S.Ha.R.K. User Manual

Volume I

Kernel Primitives

Written by

Giorgio Buttazzo (giorgio@sssup.it)

Paolo Gai (pj@sssup.it)

Luigi Palopoli (luigi@hartik.sssup.it)

Marco Caccamo (caccamo@sssup.it)

Giuseppe Lipari (lipari@sssup.it)

Scuola Superiore di Studi e Perfezionamento S. Anna

RETIS Lab

Via Carducci, 40 - 56100 Pisa

Contents

1 Introduction **5**
- 1.1 General Description . . . 6
- 1.2 SHARK.CFG . . . 7
- 1.3 Predefined Types and Constants . . . 7

2 System Start-up and Termination **10**
- 2.1 Initialization File . . . 10
- 2.2 System primitives . . . 13

3 Task Management **15**
- 3.1 Task Model . . . 15
- 3.2 The scheduling policy . . . 17
- 3.3 Task Creation . . . 17
- 3.4 Group Creation . . . 18
- 3.5 Task Activation and Termination . . . 19
- 3.6 Task Instances . . . 21
- 3.7 Task (thread) specific data . . . 21
- 3.8 Task cancellation . . . 23
- 3.9 Join . . . 24
- 3.10 Preemption control . . . 25
- 3.11 Suspending a task . . . 25
- 3.12 Job ExecutionTime (JET) estimation . . . 25

4 Synchronization and communication **28**
- 4.1 POSIX Semaphores . . . 28
- 4.2 Internal Semaphores . . . 33
- 4.3 Mutexes and Condition Variables . . . 34
  - 4.3.1 Mutex attributes . . . 34
  - 4.3.2 Functions . . . 34
- 4.4 Communication Ports . . . 37
- 4.5 Cyclical Asynchronous Buffers . . . 41
- 4.6 POSIX Message Queues . . . 44

5 Utility functions **45**
- 5.1 Reading time . . . 45
- 5.2 Getting information on tasks . . . 45
- 5.3 Printing messages on the console . . . 46

6 Signals and Exception Handling **47**
6.1 Signals 47
6.2 Exception handling 47

7 Interrupt and HW Ports handling **49**
7.1 Setting an interrupt handler 50
7.2 Reading and writing from I/O ports 50
7.3 Disabling/Enabling interrupts 51
7.4 Saving/Restoring interrupts 51
7.5 Masking/Unmasking PIC interrupts 51

8 Memory Management Functions **52**

A Errors and Exceptions **54**
A.1 Abort codes 54
A.2 Exceptions posted with kern_raise 54
A.3 POSIX error codes 55
A.4 S.Ha.R.K. error codes 55

Chapter 1

Introduction

Real-time computing is required in many application domains, ranging from embedded process control to multimedia systems. Each application has peculiar characteristics in terms of timing constraints and computational requirements (such as periodicity, criticality of the deadlines, tolerance to jitter, and so on). For this reason, a lot of different scheduling algorithms and resource allocation protocols have been proposed to conform to such different application demands, from the classical fixed or dynamic priority allocation schemes to adaptive or feedback-based systems.

However, most of the new approaches have been only theoretically analyzed, and sometimes evaluated using a scheduling simulator. In this case, the algorithm performance is not evaluated on real examples, but only on a synthetic workload. This choice is often dictated from the fact that writing a kernel from scratch every time a new scheduling algorithm is proposed would be unrealistic and would not offer the availability of meaningful applications. A more effective approach is to modify an existing kernel (such as Linux), since most of the existing applications and device drivers written for the host OS can be used in a straightforward fashion. On the other hand, a general purpose kernel is designed aiming at specific goals and generally its architecture is not modular enough for replacing or modifying the scheduling policy. Moreover, classical OSs do not allow to easily define a scheduling policy for resources other than the CPU and this poses a further limitation for testing novel research solutions. This is mainly due to the fact that the classical OS structure does not permit a precise *device scheduling* (due to problems involving resource contention, priority inversion, interrupt accounting, long non-preemptive sections, and so on). A small kernel providing short non-preemptable sections, aperiodic real-time threads for handling interrupts, and a distinction between *device drivers* accessing the hardware and *device managers* implementing the *device scheduling* algorithms would help the progress in this research field. The problems explained above emerge both in the educational and research environments, when the focus is oriented in developing and testing new scheduling algorithms rather than hacking the code of a complex system.

S.Ha.R.K. (Soft and Hard Real-time Kernel), is a research kernel purposely designed to help the implementation and testing of new scheduling algorithms, both for the CPU and for other resources. The kernel can be used to perform early validation of the scheduling algorithms produced in the research labs, and to show the application of real-time scheduling in real-time systems courses. These goals are fulfilled by making a trade off between simplicity and flexibility of the programming interface on one hand and efficiency on the other. This approach allows a developer to focus his/her attention on the real algorithmic issues, thus saving significant time in the implementation of new solutions. Another important design guideline is the use of standard naming conventions for the support libraries in order to ease the porting of meaningful

applications written for other platforms. The results have been satisfactory for applications such as an MPEG player, a set of network drivers and a FFT library.

The kernel provides the basic mechanisms for queue management and dispatching and uses one or more external configurable modules to perform scheduling decisions. These external modules can implement periodic scheduling algorithms, soft task management through real-time servers, semaphore protocols, and resource management policies. The modules implementing the most common algorithms (such as RM, EDF, Round Robin, and so on) are already provided, and it is easy to develop new modules. Each new module can be created as a set of functions that *abstract* from the implementation of the other scheduling modules and from the resource handling functions. Also the applications can be developed independently from a particular system configuration, so that new modules can be added or replaced to evaluate the effects of specific scheduling policies in terms of predictability, overhead, and performance. Low-level drivers for the most typical hardware resources (like network cards, graphic cards, and hard disks) are also provided, without imposing any form of device scheduling. In this way, device scheduling can be implemented by the user to test new solutions. To avoid the implementation of a new non-standard programming interface, which would discourage people from using the kernel, S.Ha.R.K. implements the standard POSIX 1003.13 PSE52 interface [**?**, **?**].

This manual was derived from the Hartik User Manual release 3.3.1.

1.1 General Description

S.Ha.R.K. has been designed as a library of functions which extends the classical C library, by providing a multiprogramming environment with an explicit management of time. From a logical point of view, the system is based on a *Host* computer where the application is developed and on a *Target* computer where the application executes. Development tools are located on the host system, where a general purpose operating system is used. After its compilation, the application is loaded on the target system using the appropriate *loader*. This separation, typical of many hard real-time development systems, enables the final application to run on a variety of target systems, ranging from typical PC to embedded micro-controllers. From a practical point of view, host and target may be the same computer and in the rest of this manual we will not further distinguish between them.

S.Ha.R.K. has been developed focusing on modularity of the kernel source code. S.Ha.R.K. is fundamentally a set of routines that runs on top of a library for OS development called OSLib (see http://oslib.sourceforge.net) that has these requirements:

Operating System (OS) You can compile OSLib/S.Ha.R.K. programs using some different host OS in theory, any OS supporting gcc can be used; in practice, we successfully compiled OSLib/S.Ha.R.K. from Linux, DOS and Cygwin.

Compiler The used compiler is gcc. You can use the gcc version that you prefer (we tested gcc 3.3.3 and older version), the important thing is that the linker must produce ELF binaries (in order to be MultiBoot compliant and to avoid problems with the Linux source code inside S.Ha.R.K.). An ELF cross-compile version of gcc is included inside the DJGPP distribution on the S.Ha.R.K. website, so you can easily compile OSLib/S.Ha.R.K. programs inside a standard DOS environment. To compile under Cygwin, it is required to build an ELF cross-compile gcc/linker couple.

Other utilities GNU Make , uname, pwd, cp, rm, X (these utilities can be found in the utility package on the S.Ha.R.K. web site).

Target Requirements The target have to be at least a PC based on Intel 80486 (or compatible) - SMP is not supported - with at least 4Mb of RAM. In order to load OSLib/S.Ha.R.K. programs (MultiBoot compliant), the target must have GRUB installed, or it must run a real mode operating system (such as MSDOS or FreeDOS). If you intend to boot OSLib/S.Ha.R.K. programs from DOS, you also have to download our DOS eXtender X.

Compilation and application linking can be done using the *make* utility, available in any of the development environments mentioned above. In this case, a "makefile" containing the names of all of the .C files composing the application and the directives to link the needed libraries have to be written. For more information, you can look at the installation txt file from the website download page.

1.2 SHARK.CFG

Inside `SHARK.CFG` you can find the main parameters for S.Ha.R.K. configuration. All the stettings inside this file will be crucial to run correctly S.Ha.R.K. and to get the maximum performaces on a x86 machines. The most important options related to the Real-Time behaviour are:

TSC = TRUE/FLASE

- This option enables the Time Step Counter inside the CPU (Pentium or higher). Kern_gettime function will use the TSC register which is faster and more precise than the external PIT. The default value is TRUE. If the system cannot find the TSC, this feature will be disabled and PIT will be used to get the system time.

APIC = TRUE/FALSE

- This option enables the APIC (Pentium Pro or higher). As TSC, APIC is faster and more precise than the standard PIT. It will be used to generate the timer interrupts. The default value is TRUE. If the system cannot find the APIC, the feature will be disabled. On some embedded systems or old PC, the APIC check could hang the system, so you must disable it manually. APIC requires the TSC.

Look inside `SHARK.CFG` to discover all the remaining system options, which are version dependent.

NOTE: You must recompile S.Ha.R.K. if you modify `SHARK.CFG`

1.3 Predefined Types and Constants

Tables 1.1, 1.3, 1.2, 1.4 show a subset of the predefined data types in S.Ha.R.K., a subset of the possible task states and Models and the system basic constants.

Type	*Description*
BYTE	unsigned char, [0, 255]
WORD	unsigned int, [0, 65535]
DWORD	unsigned long, [0, 0xFFFFFFFF]
TIME	unsigned long, [0, 0xFFFFFFFF]
PID	Task identifier
TASK	task
PORT	communication endpoints
CAB	cyclic asynchronous buffers

Table 1.1: Predefined types.

Identifier	*Value*
FREE	0
EXE	1
SLEEP	2
WAIT_JOIN	3
WAIT_COND	4
WAIT_SIG	5
WAIT_SEM	6
WAIT_NANOSLEEP	7
WAIT_SIGSUSPEND	8
WAIT_MQSEND	9
WAIT_MQRECEIVE	10

Table 1.2: Task states. (Note that a scheduling module can add its private task states.)

Identifier	Class
HARD_TASK_MODEL	Periodic and sporadic hard tasks
SOFT_TASK_MODEL	Periodic and aperiodic soft tasks
NRT_TASK_MODEL	Non-real-time tasks
JOB_TASK_MODEL	A task instance (job) that can be inserted into another module
DUMMY_TASK_MODEL	Model used for the Dummy Task
ELASTIC_TASK_MODEL	Elastic task, used with the Elastic Module

Table 1.3: Basic Task Models included with the default distribution (see include/kernel/model.h).

Identifier	*Value*
MAX_PROC	66
MAX_RUNLEVEL_FUNC	40
JET_TABLE_DIM	20
MAX_CANCPOINTS	20
MAX_SIGINTPOINTS	20
MAX_SCHED_LEVEL	16
MAX_RES_LEVEL	8
MAX_LEVELNAME	20
MAX_MODULENAME	20
MAX_TASKNAME	20
NIL	-1
RUNLEVEL_STARTUP	0
RUNLEVEL_INIT	1
RUNLEVEL_RUNNING	3
RUNLEVEL_SHUTDOWN	2
RUNLEVEL_BEFORE_EXIT	4
RUNLEVEL_AFTER_EXIT	5
NO_AT_ABORT	8

Table 1.4: System constants (see include/kernel/const.h).

Chapter 2

System Start-up and Termination

Each S.Ha.R.K. application starts as a sequential C program, with the classical main funcion. The multitasking environment is already initialized when the application starts. From the main task you can call any system primitive.

The system finishes when a sys_end or sys_abort function is called, or when the last user task is terminated. For more information, see *The Generic Kernel Internals* chapter of the S.Ha.R.K. Kernel Architecture Manual.

The `sys_atrunlevel()` primitive allows to post some handlers, which are automatically executed by the kernel when it changes runlevel. Such functions can be issued either in the target execution environment (generally MS-DOS) or just before terminating the S.Ha.R.K. kernel, depending on the third argument of the primitive. The handlers posted through `sys_atrunlevel()` may also be called on a kernel abort due to fatal errors.

2.1 Initialization File

When the system starts, one of the things to be done before going in multitasking mode is to initialize the devices, the resources and the schedulers which will be used by the application. To do that, the Kernel calls the __kernel_register_levels__ function, that usually registers the following modules (see the S.Ha.R.K. Kernel architecture Manual for more details):

Scheduling Modules A scheduling module implements a particular Scheduling Algorithm (for example EDF, RM, Round Robin, and so on).

Resource Modules A resource module implements a shared resource access protocol (for example the semaphores, the mutexes, and so on).

Other devices Such for example the File System, and other devices that need to be initialized when the Multitasking Mode is not started yet.

The function returns a TICK value (in microseconds) that is the time that will be used for programming the periodic timer interrupt of the PC. If a value of 0 is returned, the one-shot timer is used instead (see the OSLib documentation for more informations). Typical return values range from 250 to 2000 microseconds.

Here is a typical initialization function:

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

```
{
  struct multiboot_info *mb = (struct multiboot_info *)arg;
  EDF_register_level(EDF_ENABLE_ALL);
  RR_register_level(RRTICK, RR_MAIN_YES, mb);
  CBS_register_level(CBS_ENABLE_ALL, 0);
  dummy_register_level();

  SEM_register_module();
  CABS_register_module();

  return 1000;
  }
```

As you can see, the system initialization function registers an EDF, a Round Robin and a CBS module. Then, It register a dummy Module (that usually is the last of the Scheduling Modules). For more informations about the Scheduling policies, see Section 3.2. Finally, Semaphores and CABS are registered, and a value of 1 ms Tick time is returned to initialize the PC's real-time clock.

For a survey of the architecture of the Scheduling Modules and the Resource Modules see theKernel Overview Chapter of the S.Ha.R.K. Kernel Architecture Manual. A set of Initialization functions can be found on the kernel/init directory of the S.Ha.R.K. source tree. An explanation of each registration function for each Module can be found in the S.Ha.R.K. Module Repository Manual.

After the registration of the modules in the system, the Kernel switch in Multitasking mode, and starts the execution of the handlers that the modules have posted with the `sys_atrunlevel` primitive. Usually at least one Module will create and activate a task (for example, the Round Robin Scheduling Module does that) that will start when all the handlers will be processed. The body of that task is usually called __init__() and provides an initialization for the most commonly used devices (such the keyboard, and so on) and modules. As the last thing, the function simply call the main() function, that is, the user application starts. A sample of a typical __init__() function is showed below:

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

  HARTPORT_init();
  __call_main__(mb);
  return (void *)0;
}
```

The source code of the __init__() function is usually inserted in the initialization file after the __kernel_register_levels__ function. For more information on __call_main__ see the `include/kernel/func.h` include file.

Using the new driver layer the __init()__ function slightly change. First is executed the HARTPORT_init function that initialize the Hartik Port layer (if required), then a task that close all drivers is created. The next step is the initialization of all used drivers, followed by the registration of the shutdown task that will be executed during the system shutdown procedure.

At the end the function 'main' is executed. A tipical example with the new __init()__ function is:

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

  HARTPORT_init();

  /* Create the shutdown task.  */
  /* It will be activated at RUNLEVEL SHUTDOWN */
  set_shutdown_task();

  /* Init the drivers */
  device_drivers_init();

  /* Set the shutdown task activation */
  sys_atrunlevel(call_shutdown_task, NULL, RUNLEVEL_SHUTDOWN);

  __call_main__(mb);

  return (void *)0;
}
```

ATTENTION! In some initialization files the function that activate the shutdown task is in the form:

```
#define SHUTDOWN_TIMEOUT_SEC 3

void call_shutdown_task(void *arg) {
  struct timespec t;
  sys_gettime(&t);
  t.tv_sec += SHUTDOWN_TIMEOUT_SEC;

  /* Emergency timeout to exit from RUNLEVEL_SHUTDOWN */
  kern_event_post(&t,(void *)((void *)sys_abort_shutdown),
                  (void *)0);
  task_activate(shutdown_task_PID);
}
```

This implementation say that the task has 3 seconds to perform drivers stop. After that interval the system is forced to close even is some drivers are not closed. If a longer time is needed a greater value for SHUTDOWN_TIMEOUT_SEC constant must be used. If a shutdown without the timer is preferred the function could be in the simpler form:

```
void call_shutdown_task(void *arg) {
  task_activate(shutdown_task_PID);
}
```

2.2 System primitives

Here is a list of primitives whose use is related to the system initialization.

SYS_ATRUNLEVEL

```
int sys_atrunlevel(void (*func_code)(void *),void *parm, BYTE when);
```

Description: The Generic Kernel supports the specification of the functions to be called at system initialization and termination. These functions can be registered through this system primitive; the parameters for that function are:

`f` the function to be registered;

`p` the parameter to be passed to function `f` when the function will be called;

`when` is the situation in witch that function will be called. The correct values are the following:

- `RUNLEVEL_INIT` Used when programming Modules;
- `RUNLEVEL_SHUTDOWN` The function will be called after a call to `sys_abort` or `sys_end`; The system is still in multitasking mode;
- `RUNLEVEL_BEFORE_EXIT` The function will be called when the Kernel exits from multitasking mode;
- `RUNLEVEL_AFTER_EXIT` The function is called before the system hangs (or returns to the host OS, if the proprietary extender is used).

It is also possible to specify with an OR operator a flag `NO_AT_ABORT` that disables the call to the functions if the system is exiting with a `sys_abort` function.

You can post at most `MAX_RUNLEVEL_FUNC` functions. See the S.Ha.R.K. Kernel Architecture Manual for more details.

See also: `sys_init()`, `sys_end()`.

EXIT

```
void exit(int status);
```

Description: This function call terminates the Kernel. In this phase, the Kernel tries to correctly close all the initialized modules and drivers. The functions eventually posted with the `sys_at_runlevel` call are also executed.

If called inside an event or inside an ISR, it does return to the caller. The system shutdown will start when all the current interrupts have been serviced, and the system has been rescheduled. Otherwise, this function follows the POSIX specification.

See also: `_exit()`, `sys_panic()`, `sys_atrunlevel()`.

_EXIT

```
void _exit(int status);
```

Description: Same as `exit()`. functions posted through `sys_at_runlevel` with `NO_AT_ABORT` set or functions posted with `atexit()` are not executed.

See also: `exit()`, `atexit()`, `sys_panic()`, `sys_atrunlevel()`.

SYS_PANIC

```
void sys_panic(const char * fmt, ...);
```

Description: This function call print a message then call sys_abort(333).

See also: `sys_abort()`, `sys_end()`, `sys_atrunlevel()`.

SYS_SHUTDOWN_MESSAGE

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

Description: This function call saves a message in a reserved area, that will be printed at system shutdown. It does not end the system.

See also: `sys_panic()`.

SYS_ABORT_SHUTDOWN

```
int sys_abort_shutdown(int err);
```

Description: This function will force the system to end the `SHUTDOWN` runlevel if there are system tasks which cannot be stopped. If called when the system is still in the `RUNLEVEL_RUNNING` runlevel, the function behaves like `exit()`. If called inside an OSLib event or inside an IRQ,it does return to the caller. The system shutdown will start when all the current interrupts have been serviced, and the system has been rescheduled. Otherwise, this function does not return.

SYS_SET_REBOOT

```
int sys_set_reboot(int mode);
```

Description: This function sets the reboot `mode`, which specifies what will happen after the system end. `mode` options are:

EXIT_MODE_HALT: the system will call the halt (`HLT`) instruction.

EXIT_MODE_COLD: the system will perform the cold reboot (slow reboot).

EXIT_MODE_WARM: the system will perform the warm reboot (fast reboot).

EXIT_MODE_REAL: the system will return to the real mode (**default selection**).

Chapter 3

Task Management

3.1 Task Model

S.Ha.R.K. tasks are defined using standard C functions which return a `void *` type[1] and can have one void * argument, which is passed when the task is created. A task is identified by a system-wide unique process identifier (`PID`) and a consecutive number[2].

A task has tipically a set of Quality of Service requirements that need to be fullfilled by the Kernel. The kernel uses its registered Scheduling Modules to meet the QoS required by a specified task. The QoS reuired is specified at creation time through a Task Model, that is passed to the task creation primitives.

A Task, can be *Periodic* or *Aperiodic*. Periodic tasks are automatically activated by the kernel with a desired period, whereas aperiodic tasks can either be activated by an explicit system call or upon the occurrence of a desired event.

The typical task code consists of an optional initialization of local variables and resources, followed by a (finite or infinite) loop, representing the task's body. The last instruction of such a loop must be the primitive `task_endcycle()` or the primitive `task_sleep()` which signals the end of a generic job.

The task can access a local and a global context by following the C scoping rules; the local context is defined by the local variables and the single optional input argument. The following example shows a typical task code fragment:

```
void *my_task(void *par)
{
 /* Local Context*/
 int a, b, c;

 /* Initialization\> */
 b = c = (int)par + 1;
 a = (int)par / 2;
 ...
 while (1) {
   /* Body here!*/
   ...
```

[1]for readability, that type has been called TASK.

[2]a task with PID p has a consecutive number that is proc_table[p].task_ID .

```
    ...
    task\_endcycle();
  }
}
```

`my_task()` has just one integer input argument (passed through the void * parameter) and three local variables. The life-cycle of the local variables is the same as the task one, since they are allocated on the task's stack. Obviously they retain their values between two consecutive jobs. One of the most important parameters for a real-time task τ_i is the ***deadline***, defined as the maximum time allowed for a task job to terminate. More precisely, we distinguish between the ***absolute deadline*** (denoted by d_i) specified with respect to time 0, and the *Relative Deadline* (denoted by D_i) specified with respect to the activation time $r_{i,k}$ of the k-th job of task τ_i. We have that:

$$d_i = r_{i,k} + D_i.$$

Tasks can also have different level of criticality, for example:

- `HARD` tasks are the most critical in the system. For this reason, they are subjected to a guarantee algorithm at creation time. The system enforces a strict compliance to the deadline constraint for this kind of tasks[3]. If a hard deadline is missed, the system raises an exception which, by default, results in the program termination. Recovery actions can be programmed for this kind of exception (as shown below).
- `SOFT` tasks can miss some deadline, and are scheduled in order not to jeopardize HARD tasks' schedulability. This is done through a service mechanism (see 3.2) which guarantees each soft task a predefined bandwidth (i.e., a fraction of processor utilization) while preserving the guarantee performed on hard tasks.
- `NRT` (Non Real-Time) tasks are scheduled in background according to their relative fixed priority. Typically, they are used for monitoring or debugging purposes.

The Task criticality, periodicity and the deadlines are coded into the Task Model that is passed to the creation primitive. Each new Scheduling Module can use its own Task Model to include the specific task QoS requirements.

Each task can be in one of a set of states; the states that a task can be in depend on each particular Module. For example, typical Task states can be:

- `EXE`: at any time, in the system there is only one task in the EXE state, and it is the task actually executing.
- `READY`: it includes all active tasks ready to execute, except for the currently running task.
- `SLEEP`: it includes all aperiodic tasks which terminated a job and are waiting for the next activation. Moreover, each created task (periodic or aperiodic) that has not been activated is put in the SLEEP state.
- `IDLE`: is the state of those periodic tasks which terminated a job and are waiting for the next activation.
- `BLOCKED`: it includes all the tasks blocked on a semaphore.

[3]The guarantee algorithm tries to verify that both the newly activated hard task and the previously existing ones will finish within their deadlines

3.2 The scheduling policy[4]

The S.Ha.R.K. scheduling architecture is based on a *Generic Kernel*, which does not implement any particular scheduling algorithm, but postpones scheduling decisions to external entities, the *scheduling modules*. External modules can implement periodic scheduling algorithms, soft task management through real-time servers, semaphore protocols, and resource management policies.

The Generic Kernel provides the mechanisms used by the modules to perform scheduling and resource management thus allowing the system to abstract from the specific algorithms that can be implemented. The Generic Kernel simply provides the primitives without specifying any algorithm, whose implementation resides in external modules, configured at run-time with the Initialization function.

Scheduling Modules are used by the Generic Kernel to schedule tasks, or serve aperiodic requests using an aperiodic server. The Scheduling Modules are organized into levels, one Module for each level. These levels can be thought as priority scheduling levels (their priority correspond to the order which they appear in the Initialization function). When the Generic Kernel has to perform a scheduling decision, it asks the modules for the task to schedule, according to fixed priorities: first, it invokes a scheduling decision to the highest priority module, then (if the module does not manage any task ready to run), it asks the next high priority module, and so on. The Generic Kernel schedules the first task of the highest priority non empty module's queue.

In this way, the Scheduling policy can be tuned simply modifying the Initialization function. The standard distribution of the S.Ha.R.K. Kernel includes a set of predefined Initialization functions that can be used when developing a new application. For more informations see the S.Ha.R.K. Module Manual, where each Scheduling Modules and each predefined initialization functioon are described in detail.

3.3 Task Creation

In order to run a S.Ha.R.K. task, three steps have to be performed: parameters definition, creation, and activation. To improve the system flexibility, each task can be characterized by a large number of parameters, most of which are optional. For this reason, a set of structures derived from `TASK_MODEL` structure have been introduced to simplify the parameters' definition phase. The first thing to do in order to define a task is to declare a Model variable and initialize it using the pmacro provided (see the S.Ha.R.K. Module Manual for more informations).
Once the task's parameters have been set, the task can be created using the `task_create` or the `task_createn` system call.

TASK_CREATEN and TASK_CREATE

```
PID task_createn(char *name, TASK (*body)(), TASK_MODEL *m, ...);
```

```
PID task_create(char *name, TASK (*body)(), TASK_MODEL *m, RES_MODEL *r);
```

Description: `task_createn` creates a new task. `name` is a pointer to a string representing the task name; `body()` is a pointer to the task body (i.e. the name of the C function containing the task code); `m` specifies the Model that contain the QoS specification of the task (the value can not be equal to NULL). Then, follow a list of Resource Models terminated with NULL (see the S.Ha.R.K. Module Manual for the available Task Models and Resource

[4]This section is derived from the Kernel overview chapter of the S.Ha.R.K. Architecture Manual.

Models). `task_create` is a redefinition of `task_createn` that can be used when there is at least one Resource Model to be passed to the creation primitive.

Return value: The function returns the identifier of the newly created task, or -1 if the task creation fails (in this case the `errno()` system call can be used to determine the error's cause).

See also: `task_activate()`, `task_kill()`.

Example

```
int main(int argc, char **argv)
  {
    HARD_TASK_MODEL m;
    hard_task_default_model(m);
    hard_task_def_wcet(m,ASTER_WCET);
    hard_task_def_mit(m,10000);
    hard_task_def_group(m,1);
    hard_task_def_ctrl_jet(m);

    p1 = task_create("Aster",aster,&m,NULL);
    if (p1 == -1) {
      perror("Error:  Could not create task <aster> ...");
      sys_end();
      exit(-1);
    }
  }
```

3.4 Group Creation

Group creation is a feature provided by S.Ha.R.K. that allows a user to create a set of tasks. The group creation differs from the creation made by the task_create and task_createn primitives because in group creation the acceptance test is done for the whole set of task (and not for every task in sequence) only when every task which belong to the set has been initialized in the system. After the acceptance test, the user have to inquire the Scheduling Module to see the tasks that have been accepted and successfully created in the system.

The primitives provided by S.Ha.R.K. to support group creation are:

- group_create
- group_create_accept
- group_create_reject

The documentation about group creation can be found in the *Group creation HOWTO* available on the S.Ha.R.K. website.

3.5 Task Activation and Termination

When a task is created, it is put in the `SLEEP` state, where it is kept until activation, which can be triggered by an external interrupt or by an explicit `task_activate()` primitive). Periodic jobs that complete execution are handled by the registered Scheduling Modules (usually they are put in an `IDLE` state or similar), from which they will be automatically re-activated by the system timer. Aperiodic jobs that complete execution return to the `SLEEP` state, where they wait for an explicit re-activation.

Some scheduling models (such as the EDF and RM modules) support release offsets. In an offset is given in the task model, the `task_activate()` will put the task in the `IDLE` state, waiting for the first release to occur.

A task can be destroyed using the `task_kill()` system call, that frees its descriptor. A task can kill itself using the `task_abort()` system call.

In S.Ha.R.K., tasks can be members of groups to allow simultaneous activation or termination. A task can be put in a group through a macro that works on the task model passed at task creation. The name of the macro depends on the name of the Task Model used; usually its name is like `XXX_task_def_group(group_number)`, where XXX is the name of the task Model; the group_number 0 indicates that a task belongs to no groups.

Task cancellation, join, the cleanup handlers and the task specific data works as the POSIX standard; only the name of the primitives are changed from `pthread_XXX` to `task_XXX`. In any case, the `pthread_XXX` versions are available for POSIX compatibility.

Warning: `task_kill()` kills a task only if the cancellation type of the task is set to asynchronous. If the cancellation type is set to deferred, the task will terminate only when it reach a cancellation point.

TASK_ACTIVATE

```
int task_activate(PID p);
```

Description: It activates task `p`. Usually the activation will insert the task into the ready queue. (If the task has an offset, the task will be put in the ready queue after the offset.) Returns 0 in case of success or -1 in case of error; the `errno` variable is set to `EUNVALID_TASK_ID`.

See also: `task_create()`, `task_kill()`, `group_activate()`.

TASK_KILL

```
int task_kill(PID p);
```

Description: It asks for a cancellation of the task p. It returns -1 in case of error, 0 otherwise. If an error occurred, the errno variable is set to `EINVALID_KILL`. A task which has the NO_KILL flag set can not be killed. If the task has already been killed but it is not died yet, the primitive does nothing. A task that has the cancellation type set to asynchronous will die just when it will be scheduled again by the system; instead, if the cancellation type is set to deferred, the task will die only at the reaching of a cancellation point. This function is the correspondent of the `pthread_cancel()` primitive.

See also: `task_create()`, `task_activate()`, `group_kill()`.

TASK_ABORT

`void task_abort(void *returnvalue);`

Description: It aborts the calling task, removing it from the system. If the task is joinable, the return value will be stored by the kernel and given to any task that calls a task_join primitive on the died task.

See also: `task_create()`, `task_activate()`, `task_kill()`.

TASK_BLOCK_ACTIVATION

`int task_block_activation(PID p);`

Description: It blocks all explicit activation of a task made with `task_activate` and `group_activate`. The activations made after this call are buffered (counted) in an internal counter. It returns 0 in case of success or -1 in case of error. In the latter case, `errno` is set to `EUNVALID_TASK_ID`. If the activations were already blocked, it does nothing.

See also: `task_unblock_activation()`, `task_activate()`.

TASK_UNBLOCK_ACTIVATION

`int task_unblock_activation(PID p);`

Description: It unblocks the activations of a task after a call to task_block_activation. After this call, the task can be explicitly activated. It returns the number of buffered activations, or -1 if an error occurred. If an error occurred, the errno variable is set to EUNVALID_TASK_ID. If the activations were not blocked, it simply returns 0. Note that the primitive simply returns the number of buffered activations, *without* activating the task.

See also: `task_block_activation()`, `task_activate()`.

GROUP_ACTIVATE

`int group_activate(WORD g);`

Description: It activates all tasks belonging to group **g**. Returns 0 in case of success or -1 in case of error; the `errno` variable is set to `EUNVALID_GROUP`.

See also: `task_create()`, `task_activate()`, `group_kill()`.

GROUP_KILL

`void group_kill(WORD g);`

Descrizione: It kills all tasks belonging to group **g**. It returns -1 in case of error, 0 otherwise. If an error occurred, the errno variable is set to `EUNVALID_GROUP`. The kill request to a single task that belong to a group is done in a way similar to that done in the primitive `task_kill()`.

See also: `task_create()`, `task_activate()`, `group_activate()`, `task_kill()`.

3.6 Task Instances

S.Ha.R.K. supports the concept of instance for its task. A typical task function is composed by an initialization part and a body part that does the work for that the task was created; for example:

```
void *mytask(void *arg)
{
  <initialization part>
  for (;;) {
    <body>
    ...
    task_endcycle();
  }
}
```

In the example, the task will never terminate, and it also calls the `task_endcycle` primitive to say to the Kernel that the current instance is terminated[5].

TASK_ENDCYCLE

```
void task_endcycle(void);
```

Description: It terminates the currently executing job of the calling task. The behaviour of this primitive may sligtly change depending on the Scheduling Module registered at initialization time. Tipically, the `task_endcycle` primitive suspends the task until an automatic reactivation that is made internally by the Kernel. Moreover, the `task_endcycle()` primitive usually keeps track of pending activations[6]. Previous versions of the kernel supported a `task_sleep()` primitive with similar behavior. That function is currently unsupported. The primitive is a cancellation point.

Implementation: This primitive is implemented as task_message(NULL, 1);[7]

See also: `task_activate`.

3.7 Task (thread) specific data

These functions works in a way equal to their POSIX counterparts. These primitives are used for managing task specific data, that are a few data variables that can be referred in a common way independently from the task that asks for it. The system also ensures a proper cleanup when the task is killed. As an example, the `errno` variable can be thought as a task specific data. In this manual only their interfaces are described; for more informations, see the POSIX standard.

TASK_KEY_CREATE

```
int task_key_create(task_key_t *key, void (*destructor)(void *));
```

---

[5]The concept of instance is introduced into S.Ha.R.K. because in that way the Kernel can directly support task Quality Of Service parameters like deadlines, periods and so on in a native way. Note that the concept of instance is not covered by the POSIX standard, that only support a fixed priority scheduler. In POSIX, a periodic task can only be implemented using the Real-Time extensions and in particular using the Timer feature. S.Ha.R.K. implements also that approach, however the native primitives are better in terms of efficiency.

[6]There is a pending activation when a task is activated before the current instance has finished. In this case, if the `task_endcycle()` primitive is called and there is a pending activation, it simply does nothing.

[7]Note on the implementation: this primitive is implemented as `task_message(NULL, 1);`

Description: It creates a task key that can be used to refer a task_specific data. The name of the POSIX counterpart is `pthread_key_create`.

TASK_GETSPECIFIC

```
void *task_getspecific(task_key_t key);
```

Description: It gets the current value for the key (note that the value of the key vary from task to task). The name of the POSIX counterpart is `pthread_getspecific`.

TASK_SETSPECIFIC

```
int task_setspecific(task_key_t key, const void *value);
```

Description: It sets the current value for the key. The name of the POSIX counterpart is `pthread_setspecific`.

TASK_KEY_DELETE

```
int task_key_delete(task_key_t key);
```

Description: It deletes the current key. The name of the POSIX counterpart is `pthread_key_delete`.

3.8 Task cancellation

These primitives are used when managing task cancellation. They are directly derived from the POSIX standard. Nothe that the POSIX interface is also available.

TASK_CLEANUP_PUSH

```
void task_cleanup_push(void (*routine)(void *), void *arg);
```

Description: It pushes the specified cancellation cleanup handler routine onto the cancellation cleanup stack. The name of the POSIX counterpart is `pthread_cleanup_push`.

TASK_CLEANUP_POP

```
void task_cleanup_pop(int execute);
```

Description: It removes the routine at the top of the cancellation cleanup stack of the calling thread. If `execute` is not equal 0, the routine previously pushed is called. The name of the POSIX counterpart is `pthread_cleanup_pop`.

TASK_TESTCANCEL

```
void task_testcancel(void);
```

Description: creates a cancellation point in the calling task. The primitive has no effect if cancelability is disabled. The name of the POSIX counterpart is `pthread_testcancel`.

TASK_SETCANCELSTATE

`int task_setcancelstate(int state, int *oldstate);`

Description: This primitive sets the cancelability state of the calling thread to the indicate *state and* returns the previous cancelability state at the location referenced by *oldstate*. Legal values for state are `TASK_CANCEL_ENABLE` and `TASK_CANCEL_DISABLE`. `pthread_setcancelstate` is the name of the POSIX counterpart.

TASK_SETCANCELTYPE

int task_setcanceltype(int type, int *oldtype);

Description: This primitive sets the cancelability type of the calling thread to the indicate *type* and returns the previous cancelability type at the location referenced by *oldtype*. Legal values for state are `TASK_CANCEL_DEFERRED` and `TASK_CANCEL_ASINCHRONOUS`. The name of the POSIX counterpart is `pthread_setcanceltype`.

3.9 Join

The join primitives allow a task to wait for the termination of another task[8]. The return value of the terminated task is passed to the join primitive and the caller can use the value. These primitives are directly derived from the POSIX standard. It means that a join can be done only on a joinable task. But note that when a task created with the creation primitives[9] starts it is *not* in the joinable state[10]. This behaviour differs from the standard behavior of the POSIX standard, which specifies that every new task shall be in the joinable state. However, S.Ha.R.K. provides also the `pthread_create` primitive that is *fully compliant* with the standard. Finally, a S.Ha.R.K. task can switch between the jonable and non-joinable state using the primitives `task_joinable` and `task_unjoinable`[11].

TASK_JOIN

`int task_join(PID p, void **value);`

Description: The primitive suspends the execution of the calling task until the task p terminates, unless the task p has already terminated. On return from a successful `task_join` call with a non-`NULL` *value* argument, the value returned by the thread through a task_abort shall be made available in the location referenced by *value*. When the primitive returns successfully the target task has been terminated. The primitive returns 0 in case of success, otherwise it returns `EINVAL` if the value `p` does not refer to a task that can be joined, `ESRCH` if the value `p` does not refer to a valid task, and `EDEADLK` if a deadlock was detected. The name of the POSIX counterpart is `pthread_join`.

TASK_JOINABLE

`int task_joinable(PID p);`

Description: This function set the detach state of a task p to joinable. This function is not present in Posix standard. It returns ESRCH if p is non a valid task.

[8]not of an instance of a task!

[9]`task_create`, `task_createn`, `group_create`

[10]this is done to remain similar to the previous versions of the Hartik Kernel...

[11]In the POSIX standard, only the `pthread_detach` primitive is available.

TASK_UNJOINABLE

```
int task_unjoinable(PID p);
```

Description: This function sets the detach state of a task to detached. The name of the POSIX counterpart is `pthread_detach`. The function returns `EINVAL` if `p` can not be joined (or currently a task has done a join on it), or `ESRCH` if `p` is not correct.

3.10 Preemption control

S.Ha.R.K. provides two primitives that set the preemptability of a task. A non-preemptive task can not be preempted by another task; interrupts are handled in the usual way. These primitives can be used to implement short critical sections. Note the difference between this kind of non -preemption and the interrupt disabling done using kern_cli and kern_sti: in the latter case, interrupt can not preempt the critical sections. A new task usually starts in a preemptive state.

TASK_NOPREEMPT

void task_nopreempt(void);

Description: After the call of this primitive, the task is non-preemptive.

TASK_PREEMPT

void task_nopreempt(void);

Description: After the call of this primitive, the task become again preemptive.

3.11 Suspending a task

The following system calls can be used by a task to suspend itself for a known or unknown time. (Note: it is dangerous to use these system calls in a hard real-time task.)

TASK_DELAY

```
void task_delay(DWORD t);
```

Description: It causes the calling task to be blocked for at least `t` microseconds. Note that `t` is the *minimum* delay time. In facts, after `t` microseconds the task is inserted in the ready queue and can be delayed by higher priority tasks. This function was inherited from the previous versions of Hartik. Please, use the POSIX counterpart `nanosleep` instead!

3.12 Job ExecutionTime (JET) estimation

S.Ha.R.K. provides a set of primitives that allows to precisely estimate the system load. These primitives can be used to tune the parameters that are given at task creation, and to get statistics about the system load.

The execution time estimation is done on a task basis. That is, S.Ha.R.K. provides three primitives that allows the user to estimate the JET of every task. For every task, it is possible to know the mean execution time, the maximum execution time, the time consumed by the current instance and the time consumed by the last `JET_TABLE_DIM` instances.

The user have to explicitly enable the Kernel to record the JET informations for a specific task. This is done at task creation time; usually a macro that enable the JET is provided in the definition of every task model (see the S.Ha.R.K. Module Manual).

Here is an example of the use of the JET functions:

```
/* The Goofy Task */
void *goofy(void *arg)
{
  int i;
  for (;;) {
    for (i=0; i<100; i++)
      kern_printf("Yuk!\t");
    task_endcycle();
  }
}

PID goofy_PID;

/* a NRT task that never finish */
void *jetcontrol(void *arg)
{
  TIME sum, max, curr, last[5];
  int nact;
  for (;;) {
      if (jet_getstat(p, &sum, &max, &nact, &curr) == -1)
        continue;
      for (j=0; j<5; j++) last[j] = 0;
      jet_gettable(p, &last[0], 5);
      printf_xy(1,20,WHITE,"goofy_PID=%d mean=%d max=%d nact=%d",
                goofy_PID,sum/(nact==0 ?  1 :  nact), max, nact);
      printf_xy(1,21,WHITE,"L1=%d L2=%d L3=%d L4=%d L5=%d",
                last[0], last[1], last[2], last[3], last[4]);
  }
}

int main(int argc, char **argv)
{
  ...
  /* The task goofy is created, specifying that the Kernel
     should take care of the JET data */
  HARD_TASK_MODEL m;
  hard_task_default_model(m);
  /* ...  other hard_task_XXX macros */
  hard_task_def_ctrl_jet(m); /* JET enabling */
  goofy_PID = task_create("Goofy", goofy, &m, NULL);
  /* ...  creation of the JET control task, and so on */
}
```

JET_GETSTAT

int jet_getstat(PID p, TIME *sum, TIME *max, int *n, TIME *curr);

Description: This primitive returns some JET informations about the task p. The informations retrieved are stored into the following parameters:

`sum` is the task total execution time since it was created or since the last call to the `jet_delstat` function;

`max` is the maximum time used by a task instance since it was created or since the last call to the `jet_delstat` function;

`n` is the number of terminated instances which sum and max refers to;

`curr` is the total execution time of the current instance.

If a parameter is passed as `NULL` the information is not returned. The function returns 0 if the `PID` passed is correct, `-1` if the PID passed does not correspond to a valid PID or the task does not have the `JET_ENABLE` bit set.

JET_DELSTAT

int jet_delstat(PID p);

Description: The primitive voids the actual task execution time data mantained by the Generic Kernel. The function returns 0 if the PID passed is correct, `-1` if the PID passed does not correspond to a valid PID or the task does not have the `JET_ENABLE` bit set.

JET_GETTABLE

int jet_gettable(PID p, TIME *table, int n);

Description: The primitive returns the last `n` execution times of the task `p`. If the parameter n is less than 0, it returns only the last values stored since the last call to `jet_gettable` (up to a maximum of `JET_TABLE_DIM` values). If the value is greater than 0, the function returns the last `min(n, JET_TABLE_DIM)` values registered. The return value is `-1` if the task passed as parameter does not exist or the task does not have the `JET_ENABLE` bit set, otherwise the number of values stored into the array is returned. The table passed as parameter should store at least `JET_TABLE_DIM` elements.

Chapter 4

Synchronization and communication

This chapter describes the tasks' interaction capabilities provided by the S.Ha.R.K. kernel. In order to improve the programming flexibility without jeopardizing the hard tasks' a priori guarantee, the kernel implements different mechanisms.

In general, hard tasks should not use system calls that can cause an unbounded (or unknown) blocking time, since they can jeopardize the system schedulability. For example, consider Figure 4.1: in this case there are two periodic tasks, τ_1 (execution time $C_1 = 5$ and period $T_1 = 15$) and τ_2 (execution time $C_2 = 2$ and period $T_2 = 4$). Since the total utilization factor is $U = 1/2 + 1/3 = 5/6 < 1$, the system is schedulable by EDF, but if τ_1 blocks for 8 time units at time $t = 4$, τ_2 misses a deadline at time $t = 16$.

For efficiency reasons, the system does not perform any check to avoid the use of blocking primitives in hard tasks; so this aspect is left to the programmer responsibility.

4.1 POSIX Semaphores

The primitives described in this Section covers the semaphore mechanism interface that can be used by the S.Ha.R.K. applications. The semaphore interface directly follows the POSIX semaphore interface; the S.Ha.R.K. Kernel add also some other primitives that allows to increment/decrement the semaphore counter by more than one unit at a time.

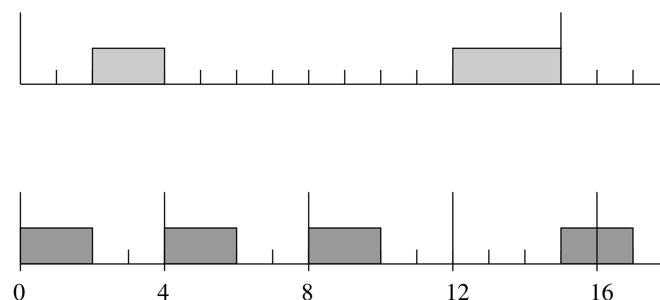

Figure 4.1: EDF Scheduling - Overload due to a task_delay().

These primitives can be used both for synchronization and mutual exclusion. It is worth noting that the traditional semaphore mechanism can cause unbounded *priority inversion*, so it is not suitable for hard real-time tasks. Concerning the synchronization, we note that the guarantee mechanism does not take synchronization into account; therefore the programmer should avoid to explicitly synchronize hard tasks by means of blocking primitives. It is instead possible to use a weak synchronization between hard real-time tasks, realized through non-blocking semaphores.

Only `SEM_NSEMS_MAX` semaphores can be created in the system. If an application needs to use the POSIX semaphores, it have to add the call to the function

`void SEM_register_module(void);`

into the `__kernel_register_levels__` function of the initialization file (see Volume III - S.Ha.R.K. Modules).

In this section will be briefly described the POSIX semaphore interface[1]. For a complete reference see the POSIX standard (the Linux manpage also works well).

SEM_INIT

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Description: It is used to initialize a semaphore referred by sem. The value of the initialized semaphore is `value`. The pshared argument is ignored. After the call to the primitive, the sem value can bve used to refer the semaphore.

Return value: on successful completion, the function initializes the semaphore in sem and returns 0. Otherwise, it returns -1 and `errno` is set according to the POSIX standard.

See also: sem_wait(), sem_trywait, sem_post(), sem_destroy().

Example

```
sem_t mutex;
TASK demo(void *arg)
{
  ...
  /* The task enters a critical section protected by a mutex semaphore */
  sem_wait(&mutex);
  <critical section>
   sem_post(&mutex);
  ...
}
int main(int argc, char**argv)
{
  ...
  sem_init(&mutex,0,1);
  ...
}
```

SEM_DESTROY

[1] This section only described unnamed semaphores. The interface for named semaphores is also provided, althoutgh it does not use a file system but resolve the names internally (as allowed by the POSIX 1003.13 PSE51 profile).

```
int sem_destroy(sem_t *sem);
```

Description: It is used to destroy the semaphore indicated by sem. Only a semaphore that was created using sem_init() may be destroyed using sem_destroy(). **Warning**: This system call does not check if the semaphore queue is empty or not, and does not awake tasks blocked on the semaphore. The programmer has to make sure that `s` is free before destroying it.

Return value: on successful completion, the function destroys the sem semaphore and returns 0. Otherwise, it returns -1 and `errno` is set according to the POSIX standard.

See also: sem_init().

SEM_WAIT and SEM_TRYWAIT

`int sem_wait(sem_t *sem);`

`int sem_trywait(sem_t *sem);`

Description: `sem_wait` is used to lock the semaphore referenced by `sem`. If the semaphore value is currently zero, then the calling task shall not return from the call to `sem_wait()` until it either locks the semaphore. `sem_trywait` locks the semaphore referenced by `sem` only if the semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, it does not lock the semaphore. `sem_wait` is a cancellation point.

Return value: on successful completion the functions return 0. Otherwise, they return -1 and `errno` is set according to the POSIX standard.

See also: sem_post().

SEM_XWAIT

`BYTE sem_xwait(sem_t *s, int n, int wait);`

Description: `sem_xwait()` is a non-portable extension to the POSIX semaphores that decreases the semaphore counter by `n`. If the counter is greater than or equal to `n` and there are no tasks blocked on semaphore `s`, the counter is decreased by `n` and `sem_xwait` returns 0, otherwise the system call's behavior depends on the `b` parameter. If `wait` is `BLOCK`, the calling task blocks on the semaphore, if `wait` is `NON_BLOCK` `sem_xwait` returns -1, errno is set to EAGAIN and the calling task does not block. The semaphore queue is ordered using a FIFO strategy, in order to avoid starvation. Hard tasks should not use blocking system calls, so it is suggested to use `sem_trywait()/xwait()` (only with `b = NON_BLOCK`). `sem_xwait` is a cancellation point.

Return value: on successful completion, the function returns 0. Otherwise, it returns -1 and `errno` is set according to the POSIX standard.

See also: sem_wait(), sem_trywait(), sem_post().

Example

```
sem_t sync;
TASK demo(void *arg)
{
  ...
  /* The demo task synchronizes itself */
  /* with the wake task, waiting */
  /* for 5 signals on the sync semaphore */
  sem_xwait(&sync, 5, BLOCK);
  ...
}
TASK wake(void *arg)
{
  while (1)
```

```
    {
      ...
      sem_xsignal(sync, 1);
      ...
      task_endcycle();
    }
  }

  void main(void)
  {
    ...
    sem_init(&sync,0,0);
    ...
  }
```

SEM_POST

`int sem_post(sem_t *sem);`

Description: It unlocks the semaphore referenced by sem by performing the semaphore unlock operation on that semaphore. If the semaphore queue is not empty and the first task in the queue requests a feasible counter decrement, it can be awaken. The task is put in the ready queue and the scheduler is invoked: for this reason this system call can cause a preemption. The semaphore queue is a FIFO queue: tasks are awoken in a FIFO order according to resource availability.

Return value: on successful completion, the function destroys the sem semaphore and returns 0. Otherwise, it returns -1 and `errno` is set according to the POSIX standard.

See also: sem_wait(), sem_trywait().

SEM_XPOST

`int sem_xpost(sem_t *s, int n);`

Description: `sem_xpost()` is a non-portable extension to the POSIX semaphores that implements the classical signal primitive on semaphore `s`, increasing the counter by `n`. If the semaphore queue is not empty and the first task in the queue requests a feasible counter decrement, it can be awaken. The task is put in the `READY` queue and the scheduler is invoked: for this reason this system call can cause a preemption. The semaphore queue is a FIFO queue: tasks are awoken in a FIFO order according to resource availability.

Return value: on successful completion, the function destroys the sem semaphore and returns 0. Otherwise, it returns -1 and `errno` is set according to the POSIX standard.

See also: `sem_init()`, `sem_wait()`, `sem_destroy()`.

Example: see `sem_wait()`.

SEM_GETVALUE

`int sem_getvalue(sem_t *sem, int *sval);`

Description: `sem_getvalue()` updates the location referenced by the sval argument to have the value of the semaphore referenced by sem without affecting the state of the semaphore. If sem is locked the value returned by sem_getvalue is a negative number whose absolute value represents the number of processes waiting for the semaphore at some unspecified time during the call.

Return value: on successful completion, the function destroys the sem semaphore and returns 0. Otherwise, it returns -1 and `errno` is set according to the POSIX standard.

See also: `sem_init()`, `sem_wait()`, `sem_destroy()`.

4.2 Internal Semaphores

When developing a complex driver of the kernel, a designer usually needs to manage a lot of shared resources that have to be accessed in mutual exclusion, and needs also a lot of synchronization points that are not cancellation points. For these pourposes the POSIX semaphores are not good because they are limited in number and they are cancellation points.

For this pourpose the S.Ha.R.K. Kernel provides a sort of lightweight semaphores called *internal* semaphores, that fulfill the designer needs[2]: they are not cancellation points and there is no limit on the number of semaphores that can be created in a system[3]. The interface of the Internal semaphores is very similar to POSIX semaphore interface.

To use the Internal Semaphores, you don't need to call any registration function at kernel startup time.

INTERNAL_SEM_INIT

```
void internal_sem_init(internal_sem_t *s, int value);
```

Description: It initializes the internal semaphore `s` with a specified `value`.

INTERNAL_SEM_WAIT

```
void internal_sem_wait(internal_sem_t *s);
```

Description: It implements a blocking wait. the semaphore counter is decremented by one.

INTERNAL_SEM_TRYWAIT

```
int internal_sem_trywait(internal_sem_t *s);
```

Description: It implements a non-blocking wait. It returns 0 if the counter is decremented, -1 if not.

INTERNAL_SEM_POST

```
void internal_sem_post(internal_sem_t *s);
```

Description: It implements a post operation.

[2]The existence of two type of semaphores is not new in Kernel development; For example, the Linux Kernel differentiate the semaphores used by the applications and the semaphors used by the Kernel.

[3]Only 8 bytes are taken for each internal semaphore. In some sense the internal semaphores are similar to the POSIX mutexes...

INTERNAL_SEM_GETVALUE

int internal_sem_getvalue(internal_sem_t *s);

Description: It returns a value greater or equal 0 if there are no tasks blocked on s, -1 otherwise.

4.3 Mutexes and Condition Variables

The primitives described in this section allows the user to define and use *mutexes* and *condition variables*. A mutex can be thought as a binary semaphore initialized to 1. In that way, a critical section can be specified using the *mutex_lock* and *mutex_unlock* primitives. Moreover, using condition variables a task can block itself waiting for an event.

The provided implementation extends the POSIX standard mutex functions implementing protocols like Stack Resource Policy and Non Preemptive Protocol, that are not part of the standard. To do that, the mutex initialization interface is different from the standard to allow the specification of the various policies. In any case, the standard interface is provided based on the extended interface.

4.3.1 Mutex attributes

A mutex can be used to implement critical sections that uses different policies (for example, the Priority Inheritance, Priority Ceiling or Stack Resource Policy protocol). The S.Ha.R.K. Kernel provides a set of structures derived from the basic structure mutexattr_t[4] that allow to handle the specification of different policies.

The mutex attributes are different foe every policy, that is implemented by a Resouce Module. To see the the description of the mutex attributes for every policy, look at the S.Ha.R.K. Modules Manual.

4.3.2 Functions

This subsection describes the functions that handle mutexes and condition variables.

MUTEX_INIT

```
int mutex_init(mutex_t *mutex, const mutexattr_t *attr);
```

Description: The mutex_init function intuiualizes the mutex referenced by *mutex* with attributes specified by *attr*. *attr* shall be not equal NULL. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

Return value: on successful completion the functions return 0. Otherwise, they return -1 and `errno` is set according to the POSIX standard.

See also: mutex_destroy().

MUTEX_DESTROY

```
int mutex_destroy(mutex_t *mutex);
```

[4]Similar to the pthread_mutexattr_t structures of the POSIX standard.

Description: The mutex_destroy function destroys the mutex object referenced by mutex. It is safe to destroy an initialize mutex that is unlocked.

Return value: on successful completion the functions return 0. Otherwise, they return -1 and `errno` is set according to the POSIX standard.

See also: mutex_init().

MUTEX_LOCK

```
int mutex_lock(mutex_t *mutex);
```

Description: The mutex_lock function locks an unlocked mutex. If the mutex is already locked , the calling thread waits until the mutex becomes available. the behaviour of the function may change depending on the particular policy passed with the mutexattr_t parameter at mutex initialization. The function is *not* a cancellation point.

Return value: on successful completion the functions return 0. Otherwise, they return -1 and `errno` is set according to the POSIX standard.

MUTEX_TRYLOCK

```
int mutex_lock(mutex_t *mutex);
```

Description: The mutex_trylock function is identycal to mutex_lock except that if the mutex us locked when the function is called, the calling task does not block but returns -1 and an errno value of EBUSY, as specified by the POSIX standard.

Return value: on successful completion the functions return 0. Otherwise, they return -1 and `errno` is set according to the POSIX standard.

MUTEX_UNLOCK

```
int mutex_unlock(mutex_t *mutex);
```

Description: The mutex_unlock function is called by the owner of the mutex object to release it. If there are thread blocked on the mutex object referenced by mutex when mutex_lock is called, the mutex becomes available, and the task that will acquire the mutex depends on the policy with that the mutex was initialized.

Return value: on successful completion the functions return 0. Otherwise, they return -1 and `errno` is set according to the POSIX standard.

COND_INIT

```
int cond_init(cond_t *cond);
```

Description: The function initializes the condition variable referenced by cond.

Return value: on successful completion the functions return 0. Otherwise, they return -1 and `errno` is set according to the POSIX standard.

COND_DESTROY

```
int cond_destroy(cond_t *cond);
```

Description: The function destroys the given condition variable specified by cond.

Return value: on successful completion the functions return 0. Otherwise, they return -1 and `errno` is set according to the POSIX standard.

COND_SIGNAL and COND_BROADCAST

```
int cond_signal(cond_t *cond);
```

```
int cond_broadcast(cond_t *cond);
```

Description: The function cond_signal unblocks at least one of the threads that are blocked on the specified condition variable cond. The function cond_broadcast unblocks all threads currently blocked on the specified condition variable cond. These functions have no effect if there are no threads currently blocked on cond.

Return value: on successful completion the functions return 0. Otherwise, they return -1 and `errno` is set according to the POSIX standard.

COND_WAIT and COND_TIMEDWAIT

```
int cond_wait(cond_t *cond, mutex_t *mutex);
```

```
int cond_timedwait(cond_t *cond, mutex_t *mutex, const struct timespec *abstime);
```

Description: These functions are used to block on a condition variable. They shall be called with *mutex* locked by the calling task. These functions release mutex and cause the calling task to block on the condition variable cond. Upon successful return, the mutex is locked and is owned by the calling task. When using condition variables, there is always a Boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the `cond_wait` or `cond_timedwait` functions may occur. Since the return from `cond_wait` or `cond_timedwait` does not imply anything about the value of this predicate, the predicate should be re-evaluated upon each return.

The `cond_wait` and cond_timedwait functions are cancellation points. When the cancelability enable state of a task is set to `TASK_CANCEL_DEFERRED`, a side effect of acting upon a cancellation request while in a condition wait is that the mutex is (in effect) reaquired before calling the first cancellation cleanup handler. To ensure a correct cancellation, a cleanup function should be pushed before the `cond_wait` call (in case of cancellation it simply unlock the mutex).

The `cond_timedwait` function is the same as the `cond_wait` function except that an error is returned if the absolute time specified by abstime passes before the condition cond is signaled or broadcasted, or if the absolute time specified by abstime has already been passed at the time of the call.

Return value: on successful completion the functions return 0. Otherwise, they return -1 and `errno` is set according to the POSIX standard.

4.4 Communication Ports[5]

S.Ha.R.K. communication ports allow tasks to exchange messages. Each port is uniquely identified by a symbolic name (i.e., a string of characters); a task willing to use this communication facility has to open the channel using the `port_create()` call, thus becoming the owner of the resource. Any other task that wants to use this communication end-point to send or receive data needs to connect to it by using the `port_connect()` primitive.

S.Ha.R.K. offers three types of ports:

- `STREAM`: it is a one-to-one communication facility, which can be opened either by the reader or by the writer task. The task executing the `port_create()` must specify the message size and maximum number of messages in the queue. The task executing the `port_connect()` must only specify the size of the messages it wants to receive/send, which can be different from the one specified by the owner. For example, a task may open a port for reading messages of 4 bytes, while another task can connect to it to write one-byte messages. This mechanism turns out to be useful for character oriented device drivers which need to fill a given structure, before the message can be processed further by a higher-level task.

[5]The S.Ha.R.K. communication ports are directly derived from the previous varsions of the Hartik Kernel.

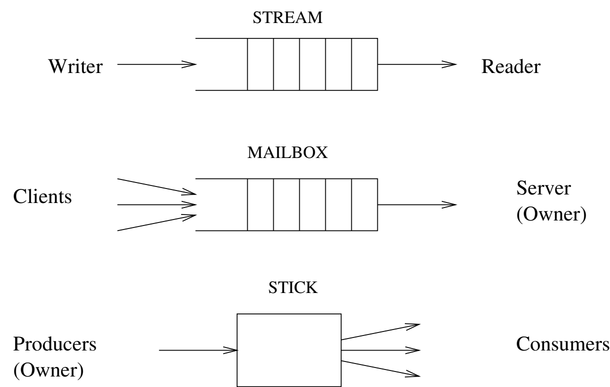

Figure 4.2: HARTIK ports.

- MAILBOX: it is a many-to-one communication facility, thought for being used in classical client/server mechanisms. This kind of port can only be opened by the reader task (the server) which wants to receive data from writer tasks (the clients). Message size is fixed and defined by the reader.
- STICK: it is a one-to-many communication facility intended to be used for exchanging periodic state-messages, for which the most recent information is relevant. It can be opened only by the (unique) writer task and the reading tasks must connect to it. It contains just one message and any new message posted by the writer will overwrite the previous one. Messages are non-consumable: a reader task can perform many readings of a given message until the writer posts a new one.

The first two kinds of port implement the synchronous communication paradigm, while STICK ports implement an asynchronous (state-message) paradigm. It is worth noting that in order to protect the internal data structures, STREAM ports use semaphores for synchronizing the accesses, STICK ports just use a mutual exclusion semaphore, and the MAILBOX ports use both kinds of semaphores.

For this reason, MAILBOX and STICK ports should not be used by critical tasks, whereas STREAM ports can be used by any task requiring a state-message non-blocking semantics. Moreover, the execution time of a transaction depends on the message size (the message is copied in/from the buffer when a send/receive is performed). The semantics associated with each port is graphically illustrated in Figure 4.2.

An application that uses the communication ports, must register the HARTPORT Module. Please see Volume III - S.Ha.R.K. Modules for details.

PORT_CREATE

```
PORT port_create(char *name, int dim, int num, int type, int mode);
```

Description: It opens the port identified by the string `name`. The argument `dim` specifies the message size in bytes, `num` specifies the queue size, `type` the port type (STREAM, MAILBOX, or STICK), and `mode` the access mode (READ or WRITE).

Return Value: The primitive returns the port identifier, which identifies the connection between the port and the task, and not the port itself, which is identified through its name. A return value -1 indicates that an error is occurred.

See also: `port_delete()`, `port_connect()`, `port_disconnect()`, `port_send()`, `port_receive()`.

Example:

```
TASK demo(void)
{
  PORT p;
  char msg[6];
  ...
  /* Demo task, of NRT type, opens the "goofy" port */
  /* and sends a message of 6 bytes.  */
  p = port_create("goofy", 6, 8, STREAM, WRITE);
  ...
  port_send(p, msg, BLOCK);
}

TASK duro(void)
{
  PORT q;
  char msg[2];
  /* Duro task (HARD) connects to the "goofy" */
  /* port and receives messages of 2 bytes */
  q = port_connect("goofy", 2, STREAM, READ);
  while (condition) {
    ...
    if (port_receive(q, msg, NON\_BLOCK)
    {
      <action 1>; /* Ready Message!  */
    }
    else
    {
      <action 2>; /* Message not Ready!  */
    }
    ...
    task_endcycle();
  }
}
```

PORT_DELETE

```
void port_delete(PORT p);
```

Description: It destroys the port identified by p.

See also: port_create(), port_connect(), port_disconnect(), port_send(), port_receive().

Example: see the example at page 39.

PORT_CONNECT

```
PORT port_connect(char *name, int dim, int type, int mode);
```

Description: It connects the calling task to the port identified by name. The argument dim specifies the message size in bytes, type the port type (STREAM, MAILBOX, or STICK), and mode the access mode (READ or WRITE). If the port has not been opened by port_create(), the task is blocked, waiting for port creation. To avoid synchronization delays, connection should be established only after opening the port.

Return value: The function returns the port identification number in the case of successful operation; else -1 is returned.

See also: port_create(), port_delete(), port_disconnect(), port_send(), port_receive().

PORT_DISCONNECT

```
void port_disconnect(PORT p);
```

Description: It closes the connection identified by p.

See also: port_create(), port_connect(), port_delete(), port_send(), port_receive().

PORT_SEND

```
int port_send(PORT p, char *msg, BYTE b);
```

Description: It sends a message pointed by msg to the port identified by p. Message dimension is defined through port_create() or port_connect() and cannot be dynamically changed. The argument b can be BLOCK or NON_BLOCK. If b = BLOCK and the port queue is full, then the task is blocked until the buffer is freed. If b = NON_BLOCK and the port queue is full, then the primitive returns 0 and the message is not sent.

Return value: 1 (TRUE) if the operation can be performed, 0 otherwise.

See also: port_create(), port_connect(), port_disconnect(), port_send(), port_receive().

Example: see the example at page 39.

PORT_RECEIVE

```
int port_receive(PORT p, char *msg, BYTE b);
```

Description: It receives a message from the port identified by p and copies it in a memory buffer pointed by msg. Message dimension is defined through port_create() or port_connect() and cannot be dynamically changed. The argument b can be BLOCK or NON_BLOCK. If b = BLOCK and the port queue is empty, then the task is blocked until a message is available. If b = NON_BLOCK and the port queue is empty, then the primitive returns 0 and no message is received.

Return value: 1 (TRUE) if the operation can be performed, 0 otherwise.

See also: `port_create()`, `port_connect()`, `port_disconnect()`, `port_send()`, `port_receive()`.

Example: see the example at page 39.

4.5 Cyclical Asynchronous Buffers

Cyclical Asynchronous Buffers (or CABs) represent a particular mechanism purposely designed for the cooperation among periodic activities with different activation rates. See [But97] for implementation details.

A CAB provides a one-to-many communication channel, which at any instant contains the most recent message inserted into it. A message is not consumed (that is, extracted) by a receiving process but is maintained into the CAB structure until a new message is overwritten. As a consequence, once the first message is put in a CAB, a task can never be blocked during a receive operation. Similarly, since a new message overwrites the old one, a sender can never be blocked.

Notice that, using such a semantics, a message can be read more than once if the receiver is faster than the sender, while messages can be lost if the sender is faster than the receiver. However, this is not a problem in many control applications, where tasks are interested only in fresh sensory data rather than in the complete message history produced by a sensory acquisition task.

Notice that more tasks can simultaneously access the same buffer in a CAB for reading. Also, if a task P reserves a CAB for writing while another task Q is using that CAB, a new buffer is created, so that P can write its message without interfering with Q. As P finishes writing, its message becomes the most recent one in that CAB. The maximum number of buffers that can be created in a CAB is specified as a parameter in the *cab_create* primitive. To avoid blocking, this number must be equal to the number of tasks that use the CAB plus one.

CABs can be created and initialized by the *cab_create* primitive, which requires the CAB name, the dimension of the message, and the number of messages that the CAB may contain simultaneously. The *cab_delete* primitive removes a CAB from the system and releases the memory space used by its data structures.

To insert a message in a CAB, a task must first reserve a buffer from the CAB memory space, then copy the message into the buffer, and finally put the buffer into the CAB structure, where it becomes the most recent message. This is done according to the following scheme:

```
buf_pointer = cab_reserve(cab_id);
<copy message in *buf_pointer>
cab_putmes(cab_id, buf_pointer);
```

Similarly, to get a message from a CAB, a task has to get the pointer to the most recent message, use the data, and release the pointer. This is done according to the following scheme:

```
mes_pointer = cab_getmes(cab_id);
<use message>
cab_unget(cab_id, mes_pointer);
```

A simple example of CABs' usage is reported below.

```
CAB   cc;

void  main(void)
{
      SYS_PARMS parms=BASE_SYS;

      /* global declaration */
      sys_def_tick(parms,1,mSec);
      sys_init(&parms);
      /* The CAB named cc contains a message of          */
      /* 5 floats and can be used by two tasks           */
      cc = cab_create("my_cab", 5*sizeof(float), 3);
      task_activate(task_create("ll", read, HARD, APERIODIC, 100, NULL));
      task_activate(task_create("ss", write, HARD, PERIODIC, 333, NULL));
      ...
}

/*-----------------------------------*/

TASK  write(void)
{
float msg[5];
char  *pun;
      ...
      while (1) {
            /* send a message to the 'cc' cab             */
            pun = cab_reserve(cc);   /* reserve a buffer   */
            memcpy(pun, msg, 5*sizeof(float));
            cab_putmes(cc, pun);     /* release the buffer */
            task_endcycle();
      }
}

/*-----------------------------------*/

TASK  read(void)
{
float msg[5];
char  *pun;
      ...
      while (1) {
            /* get a message from the 'cc' CAB            */
            pun = cab_getmes(cc);    /* reserve a buffer   */
            memcpy(msg, pun, 5*sizeof(float));
            cab_unget(cc, pun);      /* release the buffer */
            task_endcycle();
      }
}
```

CAB_CREATE

`CAB cab_create(char *name, int dim_mes, BYTE num_mes)`

Description: It initializes a CAB. `name` is a pointer to an identification string (used only for debugging purposes); `dim` is the size of the messages contained in the CAB; `numbuf` is the number of buffers the CAB is composed of. Notice that such a number must be greater than or equal to the number of tasks that use the CAB plus one.

Return Value: It returns the index of the created CAB.

See also:: `cab_delete()`, `cab_reserve()`, `cab_putmes()`, `cab_getmes()`, `cab_unget()`.

CAB_DELETE

`void cab_delete(CAB cc);`

Description: It removes the `cc` CAB from the system, deallocating its buffers and data structures.

See also: `cab_create()`, `cab_reserve()`, `cab_putmes()`, `cab_getmes()`, `cab_unget()`.

CAB_RESERVE

`char *cab_reserve(CAB cc);`

Description: it reserves a buffer belonging to the `cc` CAB and returns a pointer to it. The primitive has to be used only by writers and *never* by readers.

Return value: it returns a pointer to the reserved buffer.

See also: `cab_delete()`, `cab_create()`, `cab_putmes()`, `cab_getmes()`, `cab_unget()`.

CAB_PUTMES

`void cab_putmes(CAB id, char *pun)`

Description: It inserts the message pointed by `pun` into the CAB identified by `id`. This primitive must be used *only* by writing tasks.

See also: `cab_delete()`, `cab_create()`, `cab_reserve()`, `cab_getmes()`, `cab_unget()`.

CAB_GETMES

`char *cab_getmes(CAB cc);`

Description: It returns a pointer to the latest message written into the `cc` CAB. This primitive must be used *only* by reading tasks.

Returned value: It returns a pointer to the most recent message contained in the CAB.

See also: `cab_delete()`, `cab_create()`, `cab_putmes()`, `cab_reserve()`, `cab_unget()`.

CAB_UNGET

`void cab_unget(CAB cc, char *pun);`

Description: it notifies the system that the buffer pointed by `pun` belonging to the `cc` CAB is no longer used by the calling task.

See also: `cab_delete()`, `cab_create()`, `cab_reserve()`, `cab_getmes()`, `cab_putmes()`.

4.6 POSIX Message Queues

S.Ha.R.K. provides the message passing function defined in the POSIX standard. For more information, see Section 15 of the POSIX standard, Message Passing.

Chapter 5

Utility functions

S.Ha.R.K. provides a set of utility functions aimed at getting information about the kernel state. Mainly, they allow a user to get the actual system time and some information concerning the tasks' state. Moreover, it allows to set exception handlers and to manage interrupts.

5.1 Reading time

The S.Ha.R.K. Kernel does not have the concept of tick. Every time interval and every absolute time in the system is measured usin the Real-Time Clock available on the PC. To read the current time you can use the following function:

SYS_GETTIME

```
TIME sys_gettime(struct timespec *t);
```

Description: It returns the number of microseconds elapsed from system's initialization, that is from the end of the `__kernel_register_levels__` function. If the `t` value is not equal `NULL`, the function fills also the timespec structure passed as parameter.

5.2 Getting information on tasks

Since all the tasks are handled by a Module, it is a responsibility of each Module to hide or not hide informations about the tasks handled by the system. However, at the moment S.Ha.R.K. provides a function that simply prints the tasks state on the console[1].

[1] Old versions of the kernel supported a `void sys_status(DWORD cw);` primitive. That primitive is currently unsupported.

PERROR

```
void perror (const char *s);
```

Description: This is the POSIX `perror()` funcion, that prints on the console (using `kern_printf`) a message that explain the meaning of the `errno` variable. Note that each task has its own `errno` variable, as specified by the `POSIX` standard.

exec_shadow

```
PID exec_shadow;
```

Description: This is the internal variable used by the Kernel to track the running task. You can read its value to know the PID of the current task. You CAN NOT modify this variable.

5.3 Printing messages on the console

To print a simple message on the console, please use the c* functions (cprintf, cputs, ...) described in Volume II. If tou are debugging *the kernel*, you can use kern_printf to print very simple messages without floating point arithmetic.

Chapter 6

Signals and Exception Handling

6.1 Signals

S.Ha.R.K. implements the specification of the signals and of the real-time signald provided by the standard IEEE 1003.13 POSIX PSE51/PSE52. In particular, you can use all the functions described into the IEEE 1003.1{a,b} standards, except that:

- all the `pid_t` parameters and in general all parameters related with processes should be ignored;
- when in POSIX a signal cause the termination of the process, it causes in S.Ha.R.K. the termination of the whole system (you can think S.Ha.R.K. as a single process multithread kernel);
- The `siginfo_t` structure contains an additional parameter called `si_task` of type `PID`. It contains the `PID` of the task that queued a particular real-time signal.

In particular, you can use these functions for signal handling: `kill`, `sigemptyset`, `sigfillset`, `sigaddset`, `sigdelset`, `sigismember`, `sigaction`, `pthread_sigmask`[1], `sigprocmask`, `sigpending`, `sigsuspend`, `sigwait`, `sigwaitinfo`, `sigtimedwait`, `sigqueue`, `pthread_kill`[2], `alarm`, `pause`, `sleep` (note the difference between `sleep`, `task_sleep` and `nanosleep`!), `raise`, `signal`.

6.2 Exception handling

S.Ha.R.K. provides a flexible mechanism to handle the exceptions of the Kernel. The mechanism is based on the POSIX signals. In fact, S.Ha.R.K. exceptions are remapped on the real-time signal `SIGHEXC`[3] (9). Every time something goes wrong, the system calls the primitive `kern_raise`, that simply queue a real-time signal of number SIGHEXC.

The user can define its own exception handler simply remapping the SIGHEXC signal using the POSIX primitive `sigaction`. To fullfil the typical usage of an exception handler (exit the

[1] If you are not using the POSIX scheduling modules please use `task_sigmask`

[2] If you are not using the POSIX scheduling modules please use `task_signal` (Note that `task_kill` does not send any signal, but issue a cancellation request on a task!)

[3] see `include/signal.h` .

system after printing a message), the default behavior of the signal handler has been redefined to print a text message on system shutdown.[4]

Here is a sample code that explain how to redefine a signal handler:

```
#include <kernel/kern.h>

void thehandler(int signo, siginfo_t *info, void *extra) {
  /* the signal handler:
  info.sivalue.sival_int contains the exception number
  (see include/bits/errno.h)
  info.si_task is the task that raised the exception
  extra is not used */
  ...
}
...
int myfunc(...)  {
  struct sigaction action;
  ...
  action.sa_flags = SA_SIGINFO;
  action.sa_sigaction = thehandler;
  action.sa_handler = 0;
  sigfillset(&action.sa_mask);
  sigaction(SIGHEXC, &action, NULL);
  ...
}
```

KERN_RAISE

```
void kern_raise(int n, PID p);
```

Description: This function uses the POSIX function sigqueue to put a signal SIGHEXC into the signal queue. The parameter n is used as the exception number, and it is passed into the siginfo_t parameter (into the sivalue.sival_int field). The signal appears to be queued by the task p (the p value is stored into the si_task field of the siginfo_t structure passed as parameter).

[4]Older versions of the Kernel supported two functions to be used for standard redefinition of the kernel signal handler. These functions, called SET_EXCHANDLER_TXT and SET_EXCHANDLER_GRX, are no more supported, and can be removed from your code without problems.

Chapter 7

Interrupt and HW Ports handling

Generally speaking, I/O to and from an external peripheral device can be handled in three different ways depending on the peripheral type and on the application:

- **Polling**: the program cyclically checks the status of the I/O port, waiting for a input data to be ready or an output data to be transmittable;
- **Interrupt**: the program enables the I/O interface to send a hardware interrupt every time an input data is available or an output data is transmittable;
- **DMA**: the program enables the interface to use DMA mechaninsm for directly transferring data to/from memory.

In this chapter we will analyse the support that the S.Ha.R.K. kernel provides for using the second method (interrupt).

When an interrupt arrives, a code for the hand-shake with the interface and for transferring data has to be executed. This code can run in two different modes:

- it can be entirely encapsulated in a function to be executed immediately on the interrupt arrival, in the context of the executing task (*fast handler*);
- it can be entirely encapsulated in a task (*safe handler*) which is activated on the interrupt arrival and scheduled with its own priority together with the other tasks.

The first method is appropriate when the interrupt needs a fast response time. Its potential drawback is that if its computation time is not low, the overall schedulabuility can be severly affected. This is because the guarantee algorithm does not take into account the execution time of the interrupt handlers. The second method, on the contrary, is perfectly integrated with the kernel's scheduling mechanism, but can cause considerable delays in transferring data.

S.Ha.R.K. provides great flexibility in interrupt handling, since it allows each interrupt to be associated with a *fast handler*, a `safe handler`, or both.

On an interrupt's arrival the following operations are performed by the kernel:

- The system checks whether a fast handler is associated with the interrupt. If so, the interrupts are enabled and the handler is invoked. This method allows a handler to be interrupted by a higher priority handler. As an example, the keyboard handler (interrupt 1) can be interrupted be the timer handler (interrupt 0).

- The system checks whether a sporadic task (*safe handler*) is associated with the interrupt. If so, the task is activated and is eligible to run with enabled interrupts.

The system provides a set of functions for accessing the hardaware interfaces' ports. In the drivers directory you can find examples of a S.Ha.R.K. device driver.

7.1 Setting an interrupt handler

HANDLER_SET

```
int handler_set(int no, void (*fast)(int), PID pi, BYTE lock);
```

Description: It installs function `fast` (fast handler) and the sporadic task `p` (safe handler) on the interrupt identified by `no`. The `no` parameter must belong to the range 1...15 (interrupt 0 is associated to the timer and cannot be intercepted). On the interrupt's arrival, function `fast` is invoked and runs. Depending on the `lock` flag, the interrupts are disabled (`lock` = TRUE) or enabled (`lock` = FALSE) during handler execution. Furthermore, on the interrupt's arrival, task `p` is activated.

HANDLER_REMOVE

```
void handler_remove(int no);
```

Description: It removes the handler of the interrupt number `intno`; the interrupt is masked.

7.2 Reading and writing from I/O ports

INP, INPW, INPD

```
unsigned char inp(unsigned short _port);

unsigned short inpw (unsigned short _port);

unsigned long inpd(unsigned short _port);
```

Description: They return the data read on port `_port`.

OUTP, OUTPW, OUTPD

```
void outp(unsigned short _port, unsigned char _data);

void outpw(unsigned short _port, unsigned short _data);

void outpd(unsigned short _port, unsigned long _data)
```

Description: It writes the data `_data` into the port `_port`.

7.3 Disabling/Enabling interrupts

KERN_CLI

```
void kern_cli(void);
```

Description: It disables interrupts (as the x86 `cli` instruction).

KERN_STI

```
void kern_sti(void);
```

Description: It enables interrupts (as the x86 `sti` instruction).

7.4 Saving/Restoring interrupts

KERN_FSAVE

```
SYS_FLAGS kern_fsave(void);
```

Description: It disables interrupts (as the x86 `cli` instruction). The CPU flags are returned by the function; in that way they can be restored using kern_frestore

KERN_FRESTORE

```
void kern_frestore(SYS_FLAGS f);
```

Description: It restores the interrupt state as it was when the correspondent `kern_fsave` was called.

7.5 Masking/Unmasking PIC interrupts

IRQ_MASK

```
void irq_mask(WORD irqno);
```

Description: It mask the interrupt number `irqno` on the PC PIC. `irqno` must be in the interval [1..15].

IRQ_UNMASK

```
void irq_unmask(WORD irqno);
```

Description: It unmask the interrupt number `irqno` on the PC PIC. `irqno` must be in the interval [1..15].

Chapter 8

Memory Management Functions

The S.Ha.R.K. Kernel provides the standard set of memory allocations functions provided by the Standard C libraries. In particular, the functions listed in figure 8.1 can be used.

In particular[1]:

- calloc() allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero. The value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable, or NULL if the request fails.
- malloc() allocates size bytes and returns a pointer to the allocated memory. The memory is not cleared. The value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable, or NULL if the request fails.
- free() frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc() or realloc(). Otherwise, or if free(ptr) has already been called before, undefined behaviour occurs. If ptr is NULL, no operation is performed.
- realloc() changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If ptr is NULL, the call is equivalent to malloc(size); if size is equal to zero, the call is equivalent to free(ptr). Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc(). It returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from ptr,

[1] These descriptions came directly from the Linux man pages...

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

Figure 8.1: Memory allocation functions.

or NULL if the request fails or if size was equal to 0. If realloc() fails the original block is left untouched - it is not freed or moved.

The S.Ha.R.K. Kernel also provides a set of low-level memory management functions that can be used to allocate memory with particular requirements (for example, they are useful for getting memory blocks aligned to a page (4 Kb) boundary or with addresses under 1/16 Mb). Description of these functions is given in Chapter 3 of the S.Ha.R.K. Architecture Manual.

Appendix A

Errors and Exceptions

This appendix describes the errors and exceptions codes that can be printed into the screen, returned by a functions into the the `errno` variables or that the kernel can raise. These error constants are included from the `errno.h` standard include file, and are contained into the `bits/errno.h` include file.

A.1 Abort codes

Name	N.	Description
none	1	Generic OSLib abortASIG_DEFAULT_ACTION
ASIG_DEFAULT_ACTION	2	The default handler of a signal has been executed
ASIGINIT	3	Internal error in initializing signals (should never happens)
AHEXC	4	a set_exchandler_XXX function has been executed.
AARPFULL	5	ARP table full.

A.2 Exceptions posted with kern_raise

Name	N.	Description
XDOUBLE_EXCEPTION	1	Two exceptions has been raised. Currently not used
XUNVALID_KILL_SHADOW	2	Called into the internal function task_makefree because a task was killed while some other task shadow points to that task.
XNOMORE_CLEANUPS	3	Too many cleanups handlers has been used. Currently not used.
XUNVALID_TASK	4	The Registered Modules does not implement a primitive called by the task (usually happens when the user calls task_delay or task_sleep)
XUNVALID_GUEST	5	The Registered Modules does not handle correctly the guest tasks. Check the initfile.

XNOMORE_EVENTS	6	Too many OSLib events posted. The number of OSLib events posted is declared in `include/ll/sys/ll/event.h` into the constant `MAX_EVENT`.
XDEADLINE_MISS	7	A Task missed its deadline.
XWCET_VIOLATION	8	A Task consumed more tha its declared WCET.
XACTIVATION	9	A Sporadic task has been activated more frequently than declared.
XMUTEX_OWNER_KILLED	10	A task is terminated while it owns a mutex.
XSRP_UNVALID_LOCK	11	A task tried to lock a SRP mutex with a wrong preemption level, or a task tries to lock a SRP mutex already locked, or a task tries to lock a SRP mutex without declaring its preemption level.
XUNVALID_DUMMY_OP	12	Someone tried to execute an operation on the dummy Scheduling Module.
XUNVALID_SS_REPLENISH	13	Error in the Sporadic Server replenishments. Please look at `kernel/modules/ss.c`.
XARP_TABLE_FULL	14	Arp table full. See `drivers/net/arp.c`.
XNETBUFF_INIT_EXC	15	Network buffers error. See `drivers/net/netbuff.*`.
XNETBUFF_GET_EXC	16	Network buffers error. See `drivers/net/netbuff.*`.
XNETBUFF_ALREADYFREE_EXC	17	Network buffers error. See `drivers/net/netbuff.*`.
XNETBUFF_RELEASE_EXC	18	Network buffers error. See `drivers/net/netbuff.*`.
XUDP_BADCHK_EXC	19	UDP CRC check failed.

A.3 POSIX error codes

The POSIX error codes have numbers form 1 to 125 and are listed into `include/errno.h`.

A.4 S.Ha.R.K. error codes

Name	N.	Description
EWRONG_INT_NO	126	Wrong int number passed to handler-set or handler_remove.
EUSED_INT_NO	127	Already used int number.
EUNUSED_INT_NO	128	Int number not used.
ETOOMUCH_INITFUNC	129	Too much init functions posted. (Currently not used)
ETOOMUCH_EXITFUNC	130	Too much exit functions posted.
ENO_AVAIL_TASK	131	Task limit reached. Up to TSSMax-1 tasks can be created. See include/ll/i386/tss-ctx.h and include/kernel/const.h
ENO_AVAIL_SCHEDLEVEL	132	The Task Model passed with task_create cannot be accepted by any scheduling module.

ETASK_CREATE	133	Error during task_create.
ENO_AVAIL_RESLEVEL	134	A Resource Model passed with task_create cannot be accepted by any resource module.
ENO_GUARANTEE	135	The new task cannot be accepted by the Scheduling Modules
ENO_AVAIL_STACK_MEM	136	No space left to allocate the task stack.
ENO_AVAIL_TSS	137	No TSS free. This error should never happen.
EUNVALID_KILL	138	The PID you tried to kill is not a task or has the NO_KILL flag set.
EUNVALID_TASK_ID	139	The PID passed to task_activate is not correct.
EUNVALID_GROUP	140	Group 0 is not a valid group.
EPORT_NO_MORE_DESCR	141	HARTPORT: No more port descriptors available.
EPORT_NO_MORE_INTERF	142	HARTPORT: No more free port interfaces.
EPORT_INCOMPAT_MESSAGE	143	HARTPORT: Incompatible message (Write on a read port or viceversa)
EPORT_ALREADY_OPEN	144	HARTPORT: The port is already open.
EPORT_NO_MORE_HASHENTRY	145	HARTPORT: No more Hash entries to create a port.
EPORT_2_CONNECT	146	HARTPORT: Error creating the port.
EPORT_UNSUPPORTED_ACC	147	HARTPORT: Error in port_connect.
EPORT_WRONG_OP	148	HARTPORT: Wrong operation.
EPORT_WRONG_TYPE	149	HARTPORT: Operation not supported by the port type.
EPORT_UNVALID_DESCR	150	HARTPORT: Invalid port descriptor.
ECAB_UNVALID_ID	151	CABS: Invalid CAB ID.
ECAB_CLOSED	152	CABS: CAB Closed.
ECAB_UNVALID_MSG_NUM	153	CABS: Invalid Message number.
ECAB_NO_MORE_ENTRY	154	CABS: No more entries.
ECAB_TOO_MUCH_MSG	155	CABS: Too much messages.

Index

_exit, 13

BLOCKED, 16

cab_create(), 42
cab_delete(), 43
cab_getmes(), 44
cab_putmes(), 44
cab_reserve(), 43
cab_unget(), 44
cond_broadcast(), 36
cond_destroy(), 36
cond_init(), 36
cond_signal(), 36
cond_timedwait(), 37
cond_wait(), 37
cprintf, 46
cputs, 46

errno, 46
EXE, 16
exec_shadow, 46
exit, 13

group_activate(), 20
group_kill(), 20

handler_remove(), 50
handler_set(), 50
HARD, 16

IDLE, 16
inp(), 50
inpd(), 50
inpw(), 50
internal_sem_getvalue(), 34
internal_sem_init(), 33
internal_sem_post(), 33
internal_sem_trywait(), 33
internal_sem_wait(), 33
irq_mask(), 51
irq_unmask(), 51

jet_delstat(), 27
jet_getstat(), 27
jet_gettable(), 27

kern_cli(), 51
kern_frestore(), 51
kern_fsave(), 51
kern_printf, 46
kern_raise(), 48
kern_sti(), 51

local task context, 16

Make, 6
make, 7
makefile, 7
MODEL, 17
mutex_destroy(), 34
mutex_init(), 34
mutex_lock(), 35
mutex_trylock(), 36
mutex_unlock(), 36

NRT, 16

outp(), 50
outpd(), 50
outpw(), 50

perror(), 46
port_connect(), 40
port_create(), 38
port_delete(), 39
port_disconnect(), 40
port_receive(), 40
port_send(), 40

READY, 16

sem_destroy(), 29

sem_getvalue(), 32
sem_init(), 29
sem_post(), 32
sem_trywait(), 31
sem_wait(), 31
sem_xpost(), 32
set_exchandler_grx(), 48
set_exchandler_txt(), 48
SLEEP, 16
SOFT, 16
sys__shutdown_message, 14
sys_abort_shutdown, 14
sys_atexit(), 10
SYS_ATRUNLEVEL, 13
sys_end, 14
SYS_FLAGS, 51
sys_gettime(), 45
sys_set_reboot, 14
sys_status(), 45
system initialization, 10

task creation, 17
task_abort(), 20
task_activate(), 19
task_block_activation(), 20
task_cleanup_pop(), 23
task_cleanup_push(), 23
task_create(), 17
task_createn(), 17
task_delay(), 25
task_endcycle(), 15, 21
task_getspecific(), 23
task_join(), 24
task_joinable(), 24
task_key_create(), 21
task_key_delete(), 23
task_kill(), 19
task_nopreempt(), 25
task_preempt(), 25
task_setcancelstate(), 24
task_setcanceltype(), 24
task_setspecific(), 23
task_sleep(), 15, 21
task_testcancel(), 23
task_unblock_activation(), 20
task_unjoinable(), 25

Bibliography

[But97] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers, Boston, 1997.