The function pointed by `infun` is used when performing sampling operations, whereas `outfun` is used for playing operations. Both functions receive a pointer to the half-buffer not currently acted upon by the DMA (the half-buffer sizes are equal to one half of the `sound_init()` parameter) and have to return 0 if the operation has not yet been finished, 1 if it is going to finish in the next DMA cycle, and 2 if it finishes immediately. Attention should be paid to the fact that these functions are periodically called with a frequency equal to the operation's frequency divided by the half-buffer's size; thus, they should be very short in order not to overload the system.

Example:

```
int osc_fun(BYTE *b)
{
   int i;
   int sum = 0;
   BYTE *p;
   /* Averages the values read from the buffer */
   /* and writes the result on a CAB shared    */
   /* with a task                              */
   for (i = 0; i < (BUFFDIM » 1); i++)
       sum += b[i];
   sum /= (BUFFDIM » 1);
   p = cab_reserve(cc);
   *p = (BYTE) sum;
   cab_putmes(cc, p);
   return 0;
}
...
void * io_task(void *arg)
{
      int x, y;
      BYTE *p;
      BYTE page = 0;
      char str[50];
      short int talk, silencecount;
      /* This task reads the value put on the CAB by */
      /* the self-buffering function                 */
      /* sets the self-buffering function            */
      sound_setfun(osc_infun, -1);
      /* starts the sampling operation               */
      sound_sample(NULL, 20000, 0, DMA_OP | PCM8 | MYFUN);

      cc = cab_create("osc_cab", sizeof(BYTE), 3);
      for (;;) {
          /* reads and proccesses */
          /* the CAB's value      */
          ...
          task_endcycle();
      }
      return 0; }
```

SOUND_SAMPLE

```
void sound_sample(BYTE *buf, DWORD sps, DWORD len, BYTE t);
```

Description: It samples `len` bytes in the `buf` buffer at the frequency of `sps` samples per second with the mode expressed by `t`. The latter can be assigned one of the following constants:

- `PIO_OP` operates using PIO mode: as said earlier, in this mode values for `sps` higher than 10000 make no sense. Moreover, for the sampling and playing to happen with the correct timing, it is necessary that the audio driver be initialized with the `tick` parameter set to the system tick expressed in microseconds (see `sound_init()` for more details).

- `DMA_OP` operates using one of the DMA modes (the default is DMA-Double-Buffering). The internal buffer size is specified in `sound_init`.
- `PCM8` operates using 8 bit PCM format (it is the default). It is the only possible format in PIO mode.
- `PCM16` operates using 16 bit PCM format. This choice is meaningless in PIO mode.
- `SYNCH` synchronous operation: it is necessary to call `sound_wait()` after `sound_sample()`.
- `ASYNCH` asynchronous operation.
- `MYFUN` operates with DMA-Self-buffering mode; it makes sense only if `DMA_OP` has been set.
- `NOBUFF` operates in DMA-Raw-Mode; it makes sense only if `DMA_OP` has been set.

Example:

```
BYTE buff[0xFFFFF]; /* buffer for sampling */

void main()
{
    sys_init(&s);
    keyb_init(NULL);
    clear();

    sound_init(0x4000, TICK);
    sound_info();

    cprintf("Recording...");
    sound_sample(buff, 44000, 0x8FFFF, DMA_OP | PCM8 | SYNCH);
    ...
}
```

SOUND_PLAY

```
void sound_play(BYTE *buff, DWORD sps, DWORD len, BYTE t);
```

Description: It plays `len` bytes taken from the `b` buffer at the frequency of `sps` samples per second with the mode expressed by `t`. As far as the values of `t` are concerned, the reader can refer to `sound_sample`.

DMA_GETPAGE

```
BYTE *dma_getpage(DWORD *dim);
```

Description: It allocates a buffer having size `dim` fitting for use in DMA operations. Such a usage is possible only if the buffer does not contain bytes whose address differs in the Most Significant Bits. The best way to achieve this feature is to allocate buffers sized less than 64K starting from addresses having the LSB equal to 0. This job is performed by `dma_getpage`. It should be noted that such a feature is necessary only using DMA-Raw-Mode, since the buffer allocation is automatically performed by `sound_init` when using DMA-Double-Buffering and DMA-Self-Bufering modes.

Example

```
void main(void)
{
     BYTE *p;
     int i;

     /* Monitors the time stolen by the DMA */
     /* to the CPU during a 10 Khz sampling */
     sys_init(&s);
     keyb_init(NULL);
     ...
     clear();
     p = dma_getpage(0xFFFF);
     sound_init(0x200, TICK);
     sound_info();

     for (i = 0; i < 80; i++)
         cprintf("_");
     cprintf("ref_time:%f ", myrif);
     cprintf("Unloaded system:  %f", load(&myrif));

     cprintf("DMA Recording...");
     sound_sample(p, 10000, 0xFFFF, DMA_OP | PCM8 | NOBUFF);
     ...
```

SOUND_WAIT

```
void sound_wait(void);
```

Description: It is the synchronization primitive for synchronous operations. The task calling `sound_wait()` blocks itself until the synchronous operation is finished. The call to this function is mandatory for synchronous operations. On the other hand, using the function in conjunction with an asynchronous operation is an error.

Example

```
...
sound_sample(buff, 44000, 0x8FFFF, DMA_OP | PCM8 | SYNCH);
...
/* waits until the sampling termination */
sound_wait();
...
```

Chapter 10

The Console Library

The output on the screen in text mode is supported by a group of functions that act on the whole screen and modify, each time, the cursor's position. Since such functions are not reentrant, they must be used in mutual exclusion.

In order to use the library for displaying on the screen, the "`cons.h`" file must be included. This file contains the values for the usable colors whose symbolic names are listed below: `BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, BROWN, GRAY, LIGHTGRAY, LIGHTBLUE, LIGHTGREEN, LIGHTCYAN, LIGHTRED, LIGHTMAGENTA, YELLOW, WHITE`.

Two global read-only variables `cons_columns` and `cons_rows` contain the number of screen columns and the number of screen rows, respectively. They can be used to write applications that are indipendent from the screen dimensions.

CPUTC, CPUTS, CPRINTF

```
void cputc(char c);

void cputs(char *s);

int cprintf(char *fmt,...  );
```

Description: `cputc`, `cupts` and `cprintf` are used to print on the screen a character, a string, or a formatted string, respectively. In the latter case, the standard C I/O formatting conventions are used.

Warning: since these functions modify the cursor position, they are not reentrant, and must be used in a mutually exclusive fashion.

PUTC_XY, PUTS_XY, PRINTF_XY, GETC_XY

```
void putc_xy(int x,int y,char attr,char c)

void puts_xy(int x,int y,char attr,char *s)

int printf_xy(int x,int y,char attr,char *fmt,...)

char getc_xy(int x,int y,char *attr,char *c)
```

Description: These functions are similar to the previously defined ones; the only difference is that since they're reentrant can be used concurrently by multiple tasks (the others cannot be used for this purpose because they modify the cursor's state). `getc_xy` is used to read the charcater and its attribute at a specific position on the screen.

CLEAR, _CLEAR, SCROLL, _SCROLL

```
void _clear(char c,char attr,int x1,int y1,int x2,int y2)

void clear(void)

void _scroll(char attr,int x1,int y1,int x2,int y2)

void scroll(void)
```

Description: the `clear` and `scroll` functions are used for clearing the screen and for scrolling it upwards. They are based on the `_clear()` and `_scroll()` functions used for clearing or scrolling a window (defined by `x1, y1, x2, y2`) by filling the area with `attr` color and, in the case of `_clear`, with character `c` as well.

SET_ACTIVE_PAGE, SET_VISUAL_PAGE, GET_ACTIVE_PAGE, GET_VISUAL_PAGE

```
void set_active_page(int page)

void set_visual_page(int page)

int get_active_page(void)

int get_visual_page(void)
```

Description: The video cards working in text mode use the CGA hardware scheme. In this scheme, every screen occupies 4000 bytes (80 columns x 25 rows; each position is associated with two bytes, one for codifying the character and another for codifying the color). The video card memory is in general bigger, therefore multiple screen pages can be used simultaneously. The currently visualized screen is called the `visual page`, whereas the screen the output is directed to is called the `active page`. The cited functions are essentially used for handling, at a given instant, the visual page or the active page. SUGGESTION: if the active page or the visual page are modified, they should be restored to the original (0) value on exit; the same is true for the cursor.

PLACE, CURSOR, CURSOR_INFO

```
void place(int x, int y)

void cursor(int start_scanline, int end_scanline)

void cursor_info(int *x, int *y)
```

Description: These functions are used for handling the cursor. More specifically, `place` sets the cursor position (`x` belongs to the range 0...79, `y` to 0...24).

CURSOR_BLOB, CURSOR_STD, CURSOR_OFF

```
void cursor_blob(void)

void cursor_std(void)

void cursor_off(void)
```

Description: These macros call the `cursor()` function to set the cursor shape to be a big rectangle, the standard underscore, or invisible.

Chapter 11

The File Management

S.Ha.R.K. provides a built-in File System that currently supports Hard Disk Drivers and FAT16 partitions.

If you are using the DOS eXtender X to run the application, you can use some callback to the DOS 0x21 interrupt, to write/read some bytes from the filesystem[1]. These functions directly interact with the underlying DOS, and can not be used when the system is in protected mode. In particular, you can only use these functions into the __kernel_register_levels__() function and in the RUNLEVEL_AFTER_EXIT exit functions.

In the case of errors, a `NULL` or zero value is returned; `DOS_ferror()` can be used to get the DOS error code; the header file `<ll/i386/x-dos.h>` must be included to use these functions. A running (we hope) well documented example can be foun in the `demos/dosfs` directory.

DOS_FOPEN

```
DOS_FILE *DOS_fopen(char *name, char *mode);
```

Description: It opens a file and returns a pointer to a file structure. The `name` parameter contains the name of the file to be opened. The `mode` parameter contains a string whose value can be one of the following constants: "r" to read, "w" to write, and "rw" to read and write. In the case of error, `NULL` is returned; otherwise the returned value can be used as last parameter in the functions listed below.

DOS_FCLOSE

```
void DOS_fclose(DOS_FILE *f);
```

Description: It closes the specified file and releases all allocated DOS resources.

DOS_FREAD

```
DWORD DOS_fread(void *buf, DWORD size, DWORD num, DOS_FILE *f);
```

Description: It reads `num` objects of `size` bytes from file `f` and place them in a buffer pointed by `buf`. This function returns the actual number of bytes read from the file (it can be less than $num * size$ bytes). Zero is returned if an error occurs or *end-of-file* is found.

[1] These callbacks are useful when you don't have any partition that can be read by the filesystem... for example when running S.Ha.R.K. applications from a FAT32 filesystem!

DOS_FWRITE

```
DWORD DOS_fwrite(void *buf,DWORD size,DWORD num,DOS_FILE *f);
```

Description: It writes `num` objects of `size` bytes into file `f`. Data are picked from the buffer pointed by `buf`. This function returns the actual number of bytes written (it can be less than $num*size$ bytes). Zero is returned if an error occurs.

DOS_ERROR

```
unsigned DOS_error(void);
```

Description: Returns the error code of the latest `DOS_xxx` function.

Chapter 12

The Snapshot Library

The library allow applications to save up to 16 screen snapshot to file. Before the grabbing is necessary to allocate the memory for a snapshot slot and files can be created only at the end of execution when the system return in real mode.

SNAPSHOT_GETSLOT

```
void *snapshot_getslot(int nbuff, int wx, int wy, int byesperpixel);
```

Description: This function allocate the memory for a snapshot of given dimensions. The `nbuff` parameter indicate which slot must be allocated. `wx` and `wy` are width and height of the screen resolution while `bytesperpixel` is the depth of the screen.

SNAPSHOT_FREESLOT

```
void snapshot_freeslot(int nbuff)
```

Description: This function free the memory reserved for the slot indicated by the `nbuff` parameter.

SNAPSHOT_GRAB

```
void snapshot_grab(int nbuff)
```

Description: This function get a snap of the video memory and copy it inside the slot specified by the `nbuff` parameter.

SNAPSHOT_SAVEPGM

```
int snapshot_savepgm(int nbuff, char *fname)
```

Description: This function save the screen snapshot saved inside the nbuff slot to a PGM greyscale file. The filename passed to the function using the fname parameter.

SNAPSHOT_SAVEPPM

```
int snapshot_saveppm(int nbuff, char *fname)
```

Description: This function save the screen snapshot saved inside the nbuff slot to a PPM color file. The filename passed to the function using the fname parameter.

Index