E.R.I.K.A.

Embedded Real-tIme Kernel Architecture

ERIKA Educational User Manual 1.3

(July 12, 2004)

Paolo GAI (pj@sssup.it)

Davide CANTINI (cantini@sssup.it)

Michele CIRINEI (cirinei@sssup.it)

Antonio MANCINA (mancina@sssup.it)

Alessandro COLANTONIO (alessandro@gandalf.sssup.it)

RETIS Lab.
c/o Polo S. Anna Valdera
Via Rinaldo Piaggio
56025 Pontedera – Italy

This manual has been made using

LaTeX and LyX.

Introduction

THIS BOOK can be viewed as a reference manual that contains all the informations about the ERIKA Educational Kernel. This document is composed by two main parts:

- *Kernel Description*, that covers installations, configuration, and usage of the Kernel;
- *Kernel Internals*, that describes in detail all the implementation choices made in the design of the Kernel.

The "Kernel Description" part has the following organization:

Chapter1 Provides an overview of ERIKA Educational .

Chapter2 Explains how to install ERIKA Educational and how you can set up the development environment.

Chapter3 Describes all the functionality provided by ERIKA Educational .

Chapter4 Details the interface provided to the application.

Chapter5 Lists all the Kernel Layer currently provided, and shortly describes their peculiarities.

Chapter6 Describes the application file organization, giving some guide line to write a typical ERIKA Educational application.

Chapter7 How to use I/O drivers for each supported hardware platform.

Chapter8 Explains some additional high level libraries, that are used for communication.

The "Kernel Internals" part has the following organization:

Chapter9 Details the internal construction of HAL layers, showing common characteristics among different architectures and detailing differences.

Chapter10 Shows the internal structure of a Kernel layer.

Chapter11 Shows the H8 HAL Internals.

There is also a set of appendices:

AppendixA Summarizes the ERIKA Educational API

AppendixB Shows the ERIKA Educational file organization and their contents.

AppendixC Briefly describes some algorithm used from a theoretical point of view.

AppendixD The license used for the ERIKA Educational source code.

AppendixE The license used to make this book.

Book conventions

Italics Emphasizes important words.

`Monospaced` Denotes generic code in any programming language.

`Mono bold` Denotes keywords in any programming language.

 This symbol draw attention to situations in which the programmer should take care.

License and where to send comments

Please keep in mind that the ERIKA Educational project is an Open-Source Project distributed under the terms of the GPL License. In case you need a different kind of license (for example in case you want to use our code or parts of it into a proprietary commercial product) please contact the authors again for an agreement on the terms of use of the software. In any case every future development of the project will be released also under the GPL license.

To help us provide you a better product, email and suggestion are very welcome! Please send email to our help mailing list (`erika-help@erika.sssup.it`). You can find any other useful information at the official ERIKA Educational web page (`http://erika.sssup.it`).

Contents

I

Kernel description

1 What is ERIKA?

ERIKA (Embedded Real tIme Kernel Architecture) is a research project about micro-kernel architectures for embedded systems. ERIKA Educational was born to support all the architectures used for power-train microcontrollers in the automotive industry. Therefore the targets of ERIKA Educational are all systems where predictability and small kernel footprint (especially in RAM) are the major requirements.

E.R.I.K.A. is currently distributed in two different releases:

- ERIKA Educational

  It is an Educational version (distributed whithin the GPL license), that includes Lego Mindstorms support and a minimal working kernel. This distribution contains a set of basic functions that enables the users to create small applications under Lego Mindstorms (and on other architectures that will be avaible in the furure).

  In particular, it has been particularly designed for the Educational Environment (such as schools and universities), as a good platform to learn the Real-Time scheduling basics. The Educational version is distributed under the GPL license, and we guarantee that all the third-party contributions will be distributed under the same license (unless an agreement with them has been reached).

- ERIKA Enterprise

  It is a commercial product separated from ERIKA Educational, that includes other architectures and kernel layers. Please note that this manual only contains the ERIKA Educational informations.

The architecture of the ERIKA Educational Kernels consists of two main layers: the Kernel Layer and the Hardware Abstraction Layer.

The *Kernel Layer* contains a set of modules that implement task management and real-time scheduling policies. At the moment, the available policies are: Fixed priority with preemption threshold, and EDF with preemption threshold kernel layer. Both use SRP, that is a protocol that allows to share resources between threads and to share the system stack among all the threads while preserving time predictability.

The *Hardware Abstraction Layer* contains the Hardware dependent code that manages context switches an interrupt handling. A high degree of modularity makes the Kernels suitable for various microprocessors: currently ERIKA Educational supports Hitachi H8/Lego Mindstorms, more architectures are available on the Enterprise edition. Because most of the ERIKA Educational code is written in C, it is possible to port ERIKA Educational to a new processor family in a very short time. The interface is identical for all implementations of the operating system, in order to allow portability and re-usability of application software.

In order to obtain the fastest context switch and system call performance, the ERIKA Educational code is "plugged" into the application code. ERIKA Educational is also designed to be *scalable*. The ERIKA Educational services are implemented as a C library, thus only those services actually used by the application are included into the run-time image. The high modularity and configurability of ERIKA Educational allows to optimize the code in order to fulfill the application requirements, reducing the overall memory footprint.

2 Installing ERIKA Educational

THIS CHAPTER briefly describes where the latest source of the ERIKA Educational project can be found, and what is needed for using it. In particular, it describes various issues about installation, setup, and usage of ERIKA Educational code.

2.1 Setting up the Software (for Unix/Linux platforms)

The ERIKA Educational Snapshots need the GNU Makefile to compile in the correct way the source code of the project. Download the last snapshot of the Project, and decompress it on your Hard Disk. Finally, you need to install the Development Environment for the H8/300 platform.

2.2 Building the H8 development environment

First of all you have to get the GCC (Gnu C Compiler) which is generally distribuited as two separate packages: gcc (which is the C/C++ compiler) and binutils (which contains assembler, linker and some other tools). These packages are distribuited from the GNU project ftp server (ftp.gnu.org). To obtain the versions we used to test and develop ERIKA Educational you should download the links below:

- `ftp://ftp.gnu.org/pub/gnu/binutils/binutils-2.11.2.tar.gz`

- `ftp://ftp.gnu.org/pub/gnu/gcc/gcc-core-2.95.2.tar.gz`

Once you get this packages you have to extract them using the tar command:

```
tar xfz binutils-2.10.tar.gz
tar xfz gcc-core-2.95.2.tar.gz
```

This will create two subdirectories: binutils-2.10 and gcc-2.95.

When you have extracted the files, the next thing to do is to compile binutils. This must be done before compiling gcc, since this process depends on binutils being available for the Hitachi architecture, H8/300. Both binutils and gcc needs to be configured for the particular setup; from a Linux system you can use the shell script configure in the way showed below:

```
cd binutils-2.10
./configure --target=h8300-hitachi-hms
  --prefix=$HOME/hitachi
```

Since we want gcc to work as a cross-compiler, we also need the assembler and the linker to generate code for Hitachi H8/300. This is what the `--target=h8300-hitachi-hms` option specifies. The option `--prefix=$HOME/hitachi` specifies that binutils is to be installed relative to the directory `$HOME/hitachi`. This means that executable files are installed in `$HOME/hitachi/bin`, libraries in `$HOME/hitachi/lib` and documentation and other files in other subdirectories of `$HOME/hitachi`. If you want the tools installed somewhere else, say `/usr/local`, just specify `--prefix=/usr/local`. When configure finishes checking the system setup and provided no problems were detected, binutils is now ready to be compiled. This is done using make:

```
make
```

When the compilation is finished, binutils can be installed, which is accomplished with the make command again:

```
make install
```

The installation process creates a number of subdirectories of `$HOME/hitachi`, and copies the newly built executable files, libraries, documentation and other files to these.

Having installed binutils, gcc can be compiled in a way such this:

```
cd gcc-2.95.2
./configure --target=h8300-hitachi-hms
  --prefix=$HOME/hitachi --with-newlib
```

For the same reasons as above, target and prefix is specified as shown. The option `--with-newlib` specifies that gcc should not expect to find a standard C library for the Hitachi H8/300 architecture.

When configure finishes, gcc can be compiled. However, during the compilation, make must be able to access the executable files from binutils, so `$HOME/hitachi/bin` should be added to the path. One way to do this is to invoke make as:

```
env PATH=$HOME/hitachi/bin:$PATH
make
```

When this is done, gcc can be installed:

```
make install
```

As for binutils, this will install executable files and other files in subdirectories of `$HOME/hitachi`.

2.3 Compiling ERIKA Educational

The Snapshots contains a set of tests that can be found under the tests directory under the ERIKA Educational tree.

Suppose that:

- H8 Development environment is installed under `/usr/share/h8` (this means that above you specified `$HOME=/usr/share/h8`)
- ERIKA Educational is installed under `/home/user/erika`

Set these environment variables:

```
export H8BASE=/usr/share/h8
export ERIKABASE=/home/user/erika
```

Then, go to a test directory and run

```
make
```

The Makefile will create another directory called out, where all object files will be put.

If you have any problem, please send a mail to `erika-help@gandalf.sssup.it`.

3 Architecture of ERIKA Educational

This chapter shows the architecture of ERIKA Educational from a functional point of view, without describing the implementation details. In particular, the main information introduced here are:

- thread and interrupt handling, and thread synchronization;
- the two main layers of the project (Hardware Abstraction Layer and Kernel Layer);
- the modular architecture of ERIKA Educational .

3.1 The ERIKA Educational RTOS

The *core* of a Real Time Operating System (RTOS) is represented by a *kernel*, that is a software layer that directly interacts with the hardware on which the application runs. The main purpose of a Kernel is to give the application an interface (composed by constants, types and primitives) capable to handle a set of entities that tries to abstract from the peculiarities of the underlying hardware.

The ERIKA Educational Project provides a Kernel that is capable to:

- Handle *threads* (a thread is a user function that is executed concurrently with other entities in the system).

- Handle *synchronization between threads* (threads can communicate each other sharing informations and events).
- Handle *interrupts* (an interrupt is an hardware mechanism that allow to asynchronously interrupt a running thread).

The following sections will explain in detail the previous concepts.

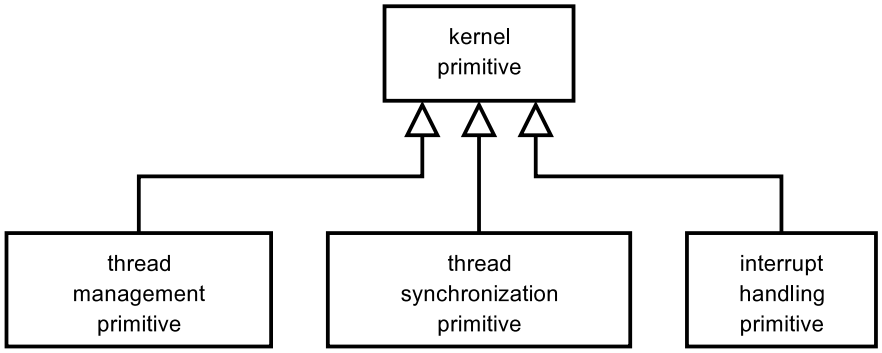


Figure 3.1 Kernel primitives

3.2 Thread Handling

Every operating system that supports a multiprogramming environment provides some kind of entities that can execute concurrently. These entities are sometimes called threads, processes and tasks.

- The term *process* is usually used for identifying an address space with one or more threads executing within that address space, and the required system resources for those threads.
- The term *thread* can be thought as a single flow of control within a process.
- The term *task* is used in the Real-Time literature with different meanings. Usually it simply represents a flow of control, without any link to eventual address spaces that can be shared or not by the processes. Hence, depending on the particular paper, it is used as synonym of process or thread.

ERIKA Educational does not support any kind of memory protection, so we can say that the ERIKA Educational Kernel can handle a set of *threads*. This can be simply explained because the project is targeted to small microcontroller systems that often do not implement any kind of MMU that can enforce memory protection.

The ERIKA Educational Kernel provides only the basic features for thread handling: it implements a real-time scheduling algorithm, a thread activation service and all the underlying context switch functions. In particular, the scheduling algorithm are derived from the Rate Monotonic (RM) Algorithm [4], the Earliest Deadline First (EDF) Algorithm [4], the Stack Resource Policy (SRP) [1], and the Preemption Thresholds [6].

Each thread owns a set of attributes (like body function, priority, stack pointer, private registers, and so on) that identifies the current state of a thread. Moreover, real-time operating systems attach to each thread a status word that identifies the behavior of the task with respect to the scheduling algorithm. In particular during this manual we will consider the following task states:

- *Stacked* state. The thread τ started its execution, and its frame is currently allocated on the its stack. That means that τ is the *running* thread (and so its frame is on the top of its stack), or that τ has been *preempted* by another thread whose stack has been put over τ's frame.
- *Ready* state. The task has been activated at it is waiting its turn to get the CPU. All the tasks in this state are queued in a data structure called *ready queue*.
- *Idle* state. The last instance of the thread τ has terminated and τ is not currently queued into the ready queue. Moreover, τ does not allocate any byte on its stack. Usually a task in idle state is waiting for an activation.
- *Blocked* state. This state is used when a task is blocked on a synchronization point. Note that this state is meaningless when using SRP and the mono stack HALs, since a thread can not block with these models.

Note One of the advantage of multi-threading is that the same C function can be called from multiple threads. This reduce code space, but it requires that C functions are *reentrant* (i.e. the caller's return address does not rely on global C variables)

3.3 Thread synchronization

The ERIKA Educational Kernels support some synchronization mechanisms that can be used to guarantee that some critical code sections are executed in mutual exclusion, and other synchronization mechanisms that can be used for thread communications and rendez-vous.

Currently, the kernel gives a basic support for single unit resources under the SRP protocol. That protocol allow to implement a mutual exclusion between threads that shares the same stack. In particular, the system provides a type `ERIKA_MUTEX` and two mutex primitives, `ERIKA_mutex_lock()` and `ERIKA_mutex_unlock()`, that can be used to set the "borders" of the critical sections.

Moreover, the kernel also supports the preemption threshold technology in a transparent way. There are also blocking primitive implemented, such as classic semaphores and CABs. Anyway, synchronization points can be directly implemented using explicit thread activations.

3.4 Interrupt Handling

The ERIKA Educational Kernel can be configured to directly interact with the underlying hardware architecture. In particular, it can handle the interrupts that are raised by the I/O interfaces, timers, and so on in a clean way. In fact, it provides an interface that allows to link an handler written by the user into an interrupt vector. That interface also provides a proper handling of eventual interrupt controllers that may be present in a particular architecture.

The handler can be considered a special function that can call a subset of the primitives. These primitives can be used to interact with the operating system, for example activating a thread that will terminate the interaction with the hardware interface. In that way, urgent pieces of code can be put directly into the handler, whereas less urgent things can be done in a separated thread without interferences with the high priority tasks.

3.5 Layers

The ERIKA Educational kernel is composed by two main layers, the Kernel Layer and the Hardware Abstraction Layer (HAL):

Kernel Layer This layer is the software layer that exports all the system primitives to the application. It is written using the C language in a way independent from the underlying architecture. This layer uses the services offered by the HAL for thread and HW interface handling.

Hardware Abstraction Layer (HAL) It is the software layer used by the Kernel Layer to abstract from a particular architecture. This is the only non-portable part of the system, and it isolates all the peculiarities of the underlying architecture from the implementation of the behavior of the primitives. All the primitives of the HAL are related to low level aspects like context switch and interrupt handling. All the other levels use the service exported by the HAL to implement their services. A porting of the whole ERIKA Educational Kernel to another architecture requires only the modification of the HAL.

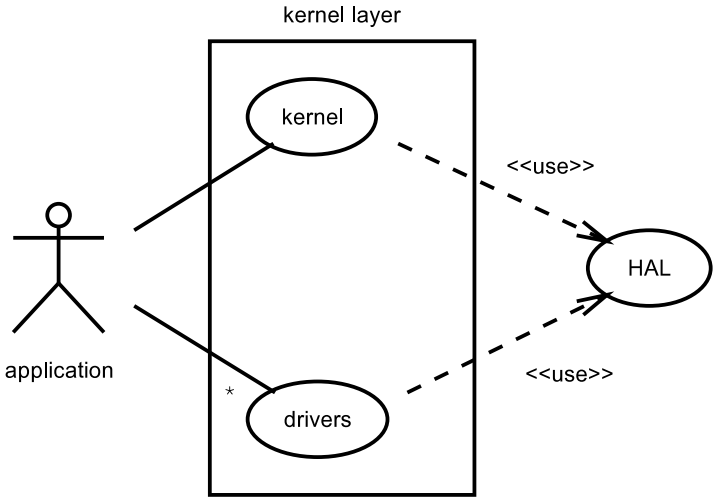


Figure 3.2 Logical division of kernel primitives: the application point of view.

3.5.1 The Kernel Layer

The function provided by the Kernel can be grouped in:

Queue Handling The Kernel use some queue data structures to keep track of the state of all the threads of an application. The functions for queue management cannot be called directly by the application, but

only inside the Kernel. The queues are usually ordered. The order of these queues may depend on the scheduling algorithm chosen.

Scheduling The kernel uses some extra functions to schedule the threads in the system. These functions use the HAL interface and the queue handling functions. These functions are not directly accessible to the user.

User interface This is the part of the kernel that exports the constants, types, data and primitives the user can use. The user interface, in particular, cover all the system services that are handled by the Kernel.

The kernel layer internally uses some extra informations about the tasks. In particular, it defines for each thread the status, the priorities, and the pending activations. Moreover it defines the mechanisms for mutual exclusion and the available resources.

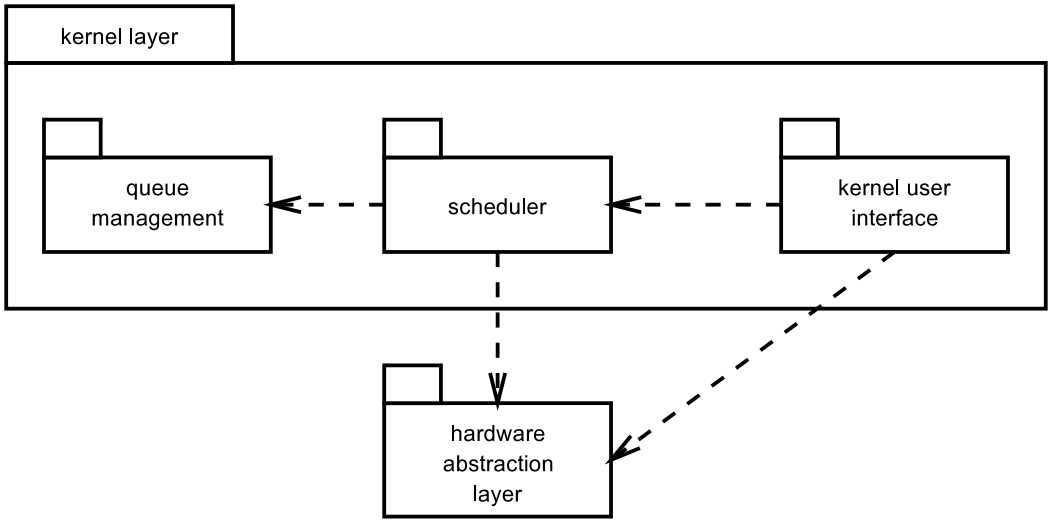


Figure 3.3 Main modules of the Kernel Layer

3.5.2 The Hardware Abstraction Layer

The main objective of the HAL Layer is to export to the Kernel Layer the abstraction of thread. In general, each thread is characterized by:

- A *body,* that is the C function that implement the control flow of a thread. That function is called by the HAL with interrupts enabled each time a particular thread become the running thread.

- A *context,* that is the CPU status of a thread at a particular instant of time. The real composition of the context varies from architecture to architectures. The context can be thought as a copy of the CPU registers that is made each time the running thread is preempted or interrupted for some reasons. In any case the Application can not rely on the composition of a context: in general a particular HAL can save only a subset of the CPU registers, that is, the minimum set of registers that can reproduce without problem the status of a thread.

In practice, the HAL offers a support for thread execution. Using the HAL interface, threads can be executed, stopped, and their context saved. The Kernel Layer internally only refers to threads using some identifiers called *Thread IDentifier* (`ERIKA_TID`). This allow the implementation of various scheduling algorithms in a way independent on the particular hardware architecture.

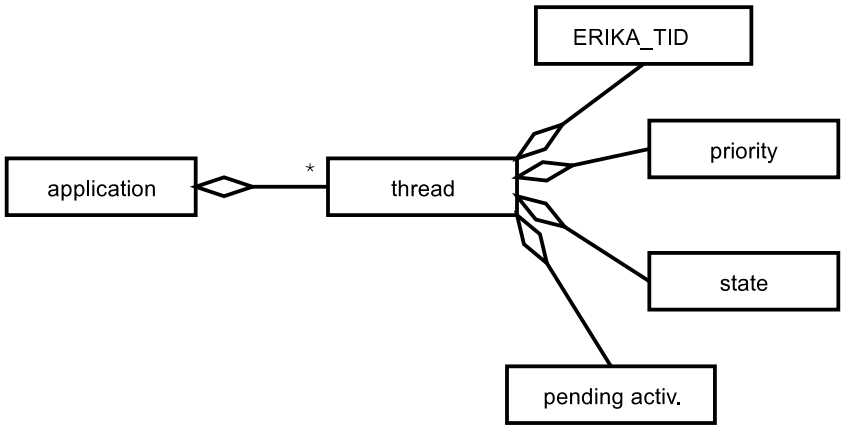


Figure 3.4 How the Kernel Layer manage an application.

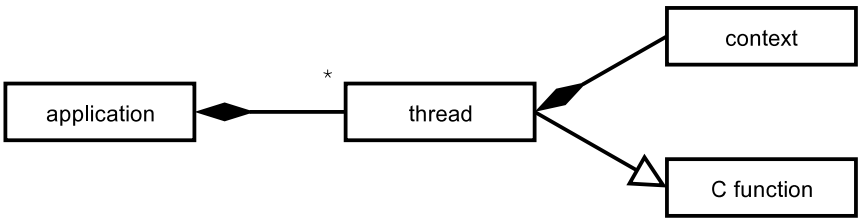


Figure 3.5 How the HAL manage an application.

The thread execution model is *one shot*: in that model, the body function simply represent an instance of a thread. Every time a thread is activated, the

body function is called; when the body function finishes, the thread instance is terminated. Note that when an instance terminates, the end of the body function automatically clean up the stack frame of the thread. Typically the end of a body function jumps to a HAL/Kernel function that is responsible of choosing the next thread that have to execute.

The memory model supported by the HAL is a *shared memory model*, where the whole address space is shared among all the threads in the system. This assumption that reflects in fact the typical hardware architectures of most microcontrollers, that typically do not support memory protection (usually they do not have a MMU).

The interface exported by the HAL, and the one shot thread model allow in general different threads to share the same stack. Anyway, stack sharing can only be exploited if a proper scheduling algorithm is used. In general, that scheduling algorithm must ensure that once a task is started and its frame is allocated on the stack, none of the tasks with lower priority and with which the task shares the stack can be executed. An example of this algorithm is, for example, the Stack Resource Policy, that allow resource and task sharing, preventing deadlocks to occur.

Depending on the scheduling algorithm used by the Kernel a proper HAL should be used. For example, if a Kernel Layer that allows tasks to block (e.g. providing blocking semaphores for synchronization) the HAL has not only to allow stack sharing but also allows different tasks to execute concurrently on different stacks in an interleaved way. Anyway, supporting a model that allow the use of more than one stack can lead to a degradation in performance for those application that do not need different stacks, and where all the threads share the same stack.

For that reason, the ERIKA Educational Project provides more than one HAL for each architecture, where each HAL is optimized for a particular usage, i.e. for mono and multi stack applications:

Monostack HALs All the threads and the interrupts share the same stack. That kind of HAL can only be used if the scheduling algorithm used by the Kernel Layer guarantees that a preempted thread cannot execute again before all the preempter threads have finished their instances.

Multistack HALs Every thread can have its private stack, or it can share it with other threads. This allows the Kernel Layer to use scheduling algorithms that *could* block threads in synchronization primitives (as for example, a fixed priority scheduler with semaphores) or for arbitrary reasons (as, for example, a classical round robin scheduler). Note

that this model can also adapt to architectures that support memory protection, since every thread stack *must* stay in different address spaces.

Please note that the monostack HAL behavior can be viewed as a particular case of the multistack HAL behavior, where all tasks shares the same stack. The main differences between the two versions are the execution time and memory required to perform the context switches (the Monostack HAL should be in general preferred if possible).

Finally the HAL, besides introducing the thread abstraction, offer a general abstraction of the peripherals provided by a particular architecture. This generally includes a set of functions that allow the application to interact with a particular interface, together with the definition of an user handler to be associated with each interrupt.

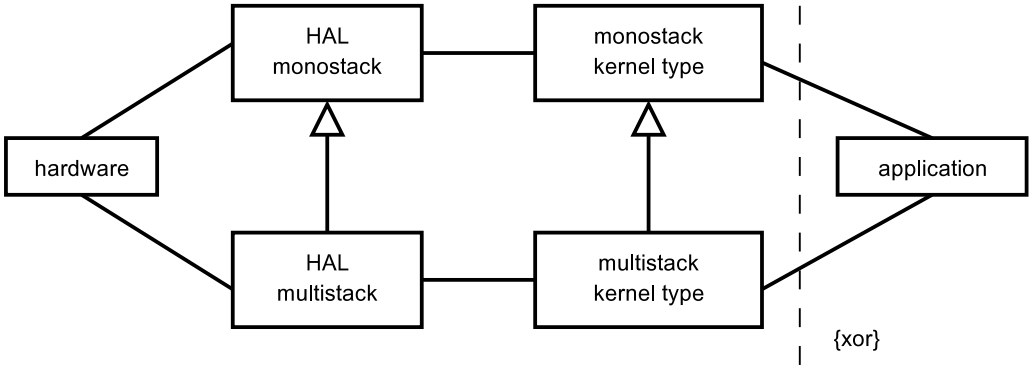


Figure 3.6 Monostack and Multistack HALs.

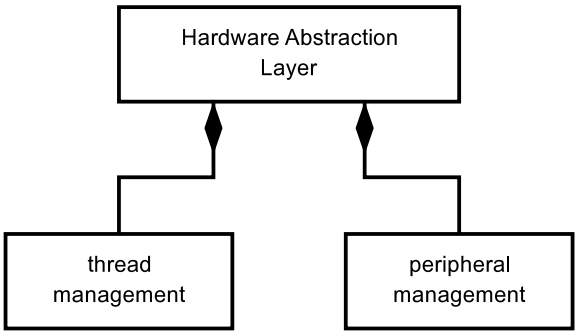


Figure 3.7 HAL services.

3.6 System modules

One of the objectives when designing an embedded application for systems that need a reduced memory footprint is to precisely tune which parts of the OS should be included in the final system, and which not. In that way both execution performance and memory footprint could be optimized.

The design of ERIKA Educational helps the process of optimizing the memory footprint, because each part of the system can be thought as a module that can be included or not in the final image file. All the Kernel is in fact compiled and linked together with the application, reducing as possible every overhead.

In particular, the user can choose the following options at compile time:

- The hardware architecture where the application will execute.
- The HAL to be used (mono stack or multi stack).
- The scheduling algorithm used by the application.
- Other external modules (as for example, semaphores, mailboxes, and so on).

Finally, the application will define the initialization of the HAL and Kernel data structures that are application dependent (for example, the number of threads and their priority).

3.7 Code portability

The division of the RTOS in two layer ease the porting of the Kernel Layer on top of different architectures. The use of a common interface for handling operating system calls ease also the migration from a scheduling algorithm to another. Moreover, the interaction between the application and the operating system is done using the syntax of an high level programming language, that allows the specification of the constraints in a programming environment independent from the architecture.

4 Kernel API

This chapter describes the Kernel Application Program Interface (API). That interface is composed by a set of types, constant and functions that can be called by the application to request its services. Unlike traditional APIs, the Application should not only *use* the interface, but also *define* the internal kernel data structures to adapt the kernel footprint to the specific application requirements.

4.1 Kernel Primitives

The Kernel Primitives can be grouped depending on their functionalities in the following way:

- thread management;
- thread management into interrupt handlers;
- shared resources management;
- interrupt management;
- utility functions.

Table 4.1 on the following page explains briefly all the primitives that are described in the following sections. To better understand the parameters of these functions, Table 4.2 on page 21 briefly lists all the types that the kernel assume to be defined.

The real dimension of each type and of each data structure can be tuned by the application. in fact, the applications must define the size of their types (i.e., the number of bits for an integer), and the dimension of the Kernel arrays to reduce the overall stack footprint.

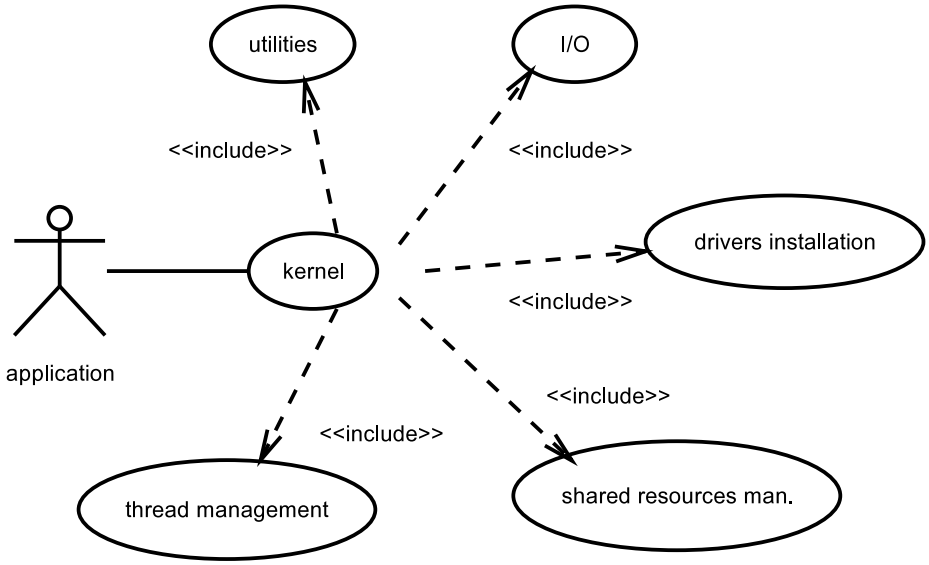


Figure 4.1 Primitives provided by the Kernel API

Table 4.1 Kernel primitives that can be used into the application

Thread management	`ERIKA_thread_make_ready()`
	`ERIKA_thread_activate()`
	`ERIKA_sys_scheduler()`
Thread enabling/disabling	`ERIKA_thread_enable()`
	`ERIKA_thread_disable()`
	`ERIKA_thread_isenabled()`
Thread management into Interrupt drivers	`ERIKA_IRQ_make_ready()`
	`ERIKA_set_handler()`
Shared resource management	`ERIKA_mutex_lock()`
	`ERIKA_mutex_unlock()`
Utility functions	`ERIKA_sys_gettime()`[1]
	`ERIKA_sys_panic()`
	`ERIKA_sys_reboot()`
	`ERIKA_sys_idle()`

[1] Only available if the symbol `__TIME_SUPPORT__` is defined.

Table 4.2 Kernel types and constants

Data type	ERIKA_TYPENACT	Number of thread pending activations
	ERIKA_TYPEPRIO	Thread priority
	ERIKA_TYPESTATUS	Thread status (ERIKA_READY, ERIKA_STACKED, ERIKA_IDLE)
	ERIKA_MUTEX	Mutex identifier
	ERIKA_TYPEIRQ	Interrupt number
	ERIKA_TID	Thread identifier
	ERIKA_TYPERELDLINE	Number of timer ticks used for indexing thread relative deadlines
	ERIKA_TYPEABSDLINE	Number of timer ticks used for indexing thread absolute deadlines
	ERIKA_TYPEENABLE	Enable/Disable state for a thread [1]
Constants	ERIKA_NIL	ERIKA_TID not valid (usually -1)

[1] Only available if the symbol __ENABLE_THREAD__ is defined in the makefile.

4.1.1 Thread management

Each thread in the system owns a flag that contains informations about its state. In particular, each thread at any instant of time can be in one of the following state:

- *stacked*, when the thread is the running one or it has been preempted by an higher priority thread.
- *ready*, when it has an instance that has been activated but never gone in execution—in particular, it never passed in the stacked state.
- *idle*, when a thread has finished all the instances that has been previously activated and it is waiting an explicit activation.
- *blocked*, when it is waiting in a synchronization point—this state is not present if the SRP algorithm is being used.

In the following paragraphs we will subsume the usage of the SRP scheduling algorithm, so we will not speak about any transition from and to the blocking state. These transitions will be documented later in the libraries where the blocking synchronization primitives will be documented.

In general the transition of a thread from a state to another one is triggered by the execution of a primitive. These primitives can have different meanings that can be summarized in the following basic behaviors:

Thread Activation Some primitives can *activate* a thread. In that case, the activated thread jumps in the ready state waiting the availability of the CPU for starting its execution. If the activated thread is already in the ready or stacked state, its current state is not changed, and the activation is stored in a pending activation counter.

Thread Wakeup Some primitives can *wake up* a thread that is *already* activated and that was not executing for some reason. For example, a thread could be waiting for a synchronization point that happens in a certain time, or a thread can finish its instance letting another thread execute.

Preemption Check Some primitives include in their behavior a check that is used to guarantee that the highest priority always preempt lower priority threads.

Figure 4.2 on the facing page summarizes all the possible state changes and the kernel primitives that cause them.

The threads are usually indexed using symbolic names, defined by the application developer. In the following, we suppose that a thread whose body is called `thread_name()` will be identified by the constant `TID_thread_name`. Note that the body functions that identifies a thread do not receive as parameter and do not return any value.

To better understand, here is a typical thread code:

```
void my_thread(void)
{
  /* thread body */
  /* new thread activation */
   ERIKA_thread_activate(TID_another_thread);
  /* end of current instance */
}
```

Note that primitives can be called using the standard C language syntax, even if what really happens when a primitive is called can be different from a typical function call.

In the following paragraphs the primitives of the Kernel layer are detailed.

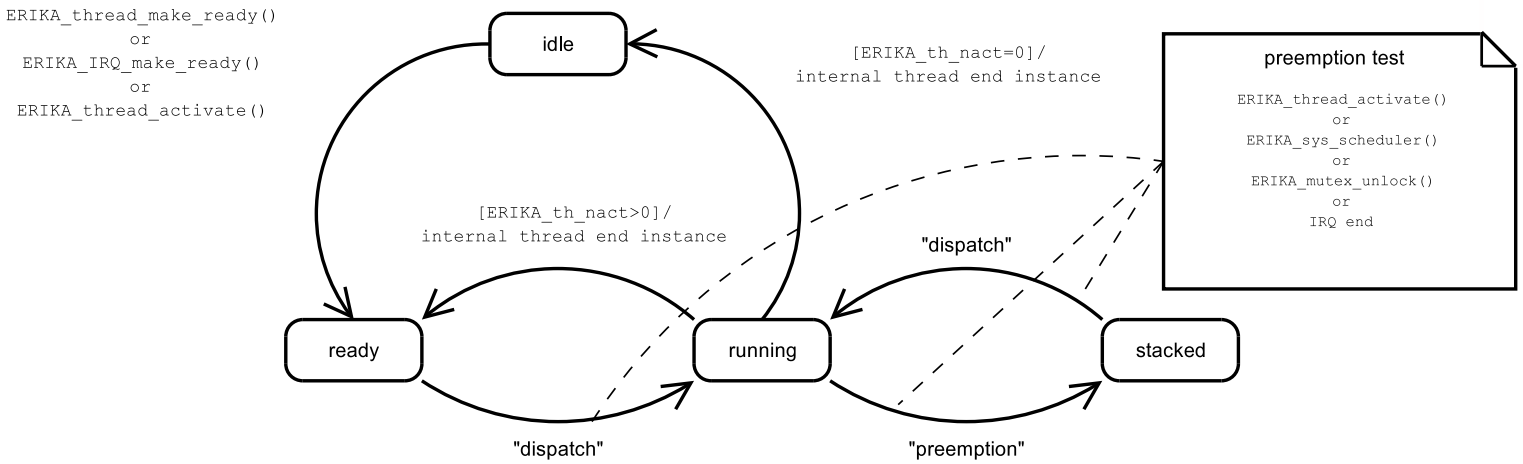


Figure 4.2 Thread transition diagram when SRP algorithm is being used (i.e. there is no `ERIKA_BLOCKED` status). The number of pending activations are named with `ERIKA_th_nact`. A thread can go in the idle and ready states only after the end of a call to `ERIKA_thread_end_instance`. Once a thread has been executed for a while, it can only jump from the running to the stacked state and vice versa.

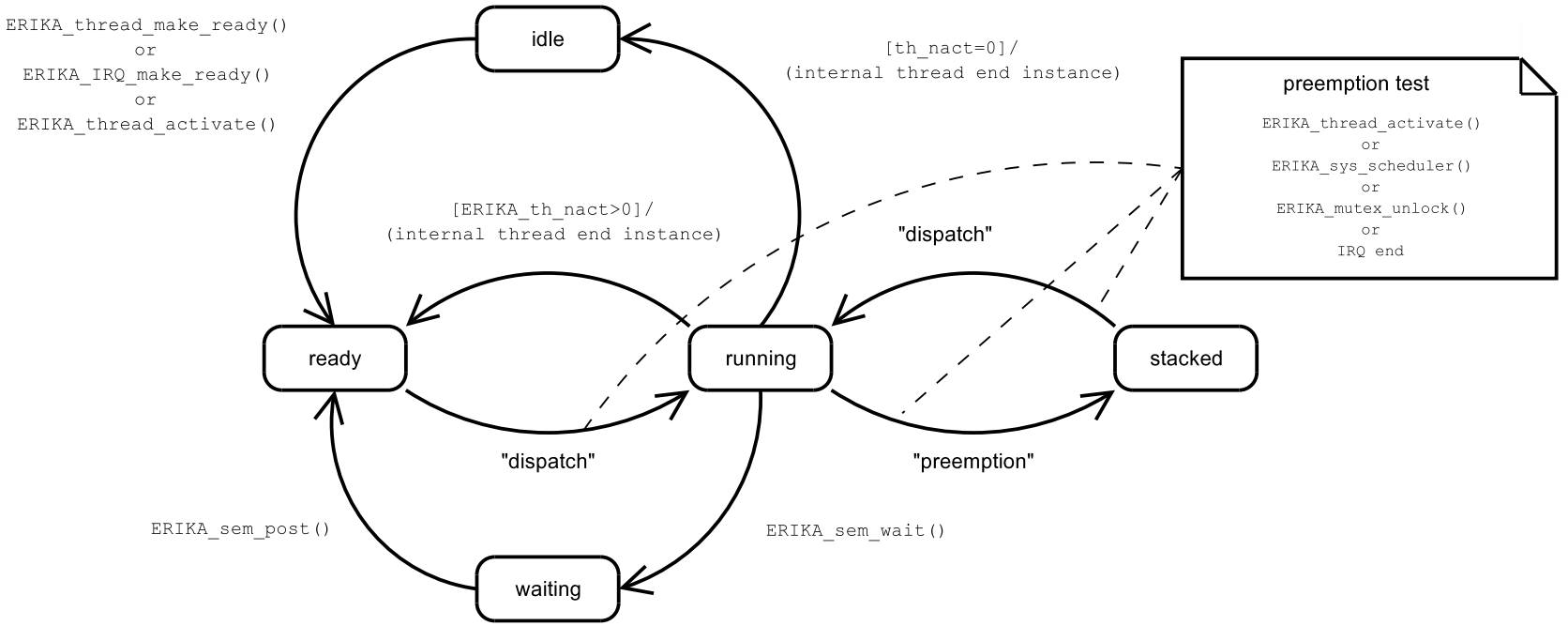


Figure 4.3 Thread transition diagram when there is a `ERIKA_BLOCKED` status. Please see also Figure 4.2 on the preceding page.

`void ERIKA_thread_activate(ERIKA_TID thread)`

Description This primitive activates the thread whose index is passed as parameter. As a result, the activated thread is put into the ready state if the running task has higher priority. If not, the activated thread becomes the running thread preempting the thread that called the primitive (see Figure 4.4). If the activated thread is already active (for example it is the running thread or it has been preempted), the thread state is not changed, and a pending activation is accounted for it. In any case the stack frame of the activated thread will be allocated as specified in the HAL configuration, independently from the thread that called this primitive. This primitive is equal to the `ERIKA_thread_make_ready()` except that it executes a preemption check.

See also `ERIKA_thread_make_ready()`, `ERIKA_sys_scheduler()`.

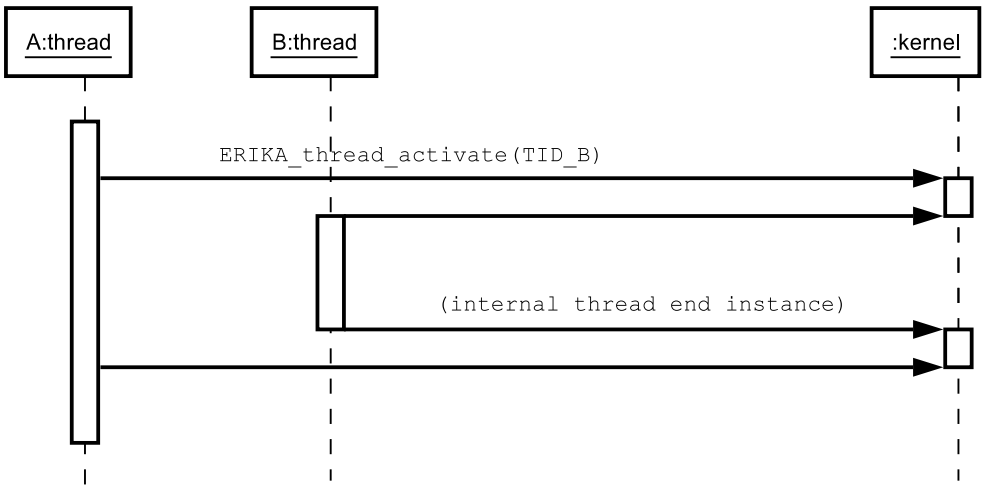


Figure 4.4 Context change due to the activation of the thread B that has higher priority with respect to the running thread A.

`void ERIKA_thread_make_ready(ERIKA_TID thread, ERIKA_TYPENACT nact)`

Description This primitive tells the kernel that the thread passed as first parameter will, when possible, execute for `nact` instances. This primitive never blocks the running thread, also if the activated thread has priority higher than it: it simply increment the pending activation counter and, if

necessary, switch the thread state to ready. An activated thread with priority greater than the running thread will execute after the next preemption check. This function can be useful if more than one thread has to be activated simultaneously.

The user can in fact activate with this primitive a few threads, and finally activate the last thread using `ERIKA_thread_activate()` or `ERIKA_sys_scheduler()`, causing the highest priority thread to preempt the running thread. Note that, although this primitive do not execute a preemption test, the following code:

```
ERIKA_thread_make_ready(t,3);
```

have a different behavior from:

```
ERIKA_thread_make_ready(t,1);
ERIKA_thread_make_ready(t,1);
ERIKA_thread_make_ready(t,1);
```

In particular, the second version can be interrupted by an interrupt routine just after the first or the second call, and that routine can execute a preemption check. The first primitive, instead, is executed atomically. This primitive is similar to `ERIKA_thread_activate()` except that it does not execute a preemption check.

See also `ERIKA_sys_scheduler()`, `ERIKA_thread_activate()`

Example

This is an example on how to use the `ERIKA_thread_make_ready` primitive:

```
ERIKA_thread_make_ready(TID_threadA, 1);
ERIKA_thread_make_ready(TID_threadB, 3);
ERIKA_sys_scheduler();
```

`void ERIKA_sys_scheduler(void)`

Description This primitive simply execute a preemption check to see if the highest priority thread in the ready queue has higher priority than the running thread. In that case, the highest priority thread in the ready queue preempts the running thread. Otherwise, nothing happens.

See also `ERIKA_thread_make_ready()`, `ERIKA_thread_activate()`.

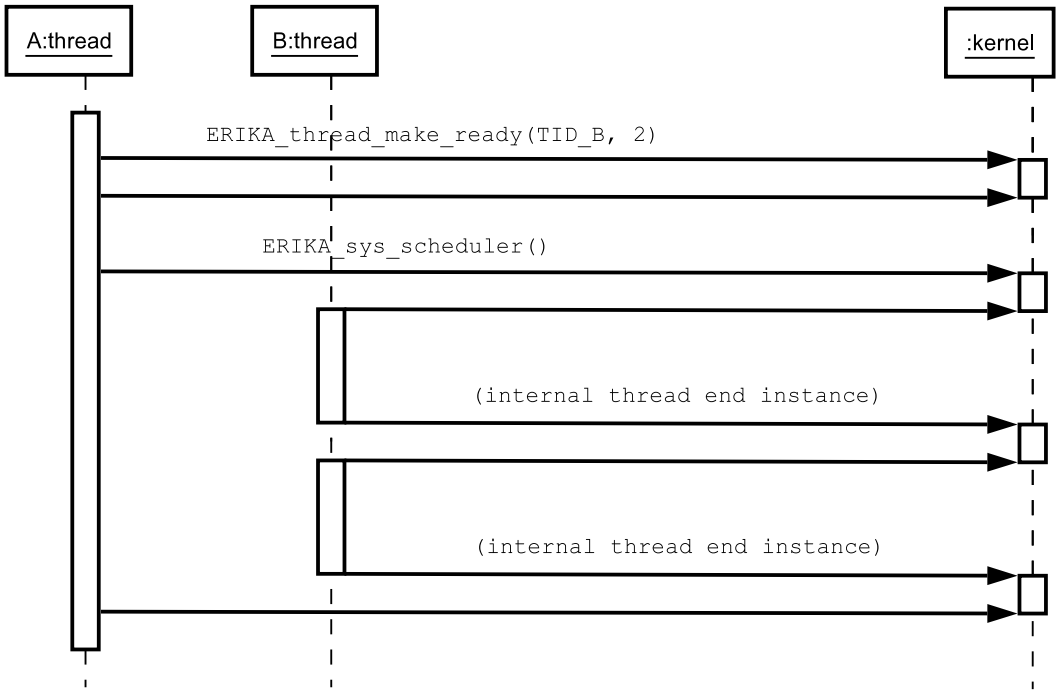


Figure 4.5 Context switches due to the activation of thread B by a call of `ERIKA_thread_make_ready()` by a lower priority thread A.

4.1.2 Thread enabling/disabling

The primitives described in this subsection can be used to toggle the enable/disable flag to a thread. That is, when a thread is enabled the behavior of the thread activation primitives ERIKA_thread_activate, `ERIKA_thread_make_ready` and `ERIKA_IRQ_make_ready` is as described in this Chapter. when a thread is disabled, all the thread activations results in a void operation.

Note that these primitives are available only if the Kernel is compiled with the `__ENABLE_THREAD__` flag. When the `__ENABLE_THREAD__` flag is defined, the user have to define also some data structures(see Section 5 on page 33). The initial state (enabled/disabled) of a thread depends on the initial values of these data structures.

```
void ERIKA_thread_enable(ERIKA_TID t)
```

Description This function enables thread `t` to be activated. If the thread t was already enabled, the function does nothing.

See also `ERIKA_thread_disable`, `ERIKA_thread_isenabled`.

```
void ERIKA_thread_disable(ERIKA_TID t)
```

Description This function disables thread `t` to be activated. If the thread `t` was already disabled, the function does nothing.

See also `ERIKA_thread_enable`, `ERIKA_thread_isenabled`.

```
ERIKA_TYPEENABLE ERIKA_thread_isenabled(ERIKA_TID t)
```

Description This function returns a flag that is `0` if the thread have been disabled using `ERIKA_thread_disable` (or if it was disable since the system start), different from `0` otherwise.

See also `ERIKA_thread_enable`, `ERIKA_thread_disable`.

4.1.3 Thread management into interrupt handlers

An interrupt handler is a routine written by the user and executed to handle the arrival of an hardware interrupt (hardware interrupts are typically raised by external interfaces). For that reason, the names of the primitives of this class start with '`IRQ_`'. Interrupt handlers are executed in special contexts that can be different (depending on the underlying architecture) from the context of the running thread that the handler interrupted. For that reason an interrupt handler can *only* call the primitives of this class.

Note Each Architecture differs on the way it specifies that a particular handler should be called in response to an interrupt. For that reason each HAL provides its own way to set interrupt handlers (anyway we tried to keep the names similar between the various HALs).

```
void ERIKA_IRQ_make_ready(ERIKA_TID thread, ERIKA_TYPENACT nact)
```

Description This function works as `ERIKA_thread_make_ready()` primitive.

See also `ERIKA_thread_make_ready()`.

4.1.4 Shared resource handling

The Kernel Layer provides a set of functions that can be used to enforce mutual exclusion between threads. These functions manages *mutexes*, that are basically binary semaphores statically defined and initialized. Each mutex can be addressed through an index that typically is named `MUTEX_mutexname`.

The Kernel Layer provides two functions for managing mutexes, that are `ERIKA_mutex_lock()` and `ERIKA_mutex_unlock()`. Every critical section that use mutexes must start locking a mutex and must end unlocking it. The Kernel Layer does not provide any synchronization primitive (such as condition variables) that can be used with mutexes.

The synchronization protocol used by these mutex depends on the particular Kernel used; the only thing that it can be assumed is that a correct usage of mutexes will enforce the mutual exclusion between threads. Please remember that the lock is a non-blocking primitive only if the SRP protocol or its variants are used.

`void ERIKA_mutex_lock(ERIKA_MUTEX mutex_id)`

Description This primitive locks a mutex not already locked by another thread. Nested locks of the same mutex in the same thread are prohibited. This function is a non-blocking function only if the SRP or a similar protocol is used.

See also `ERIKA_mutex_unlock()`.

`void ERIKA_mutex_unlock(ERIKA_MUTEX mutex_id)`

Description This primitive frees a mutex that was previously locked by the same thread. The unlocking of the mutex usually provokes a preemption check (so a high priority task that was waiting for the mutex can preempt the running task).

See also `ERIKA_mutex_lock()`.

4.1.5 Utility Functions

This subsection shortly describes a few functions that can be used into the application.

`void ERIKA_sys_panic(void)`

Description This function is invoked in case of an abnormal state of the application. Typically this function should reset the system and/or signal a system fault.

`void ERIKA_sys_reboot(void)`

Description This function reset the system.

`void ERIKA_sys_idle(void)`

Description This function is the function that should be used into the idle loop in the `ERIKA_dummy()` function (see Section 4.2 on page 32). On some architectures, this function leads to a reduced consumption CPU state. If the particular CPU do not support an idle low power state, the function simply does nothing.

```
void ERIKA_sys_gettime(ERIKA_TIME *t)
```

Description This primitive returns the timer value that is currently used as system clock. This function is only available if the user defines the symbol __TIME_SUPPORT__.

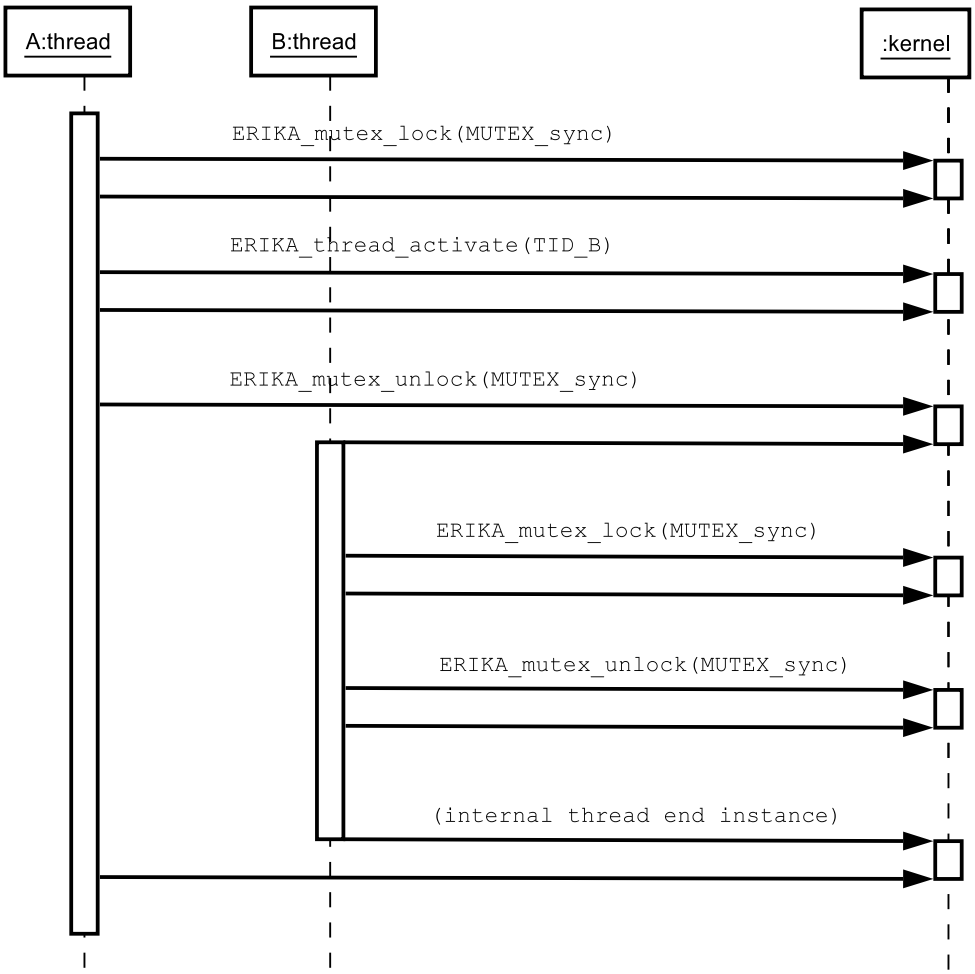


Figure 4.6 Example of context change due to the use of mutexes. In the figure, thread A has lower priority than thread B.

4.2 The ERIKA_dummy() thread

The startup point for an application is a function whose called `ERIKA_dummy()`[1] whose prototype is

```
void ERIKA_dummy(void)
```

The dummy function can be thought as the thread that is responsible of system initialization and of the idle loop. The dummy thread is always the lowest priority thread in the system, it does not have a `TID`, and it never ends. It is always active and for that reason it cannot be activated. In other words the `ERIKA_dummy()` function is used for:

- *System initialization*: the `ERIKA_dummy()` thread should initialize all the peripherals (e.g. interrupt controller initialization).
- *Thread activation*: it should also manage initial thread activations through the use of the `ERIKA_thread_make_ready()` primitive to proper activate the first threads that should be active.
- *Preemption*: the `ERIKA_dummy()` function usually call the `ERIKA_sys_scheduler()` or `ERIKA_thread_activate()` primitives to execute a preemption check.
- *Idle loop:* the `ERIKA_dummy()` function should execute an infinite idle loop. During the idle loop any kind of background code can be executed; if no code has to be executed, the `ERIKA_sys_idle()` primitive should be called.

For an example of the `ERIKA_dummy()` thread, see Chapter 6 on page 39.

[1] The name dummy() was chosen because the main() function is not available as startup point on all the architectures.

5 Kernels

THIS CHAPTER shortly describes all the peculiarities of each Kernel Layer implementation available. In particular, currently ERIKA Educational provides a *FP* (Fixed Priority scheduling with Preemption Thresholds and SRP protocols) Kernel. This chapter do not explain in detail the implementation of every primitive.

5.1 The FP Kernel

The scheduling algorithm used by the FP Kernel is the Fixed Priority with Preemption Threshold Scheduling Algorithm. Each thread in the system has a priority called *ready priority*, that is used to enqueue threads in the ready queue. When a task becomes the running task, its priority is raised to another priority level, called *dispatch priority* (that usually is greater than the ready priority). Lower priority threads do not execute until there are no higher priority threads ready for execution, so at any instant of time the thread with the highest priority is the running thread.

Threads can share resources, and mutexes are used to enforce mutual exclusion between them. Each mutex has assigned a ceiling, that is the maximum *ready* priority of the threads that use that mutex. When a mutex is locked, the priority of the task is raised to the mutex ceiling.

The protocol just showed avoids deadlocks, chained blocking and priority inversion. Moreover, all the threads can share the same stack, and dispatch priorities can be used to reduce the preemption between threads in a controlled way, further reducing the overall stack space needed. For more informations, please read the documentation on the ERIKA Educational web site[1] and their cited bibliography.

[1] `http://erika.sssup.it`

5.1.1 Types and data structures

For efficiency reasons, this kernel exports its internal data structures to the application. To initialize the system, the application should define and initialize these structures using the C language '{...}' initializer.

The advantage of this approach are twofold:

- most of kernel data structures can be allocated in ROM;
- the kernel arrays can be sized exactly to the number of threads and mutexes needed by a particular application.

Using that approach, the Kernel adapts itself to the particular application, without requiring additional primitives for thread creation and thread handling.

The disadvantage of this approach is that the application have to rely and have to know the kernel internal functions. Note that anyway this knowledge is very little, and can be always isolated in one simple file.

5.1.1.1 Priorities encoding

The FP Kernel implements a particular priority encoding. A priority can only be coded as a power of 2. In practice its binary representation is a `WORD` with all the bits set to '`0`' but one that is set to '`1`'. Bigger numbers define higher priorities.

If we consider a `WORD` of 32 bits, both `0x0001 0000` and `0x0000 0800` are valid priority encodings, whereas `0x0000 3000` not. For more information, see Section 9.3 on page 78.

5.1.1.2 Thread status encoding

The FP Kernel simplified the encoding of the thread status. In particular, these are the encoding of the states listed in Table 5.1 on the facing page.

5.1.1.3 Kernel types

The user can tune the size of some data types used by the Kernel to reduce the footprint of the application or to speedup the execution of the primitives. The size that should be typically used in an application is given in the example

Table 5.1 Possible thread status

Ready	Status set to `ERIKA_READY`, number of pending activations greater than one, thread inserted into the ready queue.
Idle	Status set to `ERIKA_READY`, number of pending activations equal to 0, thread *not* inserted into the ready queue.
Stacked	Status set to `ERIKA_STACKED`, number of pending activations greater than 0, thread inserted into the stacked queue.
Blocked	Status set to `ERIKA_BLOCKED`, number of pending activations greater than 0, thread inserted into the blocked queue.

that are distributed with the project. However, the user can change them to adapt its application to specific needs.

The types that can be defined are:

ERIKA_TYPENACT This type is used to store the pending activations of a thread. The value $2^{\text{number of bits}} - 1$ is the maximum number of pending activations that a process can have.

ERIKA_TYPEPRIO It is used to store a thread priority. Remember that the number of bits plus one are the number of different priorities available on the system (from `0x00...0` to `0x80...0`).

ERIKA_TYPESTATUS It is used to contain the thread status, that can have a value `READY`, `STACKED` or `BLOCKED`.

ERIKA_TYPEENABLE This type is used to store a flag that influences the activation state of a thread, that is, if thread activations are discarded or not. This type is available only if the symbol __ENABLE_THREAD is defined into the ERIKA Educational makefile.

ERIKA_MUTEX This is the type that will identify a `MUTEX` in the system. In practice, it is an index in the internal kernel mutex arrays, but the user has to identify a mutex '`name`' with the symbol '`MUTEX_name`' instead of just a number.

5.1.1.4 Kernel data structures

Here is a list of data structures the user must define and initialize[2]:

[2]Arrays whose name begins with '`th_`' should be dimensioned on the number of threads in the system. Arrays whose name begins with '`mutex_`' should be dimensioned on the

const ERIKA_TYPEPRIO ERIKA_th_ready_prio[] This ROM data structure should contain for each TID the priority used for queuing the threads into the ready queue.

const ERIKA_TYPEPRIO ERIKA_th_dispatch_prio[] This ROM data structure should contain for each TID the priority that a thread has when it become the running thread, according to the Preemption Threshold algorithm.

ERIKA_TYPESTATUS ERIKA_th_status[] This is used for storing the thread status. All the entries of the array must be set to IDLE. To activate a thread you should use ERIKA_thread_make_ready() or ERIKA_thread_activate(). Note that the use of a thread state is useful for consistency checks during application testing. In some cases the statuses and the th_status[] array can be removed.

ERIKA_TYPENACT ERIKA_th_nact[] This array is used to store the threads pending activations. All the entries must be set to '0'.

ERIKA_TYPENACT ERIKA_th_next[] This array is used to store the ready queue *next* pointers[3]. All the entries should be initialized to ERIKA_NIL.

ERIKA_TYPEPRIO ERIKA_sys_ceiling This value stores the system ceiling used for the preemption checks. It should be initialized with a value of '0'.

ERIKA_TID ERIKA_rqfirst This value points to the first task into the ready queue. It should be initialized with ERIKA_NIL.

ERIKA_TID ERIKA_stkfirst This value points to the first task into the stacked queue, that is the running task. It should be initialized with ERIKA_NIL.

ERIKA_TYPEENABLE ERIKA_th_enabled[] This array contains the activation state for each thread. An initial value for a single thread of 0 means that every activation of the thread is discarded. A value of 1 means that the thread activation will follow its usual behavior. This data structure is available only if the symbol __ENABLE_THREAD__ is defined in the ERIKA Educational makefile.

const ERIKA_TYPEPRIO ERIKA_mutex_ceiling[] This ROM array stores the ceiling of each mutex.

number of mutexes in the system. When not stated, the variables are stored in RAM.

[3]The pointers are actually TID indexes.

ERIKA_TYPEPRIO ERIKA_mutex_oldceiling[] This array is used to store the system ceiling of the system each time a resource is locked. It does not need to be initialized with a particular value.

Note that the application can start with a set of tasks already active. In that case, the data structures presented before should be initialized in a way that the activated tasks should be inserted in the ready queue in the right positions (following the priority order) with a pending activation of at least one.

5.1.2 An Example

As an example, in this section we depict a typical situation that explains the internal mechanisms of the kernel.

Suppose to have the thread set depicted in Table 5.2 with the corresponding initialization values. The thread set is composed of six threads, each with different priority; there is a non-preemption group composed by thread 4 and 5 (they have the same dispatch priority), and there is a shared resource used by threads 1 and 3 through a mutex. The system ceiling[4] has a starting value of `0x01` (because the thread 1 starts on the stack).

Table 5.2 The example thread set (the ERIKA_ prefix on the constants has been removed)

Thread	Ready	Dispatch	*Initial Values*			*After the events*		
Number	Priority	Priority	Status	Pend. Act.	next	Status	Pend. Act.	next
0	0x01	0x01	READY	0	NIL	STACKED	1	NIL
1	0x02	0x02	READY	0	NIL	STACKED	1	0
2	0x04	0x04	READY	0	NIL	READY	0	NIL
3	0x08	0x08	READY	0	NIL	READY	1	NIL
4	0x10	0x20	READY	0	NIL	STACKED	1	1
5	0x20	0x20	READY	0	NIL	READY	1	3

Then suppose that these events happen in the system:

1. System start: Thread 0 is activated, so it starts in the ERIKA_STACKED state;

[4] As there are only 6 threads in this example, only the first 8 bits are showed; the MSB is considered equal to `0x00`.

2. Thread 1 arrives and preempts Thread 0 since it has a higher priority;
3. Thread 1 locks the mutex; therefore, the system ceiling is updated to `0x0B`;
4. Thread 3 arrives. Even though its priority is greater than the priority of Thread 1, it is not greater than the current system ceiling: hence it goes in the ready queue;
5. Thread 4 arrives and become the running thread;
6. Thread 5 arrives and since it belongs to the same non-preemption group of Thread 4, and Thread 4 is still active, then it goes into the ready queue.

Finally, suppose that all the threads shares the same system stack.

After these events, the configuration of the kernel data structures is shown in Figure 5.1.

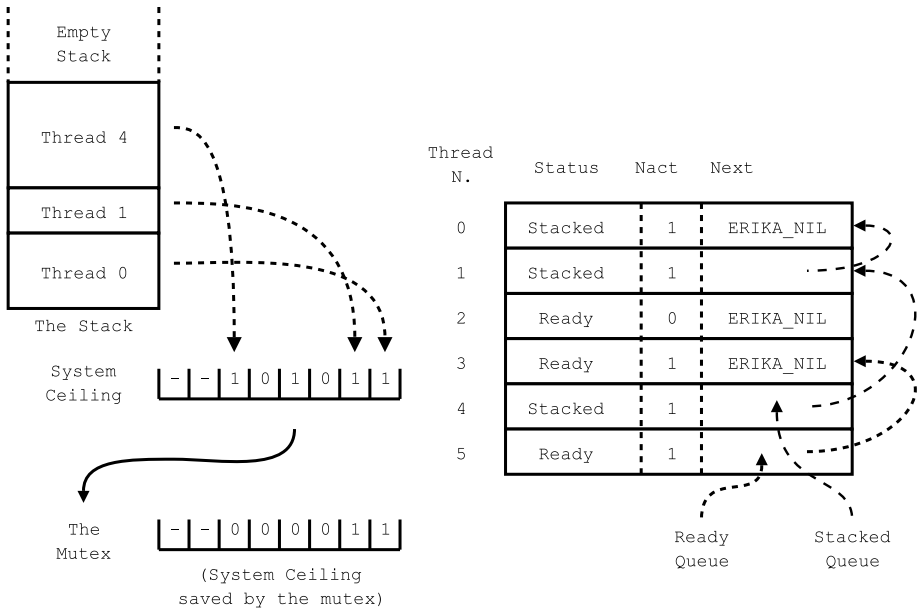


Figure 5.1 Kernel Data structures snapshot after the events listed in Section 5.1.2 on the preceding page

6

How to write an application

THIS CHAPTER contains some information useful for writing an application using the ERIKA Educational Project. In particular, it contains some details on the conventions for the names of the application files, of the predefined environment variables, on the writing of the `dummy()` thread body, and on the definition of the HAL/Kernel symbols.

6.1 Application structure

The ERIKA Educational project, to ease the configuration and development of an application, impose a set of rules that every application must follow. These rules only influences some files that are used to tune particular HAL/Kernel aspects, such as the predefined types or the definition of internal data structures.

6.1.1 Directories

Every ERIKA Educational Application is composed by a set of files that resides under the same directory, that will be called in the following the *Application Directory*.

6.1.2 Mandatory files

These are the files that *must* be provided into the application directory in order to successfully compile an application.

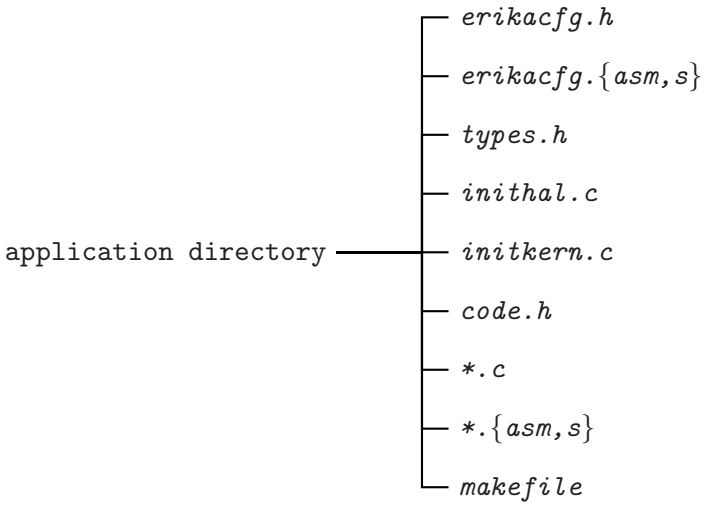


Figure 6.1 A typical application directory structure

These files are required because they are included by the HAL/Kernel source code; this is the reason because the Kernel and the application are compiled together at each build.

erikacfg.h This file is included by every include file used in the system. The purpose of this file is to define the main configuration options of the Kernel. In practice, it contains the definition of a set of symbols that allows to choose the hardware architecture, the scheduler, the libraries and devices used. A complete list of the symbols used into the Kernel can be found in Appendix A on page 107.

It can also be used to define the symbols that identifies the threads, the mutexes, and so on. Typically we use the pattern `TID_thread_name` and `MUTEX_mutex_name` as a convention for the naming of these objects. This file may also contain the definition of the number of threads/mutexes in the system (called `THREAD_MAX` and `MUTEX_MAX`), and some other application-specific symbol

Some architectures require a similar file that is included by the HAL assembler files. The name of that file and its syntax can vary and it strictly depends on the assembler conventions used by the target platform.

Finally note that, with the makefiles (see Section 6.1.3 on the facing page), the symbols listed into the `ERIKAOPT` environment variable are

also pre-defined in all the files compiled using proper command-line options. For example, there is no need to write a line such as'`#define __MONO__`' in the `erikacfg.h` file because, if used, it will be inserted into the `ERIKAOPT` environment variable of the makefile.

`types.h` This file contains all the type definitions that have to be defined to use a particular HAL/Kernel/library into an Application. In this way the user is allowed to control the the RAM/ROM footprint of Kernel data. The file `types.h` is a typical C file.

6.1.3 The Makefile

The ERIKA Educational project provides a standard makefile to help the compiling phase of the application. The makefile will take care of compiling only the needed source files, passing the right compiler options, and checking for most of the files dependencies.

The makefile subsumes the existence of the following environment variables:

`H8BASE` that contains the base directory used during the installation process of the target compiler tool-chain.

`ERIKABASE` that contains the base directory that stores the ERIKA Educational files.

Here is an example of commands that have to be issued under the GNU bash under the Cygwin environment.

```
export H8BASE=/cygdrive/d/programs/h8

export ERIKABASE=/cygdrive/d/madess/erika
```

The makefile that the user have to write is quite simple. Here is an example:

```
ERIKAOPT = __H8__ __MONO__ __FP__

PROJ = rmtest

APP_SRCS = code.c initdrvs.c inithal.c initkern.c

include $(ERIKABASE)/config/config.mak
```

The first line is used to configure the compilation process. In the example, the application will compile using the GNU tool-chain for H8 using a Fixed

priority kernel over a Mono Stack HAL. A complete list of the symbols can be found in Appendix A on page 107.

The second line simply says the name (not the extension, that is target dependent) of the final image.

The third line simply list the files (that resides on the same directory of the makefile) that compose the application and that must be compiled with the rest of the files implicitly chosen in the first line.

Finally, the fourth line includes the standard definitions that contains all the rules needed to compile a makefile. These rules are contained in the files under the `config` directory.

6.1.4 Object files

The compilation process creates a sub-directory called '`out`' under the application directory. It contains all the (object) files created during the build, and the final application image that must be downloaded into the target hardware.

The standard makefile provided with the project will create its files only into the '`out`' directory. The removal of that directory correspond to the cleanup operation typically issued using the '`make clean`' line command.

Note Every build recompiles all the HAL/Kernel/Libraries that are used in the application, and that these files are put in the same out directory that is used to store the application files. For that reason the file names used for the application files *must* be different from the file chosen in the other directories of the source tree.

6.1.5 Other conventions

The information provided in this subsection are not mandatory, but represent some guidelines that help to write applications with a common basic file structure. The test applications presented under the `tests` directory tries to follow that guidelines.

In particular, we recommend to use these file names for the items listed below:

`initkern.c` This file contains all the definitions of the internal kernel variables. The symbols defined here depends strictly on the particular kernel that has been configured into the Makefile.

inithal.c This file contains all the definitions of the internal HAL variables. The symbols defined here depends strictly on the particular HAL and on the particular Stack Model (Mono Stack, Multi Stack) that has been configured into the Makefile.

code.h This file contains the definition of the global data and of the thread bodies.

***.c**, ***.{asm,s}** These are the application C and Assembler files.

6.2 The ERIKA_dummy() thread

This section contains a little example on how the `ERIKA_dummy()` thread (see Section 4.2 on page 32) should be written. Some more examples are available under the `tests` directory. Basically, the `ERIKA_dummy()` thread of this example is divided in three parts (in turn):

1. *Initialization.* In the first phase, the `ERIKA_dummy()` thread should initialize all the application data structures and all the peripherals present in the system. Note that the ERIKA_dummy thread is executed with

2. *Thread activation and preemption.* The application threads that must be activated at startup are put in the ready queue and then scheduled.

3. *Idle Time.* The `ERIKA_dummy()` thread does an infinite loop that is used to spend the system idle time. During this time, the system can call the `ERIKA_sys_idle()` primitive that eventually put the CPU in a low power state.

The `ERIKA_dummy()` thread is assumed to *never* end!

Here is the example:

```
void ERIKA_dummy(void)
{
  /* Initialization */
  my_init();
  /* Thread activation and preemption */
  ERIKA_thread_make_ready(TID_thread1,1);
  ERIKA_thread_make_ready(TID_thread2,1);
  ERIKA_thread_make_ready(TID_thread3,1);
  ERIKA_sys_scheduler();
  /* Idle Time */
  for (;;) {
    background_activities();
    ERIKA_sys_idle();
  }
}
```

7 Peripherals

THIS CHAPTER contains some information on the interface exported by the various HALs that allow to handle the various target platform specific peripherals.

7.1 H8 RCX

This section contains the list of functions that can be used with the RCX peripherals.

The full functions list can be found in **include/h8/drivers.h**. The full prototypes can be found in **include/h8/inline_d.h**, **src/h8/func_d.c** and **src/h8/drivers.s.**

7.1.1 Interrupt handling

`ERIKA_enable_irq(ERIKA_TYPEIRQ irq)`

Description Enable the specified *irq* interrupt source to request interrupts.

`ERIKA_disable_irq(ERIKA_TYPEIRQ irq)`

Description Disable the specified *irq* interrupt source to request interrupts.

Interrupt source	H8 Interrupt type	Costant (*irq*)
RUN button	IRQ0	RUNBUTTON_IRQ
ON/OFF button	IRQ1	ONOFFBUTTON_IRQ
16 bit Timer	ICIA (Input capture A)	ICIA_IRQ
	ICIB (Input capture B)	ICIB_IRQ
	ICIC (Input capture C)	ICIC_IRQ
	ICID (Input capture D)	ICID_IRQ
	OCIA (Output compare A)	OCIA_IRQ
	OCIB (Output compare B)	OCIB_IRQ
	FOVI (Overflow)	FOVI_IRQ
8 bit Timer Channel 0	CMI0A (Compare match A)	CMI0A_IRQ
	CMI0B (Compare match B)	CMI0B_IRQ
	OVI0 (Overflow)	OVI0_IRQ
8 bit Timer Channel 1	CMI1A (Compare match A)	CMI1A_IRQ
	CMI1B (Compare match B)	CMI1B_IRQ
	OVI1 (Overflow)	OVI1_IRQ
Serial Interface	ERI (Receive error)	ERI_IRQ
	RXI (Receive end)	RXI_IRQ
	TXI (TDR empty)	TXI_IRQ
	TEI (TSR empty)	TEI_IRQ
A/D Converter	ADI (Conversion end)	ADI_IRQ
Watchdog timer	WOVF (WDT overflow)	WOVF_IRQ

Table 7.1 Constants used to specify an *irq* interrupt source

Button	Constant (*button*)
Run	RUN_BUTTON
OnOff	ONOFF_BUTTON
View	VIEW_BUTTON
Prgm	PRGM_BUTTON

Table 7.2 Constants used for buttons identification

`ERIKA_interrupt_on()`

Description Each interrupt request (generated by an enabled source) is accepted.

`ERIKA_interrupt_off()`

Description Each interrupt request (generated by an enabled source) is masked. Such a request remains pending (Note that further requests will be discarded, since H8 is able to store at most one pending interrupt request).

7.1.2 Buttons

`ERIKA_button_pressed(ERIKA_UINT8 button)`

Description Return 1 if the specified *button* is pressed (when the function is invoked), 0 otherwise.

`ERIKA_button_irq_mode(ERIKA_UNIT8 mode)`

Description It determines whether button interrupt request (IRQ1-IRQ0) are level-sensed or sensed on the falling edge. Note that when level-sensed, interrupt requests are continuosly generated during all the time the button is pressed.

Sensing	Constant (*mode*)
OnOff edge, Run edge	ONOFF_EDGE_RUN_EDGE
OnOff edge, Run level	ONOFF_EDGE_RUN_LEVEL
OnOff level, Run edge	ONOFF_LEVEL_RUN_EDGE
OnOff level, Run level	ONOFF_LEVEL_RUN_LEVEL

Table 7.3 Constant for button interrupt request sensing

7.1.3 LCD

```
ERIKA_LCD_display(ERIKA_UINT16 num)
```

Description Write the integer *num* on LCD. If *num*>9999 then 9999 is displayed; if num<-9999 then -9999 is displayed. Note that this function uses some ROM routine of the RCX.

```
ERIKA_LCD_clear(ERIKA_UINT16 num)
```

Description Clean the LCD. (Note that this function uses some ROM routine of the RCX).

7.1.4 Motors

There are three possibilities: fixed power motor (FP), variable power motor (VP) and Pulse Width Modulation motor (PWM).

7.1.4.1 FP and VP motors

FP motors can only be turned on (left or right) and turned off (brake or float). With VP and PWM motors you can also set the power: PWM uses all the three interrupt sources of channel 1 of 8 bit timer, so if your application needs to use that for different purposes you cannot use PWM. However PWM motors have a more linear behaviour and a better granularity than VP motors.

```
ERIKA_motorX_action(ERIKA_UINT8 action)
```

Description It determines the action of the motor X, where X can be A, B, C.

Motor Action	Constant (*action*)
Turn left	`MotorGo0`
Turn right	`MotorGo1`
Brake	`MotorBrake`
Float	`MotorFloat`

Table 7.4 Constants for motor action

Power	Constant (*power*)
minimum	`PWM_MIN_MOTOR_POWER / VP_MIN_MOTOR_POWER`
maximum	`PWM_MAX_MOTOR_POWER / VP_MAX_MOTOR_POWER`

Table 7.5 Constants for power range of motors

```
ERIKA_motorX_power(ERIKA_UINT8 power)
```

Description It determines the power of the motor X. *Note that X can only be A or C for PWM motors, while B is also available for VP motors.*

7.1.4.2 PWM motors

We used the 8 bit timer to generate a wave that have to be applied to the output ports: in particular, we programmed a 1 KHz PWM, that can be used to set a voltage between 0 and Vcc, setting in that way the intensity of a light, the speed of a motor, and so on. PWM can be used only with ports A and C, whereas motor B can be used or not depending on the particular application configuration.

These are the macros to be used when setting the PWM:

__ERIKA_PWM_USED__ must be defined to use PWM motors

__ERIKA_PWM_TOFF_BRAKE__ during the OFF period (PWM period $T = T_{OFF} + T_{ON}$) Vcc is set to both pins of the port. That can be useful whenever we are controlling some motors, because in that case the the motor is set to brake, giving more precision in the control. If not defined, during the off period the port is set to float.

__ERIKA_PWM_FAST__ if set, the PWM implementation is as fast as possible. That is,

- port B cannot be used by the application because it is set to 0 every T seconds
- ports A and C can be controlled in terms of voltage only (that is,the duty cycle of the PWM, but not not in terms of polarity applied to the port pins. That is, during T_{ON} a pin is set to Vcc and the other to 0, whereas during T_{OFF} the voltage is 0 (or Vcc if __ERIKA_PWM_TOFF_BRAKE__ is set) on both pins. If the port has a motor connected to it,that means that only the velocity of the motor can be set,but not the rotation sense .

Available functions:

`ERIKA_PWM_init()`

Description called automatically by ERIKA_init().

`ERIKA_PWM_set(voltageA, voltageC, [polarityA], [polarityC])`

Description voltageA and voltageC set the voltage level that is available on ports A and C. Their value must be in [0...ERIKA_PWM_MAX]. polarityA and polarityC must be specified when __ERIKA_PWM_FAST__ has not been defined: they set the pins where the voltage will be applied during T_{ON}. They can only have values inside the following table:

polarityA (binary value)	polarityC (binary value)	voltage on each pin of a port
ERIKA_PWM_A_PWR00 (00000000)	ERIKA_PWM_C_PWR00 (00000000)	0 on both pins
ERIKA_PWM_A_PWR01 (01000000)	ERIKA_PWM_C_PWR01 (00000001)	0 on one, Vcc on the other
ERIKA_PWM_A_PWR10 (10000000)	ERIKA_PWM_C_PWR10 (00000010)	Vcc on one, 0 on the other
ERIKA_PWM_A_PWR11 (11000000)	ERIKA_PWM_C_PWR11 (00000011)	Vcc on both

`ERIKA_PWM_end()`

Description Disable the timer, the interrupts and set voltage equal to 0 (float) on port A and C.

Implementation notes Due to the possible utilization of PWM management and IR communication at the same time, 8 bit Timer channel's selection is based on the particular options' configuration chosen by the user. This choice is made according to the following rules:

- If `__IR_SUPPORT__` is defined, PWM motors' management is moved on channel 0, preventing the sound support to work. So when you need IR communication and PWM motors' management you cannot use `__PLAY_NOTE_SUPPORT__` and `__PLAY_BUFFER_SUPPORT__`.
- If `__IR_RX_SUPPORT__` or no ir support is defined, PWM motors' management uses 8 bit timer channel 1.

To implement the PWM support we used 3 interrpts linked to the 8 bit timer:

CMIxA,CMIxB when this interrupt raise, $T_{OFF}$is started for ports A and C, respectively. A voltage equal to 0 (or Vcc) is applied to both pins of each port.

OVIxA starts T_{ON}. the right voltage is applied to both ports,

where **x** is the number of the used channel.

That is, the bigger the value inside CMPA and CMPB, the bigger the duty cycle of the PWM (that is, the average voltage (speed in the case of a motor) on the corresponding port).

In FAST mode, when there is a match with CMPA or CMPB, the corresponding port passes in the T_{OFF}phase.T_{OFF}ends whenever an overflow occurs (OVI1A is called): in that case, a different voltage is applied to both ports. If the duty cycle is 0¸the corresponding port is not set.

See also IR communication section, Sound section

7.1.5 Sensors

`ERIKA_ad_select(ERIKA_UINT8 c)`

Description Select ports and mode for conversion. Note that bits of c have this meaning (from the MSB to LSB): "`pqrmccc`". Where:

- `p` : 0 if port 1 has a passive sensor, 1 otherwise;

- `q`: 0 if port 2 has a passive sensor, 1 otherwise;
- `r`: 0 if port 3 has a passive sensor, 1 otherwise;
- `m`: 0 convert in single mode, 1 scan mode;
- `ccc`: specify the channel. Consider that H8 supports eight channels, but RCX uses only three: channel 0 for port 3, channel 1 for port 2 and channel 2 for port 3. See H8 manual[1] for more details on channels and scan mode.

`ERIKA_ad_start()`

Description Start convertion of selected ports.

`ERIKA_ad_start_fast()`

Description Start fast convertion of selected ports (half time needed).

`ERIKA_ad_stop()`

Description Stop convertion. It is useful in scan mode.not aware

`ERIKA_ad_reset_end_flag()`

Description It should be used at the end of the A/D handler to avoid continuous interrupt requests.

`ERIKA_ad_get_portX()`

Description It returns the value stored at the end of last A/D convertion of port X, where X=1, 2, 3.

7.1.6 Sound

It is possible to use the on board speaker to play notes or melodies in background: this means that application invokes the play functions and go on without polling.

[1] http://semiconductor.hitachi.com/H8

7.1.6.1 Play single notes

`ERIKA_play_note(ERIKA_UINT8 note, ERIKA_UINT8 length)`

Description Play *note* for the specified *length* time.Note that if you try to play a new note while the previous one is still playing, new note is ignored: to avoid this call the function **`ERIKA_play_note_stop()`** just before it. See table below for possible values of *note*.

Furthermore, consider that:

note length time = *length* * 2 ms

This means that a note can play from 0 to 510 ms.

`ERIKA_play_note_stop()`

Description It stops playing current note.

Note	Octave	Frequency (Hz)	Constant (*note*)
DO	0	1000	_DO0
RE	0	1126	_RE0
MI	0	1250	_MI0
FA	0	1330	_FA0
SOL	0	1497	_SOL0
LA	0	1667	_LA0
SI	0	1880	_SI0
DO	1	2000	_DO1
RE	1	2252	_RE1
MI	1	2500	_MI1
FA	1	2660	_FA1
SOL	1	3012	_SOL1
LA	1	3333	_LA1
SI	1	3788	_SI1
DO	2	4032	_DO2
RE	2	4545	_RE2
MI	2	5000	_MI2
FA	2	5319	_FA2
SOL	2	5952	_SOL2
LA	2	6579	_LA2
SI	2	7575	_SI2

Table 7.6 Constats for notes

7.1.6.2 Play melodies

```
ERIKA_play_buffer(
struct ERIKA_note_type * buffer_addr,
ERIKA_UINT8 num_note, ERIKA_UINT8 mode)
```

Description Play a melody wich is composed by *num_note* notes stored in a buffer whose address is *buffer_addr*. The melody can be played once (*mode*=PLAY_ONCE) or continuously (*mode*=PLAY_BACKGROUND).

Each element of the buffer has this form:

{ *note*, *length*, *pause_length* }
where *note* is the note to be played for a *length* time, as specified above, while *pause_length* specifies the length of the pause which will follow the note.

Let's see an example:

```
struct ERIKA_note_type my_melody[] =
{ {_DO0,  100, 80},
  {_RE0,  100, 80},
  {_MI0,  100, 80},
  {_FA0,  100, 80},
  {_SOL0, 100, 80},
  {_LA0,  100, 80},
  {_SI0,  100, 80},
  {_DO1,  100, 250},
  {_DO0,    0, 250}
};
```

So, to play the scale in background you have to do this:

```
ERIKA_play_buffer(my_melody,9,PLAY_BACKGROUND);
```

Note that if you specify *pause_length*=0 no pause will be played; furthermore, if you specify *length*=0 then *note* is not played. As you can see, it is possible for example to play a note for a time which is bigger than 510ms: all you need is to specify *pause_length*=0 and play the same note after that.

`ERIKA_play_buffer_stop()`

Description It stops to play current note.

`ERIKA_is_playing_buffer()`

Description It returns 1 if a melody is currently in play mode, 0 otherwise.

Please note: If you need sound support in your application (`__PLAY_NOTE_SUPPORT__` or `__PLAY_SOUND_SUPPORT__`), keep in mind that sound uses channel 0 of the 8 bit timer, so you cannot use also `__IR_SUPPORT__` and `__ERIKA_PWM_USED__` at the same moment.

See also PWM motors' section, IR Communication

7.1.7 InfraRed (IR) Communication

Lego RCXs are able to communicate using the provided infra-red (IR) transmitter/receiver, that is mapped to the H8 serial interface. By using a correct temporization in switching on and off the IR led (nominally at a frequency of 38 KHz) IR transmitters are able to send IR messages to other components as RCXs or IR Towers.

IR communication protocol works as shown in Figure 7.1.

When a component (e.g., a RCX) needs to transmit, it invokes a sending request. At the end of the transmission an interrupt notifies that the data sending was over, so the transmitter can control if transmission was OK or not, and is able to transmit other data. Note that if channel is busy, it's possible, at IR startup, to decide to wait some time if channel becomes free, or to raise immediatly the interrupt. On the receiving side, at the end of the data receiving an interrupt is raised to notify that a new data is available. Note that after one of the interrupts is raised, reception is disabled, so the application should re-enable it explicitly (if it need reception enabled).

When the component needs to transmit only one byte per transmission, transmission format uses a 9 byte packet for each byte that have to be transmitted, as explained in various web pages [5].

When the component needs to transmit more than one byte, transmission format adds 7 bytes of header and 2 bytes of trailer, and sends each byte together with its complement.

It's possible to request ACK support adding `__IR_ACK_REQ__`. In this case:

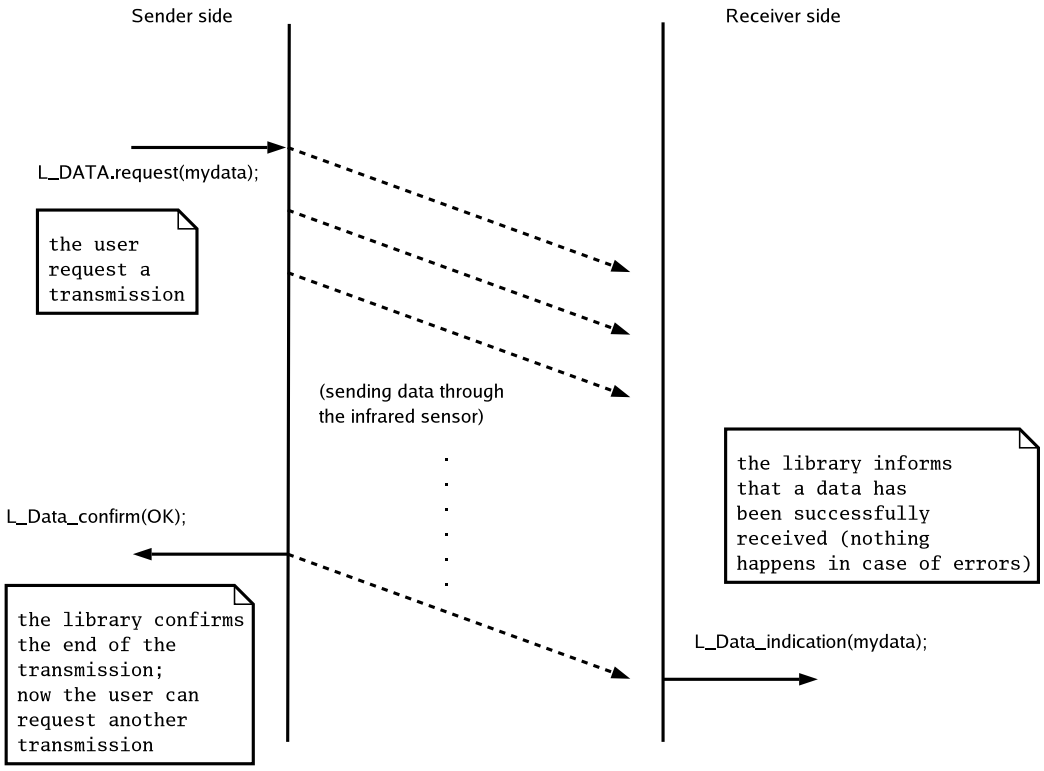


Figure 7.1 The IR communication protocol

- transmission ends only when RCX receives and ACK or timeout is reached;
- reception ends only after transmission of an ACK.

It's possibile to send 0 bytes. If ACK support is requested, 0-bytes message is treated as an ACK, else nothing is sent and the interrupt of transmission end is raised immediatly.

IR transmission is accomplished by using channel 1 of the 8 Bit Timer.

There are two ways of using IR support:

- if you want just to receive IR messages, you may add `__IR_RX_SUPPORT__`. In this case, among the following listed functions you won't be able to use the L_Data.request(). Besides, when you'll invoke the result function, you shall pass her just one argument, that is the name of the indication callback.
- if you need both the receiving and transmitting capabilities you have to add the `__IR_SUPPORT__`, which enables 8 Bit timer-Channel 1 to generate the correct square wave.

Note that you won't be able to use `__IR_SUPPORT__`, `__PLAY_NOTE_SUPPORT__` and `__ERIKA_PWM_USED__` at the same moment .

See also PWM Motors' section, Sound section

```
ERIKA_IR_result(void (*confirm)(ERIKA_UINT8 msg_status),
void (*indication)(ERIKA_UINT8 len, ERIKA_UINT8 *data));
```

Description By invoking this function, you have to indicate which functions shall be used to manage data sending and receiving. When a message is sent, completion of this operation is signaled by an interrupt which invokes the *confirm* function: then the application will be sure of the transmission. Besides, on the receiving side, when a new message is available, a consequent raised interrupt invokes the *indication* callback in which message received is stored into a buffer addressed by *data* pointer. Each execution of the *indication* callback overwrites the buffer, so it's safer to copy the buffer before exiting the callback. An application that uses IR communication shall always provide the *indication* callback. If the application uses not only IR reception

but also IR transmission, it shall provide also the *confirm* callback.Remember that, due to problems with ACK, after interrupts are raised, reception is alwais disabled, also if it was enabled before, so if you want to receive you should re-enable it at the end of the callbacks.

```
ERIKA_IR_start(ERIKA_UINT16 tx_timeout);
```

Description This function initializes the IR communication. *tx_ timeout* is the time to wait in transmission if channel is busy. If *tx_ timeout* is 0, transmission ends immediatly if channel is busy; if *tx_ timeout* is 0xFFFF, transmission waits until channel becomes free. In any other case, transmission waits until channel becomes free or *tx_ timeout* is reached.

```
ERIKA_IR_receive_enable();
```

Description It enables the RCX to receive data through the IR Interface. Remember that, due to problems with ACK, after interrupts are raised, reception is alwais disabled, also if it was enabled before, so if you want to receive you should re-enable it at the end of the callbacks.

```
ERIKA_IR_receive_disable();
```

Description It disables data reception through the IR interface.

```
L_Data.Request(ERIKA_UINT8 len, ERIKA_UINT8 *data);
```

```
ERIKA_IR_send(ERIKA_UINT8 len, ERIKA_UINT8 *data);
```

Description It sends *len* bytes contained in the buffer addressed by *data* pointer. Note that, at the moment, it's not possible to specify a destination for the message which is simply put *on air*. Note also that if you try to send a message while the previous one is still in transmission, new message starts being transmitted and the older one is lost.

7.1.8 Wireless (RF) Communication

As it always happens when IR light is involved, IR communication is severely affected by several problems:

- the communication end points must be in a very short range (for Lego RCXs this limit is about 10 meters);
- it must exist a direct line of view among the end points;
- finally, only a point-to-point connection is possible

These problems have been addressed and solved designing a Wireless RF Gateway and a Token Passing MAC Protocol which allow the communication to happen among up to 16 units at the same time. These units can communicate at far higher distances and also with obstacles among them.

From a RCX's point of view, it's almost unaware about the presence of a one more communication layer: it will just have to initialize by means of an IR Init message. This message must include:

- header composed by 0x55, 0xFF and 0x00;
- the type of the message ($0xf5$, multibyte message);
- length of the IR payload ($0x03$);
- **ADD**: address number of the unit;
- **MADD**: maximum address number;
- **TCONST**: timeout constant calculated as $(0x40/(MADD + 1)) * ADD$;
- **CHK**: checksum of the packet, calculated as $0xf5 + 0x03 + ADD + MADD + TCONST$.

Definitive IR message is shown:

$$[0x55\,0xff\,0x00\,0xf5\,0x0a\,0x03\,0fc\,ADD\;\;ADD\,MADD\;\;MADD$$

$$TCONST\;\;TCONST\,CHK\;\;CHK]$$

After this message has been sent, the Token Ring-like protocol starts working and every message the RCXs have to send will have to include the address of the destination as the first byte of the message payload.

8 Libraries

THIS CHAPTER shortly describes some additional library provided by the ERIKA Educational kernel. Currently, the projects supports Cyclical Asynchronous Buffers (CABs) and blocking semaphores besides the basic Kernel services.

8.1 Cyclical Asynchronous Buffers (CABs)

Cyclical Asynchronous Buffers (or CABs) represent a particular mechanism purposely designed for the cooperation among periodic activities with different activation rates. See [2] 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 thread 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 semantic, 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 threads are interested only in fresh sensory data rather than in the complete message history produced by a sensory acquisition thread.

Notice that more threads can simultaneously access the same buffer in a CAB for reading. Also, if a thread P reserves a CAB for writing while another thread 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 CABs initialization. To avoid blocking, this number must be equal to the number of threads that use the CAB plus one.

8.1.1 Compiling

To compile the CABS you should add `__CABS__` in the `ERIKAOPT` variable of the makefile.

8.1.2 CAB Types

Some CAB types have to be defined. These types are:

`ERIKA_CAB_INDEX` An integer type that is used to index a single buffer in the CAB;

`ERIKA_CAB_DATA` A type used for storing data; usually it is a `ERIKA_WORD`.

8.1.3 CAB Initialization

To limit the overall usage of RAM memory, the CAB descriptor has been divided in two parts:

ROM descriptor It contains the minimum set of references to the RAM data, plus the informations on the dimension and the number of buffers

RAM descriptor It is composed by a set of arrays that contains the data to be exchanged, and of a few other variables to store the CAB status.

Every CAB buffer must be defined. To define a CAB, use the macro `ERIKA_DEFINE_CAB`. That macro can be used to declare all the data structures needed. Note that after defining it, the initialization must be finished calling `ERIKA_cab_init()`.

Note At the end of the development process, the macro can be substituted with its preprocessor expansion, allowing a proper separation between ROM and RAM parts; this way of initializing a `CAB` do not need the call to `ERIKA_cab_init()`, since all the data structures can be initialized inline.

8.1.4 Using CABs

To insert a message in a CAB, a thread 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:

```
ERIKA_cab_reserve(&cab_struct, &buf_pointer, &msg_num);
... /* copy message in *buf_pointer */
ERIKA_cab_putmes(&cab_struct, msg_num);
```

Similarly, to get a message from a CAB, a thread 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:

```
ERIKA_cab_getmes(&cab_struct, &mes_pointer, &msg_num);
... /* use message contained into *mes_pointer */
ERIKA_cab_unget(&cab_struct, msg_num);
```

Currently, CABs can not be used into interrupt handlers.

8.1.5 The functions

`ERIKA_DEFINE_CAB(cab_name, msg_num, msg_size)`

Description This macro can be used to define a CAB. `cab_name` is the name of the CAB, whereas `msg_num` and `msg_size` are respectively the number of buffers allocated for the CAB and the number of `ERIKA_CAB_DATA` items that composes each message. Please note that CABs must always be defined as *global* variables.

The macro defines a set of data structures that can be allocated either in RAM or ROM. Moreover, it defines a

```
struct ERIKA_cab_ROM_desc cab_name;
```

in a way that the name `cab_name` can be used as parameter for the other CAB functions.

`ERIKA_EXTERN_CAB(cab_name)`

Description This macro can be used to declare a CAB that is defined on another file. Please note that CABs must always be defined as *global* variables.

```
void ERIKA_cab_init(struct ERIKA_cab_ROM_desc *c)
```

Description This macro must be used to initialize a CAB that was previously defined using `ERIKA_DEFINE_CAB`. This function is not used if a static initialization of the system is made.

```
void ERIKA_cab_reserve(
struct ERIKA_cab_ROM_desc *c, void **msg,
ERIKA_CAB_INDEX *msg_num)
```

Description This function reserves a buffer belonging to the `c` CAB and returns a pointer to the allocated message into `msg`, and its index into `msg_num`. The primitive has to be used only by writers and *never* by readers.

```
void ERIKA_cab_putmes(
struct ERIKA_cab_ROM_desc *c,
ERIKA_CAB_INDEX msg_num)
```

Description This function inserts the message whose number is by `msg_num` into the CAB identified by `c`. This primitive must be used *only* by writing threads.

```
void ERIKA_cab_getmes(
struct ERIKA_cab_ROM_desc *c, void **msg,
ERIKA_CAB_INDEX *msg_num)
```

Description This function returns a pointer (`msg`) to the latest message written into the `c` CAB, and its message number (`msg_num`). This primitive must be used *only* by reading threads.

```
void ERIKA_cab_unget(struct ERIKA_cab_ROM_desc *c,
ERIKA_CAB_INDEX msg_num)
```

Description This function notifies the system that the buffer whose index is `msg_num` belonging to the `c` CAB is no longer used by the calling thread.

8.1.6 An example

An example that fully utilize all the CABs functions can be found under the directory `tests/cabs`.

8.2 Semaphores

The primitives described in this section covers the semaphore mechanism interface that can be used by the ERIKA Educational applications. These primitives can be used both for synchronization and mutual exclusion. It is worth nothing that the traditional semaphore mechanism can cause unbounded *priority inversion*, so it is not suitable for hard real-time. There is no limit on the number of semaphores the application can create.

> The semaphore primitives can *only* be used with a multistack HAL, because these primitives are *blocking*. Currently, they cannot be used into an interrupt handler.

To use the semaphores, the user must define the `ERIKA_TYPESEMCOUNT` type (it is used to store the semaphore counter and typically it is a `ERIKA_WORD`).

8.2.1 Semaphore Initialization

Semaphore can be done either by inline initialization or using a macro.

This is an example of inline initialization of a semaphore:

```
ERIKA_SEM sem1 = {0,ERIKA_NIL,ERIKA_NIL};
```

and here is another piece of code for initializing a semaphore using a macro:

```
/* global definition */
ERIKA_SEM sem2;
...
void dummy() {
  ...
  ERIKA_sem_init(sem2,0);
  ...
}
```

In the examples showed before, both `sem1` and `sem2` were initialized with an intitial value of '`0`'.

8.2.2 Another example

The example describes a thread that uses a semaphore '`s`' to access a critical section in mutual exclusion.

```
ERIKA_SEM s = {1,ERIKA_NIL,ERIKA_NIL};
void thread_example(void)
{
  ...
  /* The task enters a critical section protected
     by a mutex semaphore */
  ERIKA_sem_wait(&s);
  <critical section>
  ERIKA_sem_post(&s);
  ...
}
```

8.2.3 The functions

```
void ERIKA_sem_wait(ERIKA_SEM *sem)
```

```
void ERIKA_sem_trywait(ERIKA_SEM *s, int *result)
```

Description `ERIKA_sem_wait` is used to decrement the value of the semaphore referenced by `sem`. If the semaphore value is currently zero, then the calling task does not return from the call to `ERIKA_sem_wait()` until it has been incremented by another thread using `ERIKA_sem_signal()`. `ERIKA_sem_trywait` decrements the semaphore value only if that value is greater than '`0`'. Otherwise, it does not lock the calling thread. `result` is set to '`1`' if the semaphore counter has been successfully decremented, '`0`' otherwise.

```
void ERIKA_sem_post(ERIKA_SEM *sem)
```

Description It increments the value of the semaphore referenced by `sem`. If the semaphore queue is not empty, the first task in the queue is put again into the ready queue and the scheduler is invoked: for that reason this function can cause a preemption. The semaphore queue is a FIFO queue: tasks are awoken in a FIFO order according to resource availability.

Type	Range	Granularity
Unsigned Fixed	$0.0 \div (2^{(16-N)} - 1).((2^N - 1)/2^N)$	2^{-N}
Signed Fixed	$-(2^{16-N-1}).0 \div (2^{16-N-1} - 1).((2^N - 1)/2^N)$	2^{-N}
Double Fixed	$-32768.0 \div 32767.999984$	0.00001526

Table 8.1 Range and granularity for fixed point types

```
void ERIKA_sem_getvalue(ERIKA_SEM *sem, ERIKA_TYPESEMCOUNT *value)
```

Description `ERIKA_sem_getvalue()` updates the location referenced by the `value` 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 `ERIKA_sem_getvalue()` is '`0`'.

8.3 H8 fixed point math

H8/300 does not support any floating point unit, so this mathematical library was created to supply this lack.

There are three numeric types:

- Unsigned fixed: 16 bit.
- Signed fixed: 16 bit.
- Double fixed: 32 bit.

For 16 bit types, you can choose how many bits you use for fractional part by defining these symbols in the file **fmath.s** which should be present in your application directory:

```
.equ ERIKA_uf_num_bit_frac, N (for unsigned fixed)
.equ ERIKA_sf_num_bit_frac, N (for signed fixed)
```

For double fixed, 16 bit are always used for fractional part.

So for each type we have this ranges:

8.3.1 Unsigned fixed

```
ERIKA_UF_TYPE ERIKA_uint2uf (ERIKA_UINT16 u)
```

Description Convert an unsigned, u, to an unsigned fixed and return it.

```
ERIKA_UNIT16 ERIKA_uf2unit(ERIKA_UF_TYPE uf)
```

Description Convert an unsigned fixed to an unsigned integer and return it.

```
ERIKA_UF_TYPE ERIKA_uf_mul(ERIKA_UF_TYPE a, ERIKA_UF_TYPE b)
```

Description Calculate a * b and return the result as unsigned fixed.

```
ERIKA_UF_TYPE ERIKA_uf_div(ERIKA_UF_TYPE a, ERIKA_UF_TYPE b)
```

Description Calculate a / b and return the result as unsigned fixed.

Note

Note that to calculate sum and subtraction you can use the standard operator + and -.

8.3.2 Signed fixed

```
ERIKA_SF_TYPE ERIKA_int2sf (ERIKA_INT16 i)
```

Description Convert an integer, i, to a signed fixed and return it.

```
ERIKA_INT16 ERIKA_sf2int(ERIKA_SF_TYPE sf)
```

Description Convert a signed fixed to an integer and return it.

```
ERIKA_SF_TYPE ERIKA_sf_mul(ERIKA_SF_TYPE a, ERIKA_SF_TYPE b)
```

Description Calculate a * b and return the result as signed fixed.

```
ERIKA_SF_TYPE ERIKA_sf_div(ERIKA_SF_TYPE a, ERIKA_SF_TYPE b)
```

Description Calculate a / b and return the result as signed fixed.

Note

Note that to calculate sum and subtraction you can use the standard operator + and -.

```
ERIKA_SF_TYPE ERIKA_sf_cos(ERIKA_UINT16 theta)
```

Description Calculate cosine of theta and return it as signed fixed.

```
ERIKA_SF_TYPE ERIKA_sf_sin(ERIKA_UINT16 theta)
```

Description Calculate sine of theta and return it as signed fixed.

Note

Note that the argument of trigonometrical functions is expressed in degree and it is normally executed the modulo 360 operation on the argument. So if you pass 370 to the function, it will be equals to pass 10.

It is possible to avoid this operation, to reduce execution time, but you have to be sure that argument is between 0 and 359, otherwise wrong result will be produced .

8.3.3 Double fixed

Functions usable by the applicaton:

```
void ERIKA_int2df(ERIKA_DF_TYPE *res, ERIKA_INT16 i)
```

Description Convert an integer i to a double fixed and result is stored at the *res* address.

```
ERIKA_INT16 ERIKA_df2int(ERIKA_DF_TYPE *f)
```

Description Convert the double fixed whose address is f to an integer and return it.

Condition	Returned value
a<b	-1
a==b	0
a>b	1

Table 8.2 Compare double fixed and returned value

```
Evoid ERIKA_df_negate(ERIKA_DF_TYPE *res, ERIKA_DF_TYPE *f)
```

Description Negate the double fixed whose address is *f* and store the result in the double fixed whose address is *res*.

```
ERIKA_INT16 ERIKA_df_cmp(ERIKA_DF_TYPE *a, ERIKA_DF_TYPE *b)
```

Description Compare two double fixed whose addresses are *a* and *b* and return a value how you can see in Table 8.2.

```
void ERIKA_df_add(ERIKA_DF_TYPE *res, ERIKA_DF_TYPE *a, ERIKA_DF_TYPE *b)
```

Description Add two double fixed and store result as the address *res* as double fixed.

```
void ERIKA_df_sub(ERIKA_DF_TYPE *res, ERIKA_DF_TYPE *a, ERIKA_DF_TYPE *b)
```

Description Calculate a minus b and store result into the double fixed whose address is *res*.

```
void ERIKA_df_mul(ERIKA_DF_TYPE *res, ERIKA_DF_TYPE* a, ERIKA_DF_TYPE *b)
```

Description Calculate a * b and store the result into the double fixed whose address is *res*.

```
void ERIKA_df_div(ERIKA_DF_TYPE *res, ERIKA_DF_TYPE *a, ERIKA_DF_TYPE *b)
```

Description Calculate a / b and store the result into the double fixed whose address is *res*.

```
void ERIKA_df_cos(ERIKA_DF_TYPE *res, ERIKA_DF_TYPE theta)
```

Description Calculate cosine of theta and store the result into the double fixed whose address is *res*.

```
void ERIKA_df_sin(ERIKA_DF_TYPE *res, ERIKA_DF_TYPE theta)
```

Description Calculate sine of theta and store the result into the double fixed whose address is *res*.

> Calculate cosine of theta and store the result into the double fixed whose address is *res*.

Note that the angle is expressed in degree and so it is expected to be between 0 and 359. See what said above for signed fixed.

II

Kernel Internals

9 HAL internals

THIS CHAPTER explains the internal structure of the kernel, and its internal functions. Moreover, the HAL interface is presented and it is explained how the kernel uses it. This chapter is not just a reference for kernel programmers, but it is also useful to understand how data structures have to be initialized.

9.1 Kernel code portability

One of the main design issue in the design of the Kernel Layer is the *code portability*, in the sense of the ability of reusing all the Kernel Layer source code in more than one architecture. For that reason, all the code of the Kernel Layer is standard C code, whereas all the architecture dependent issues are hided under the HAL Layer interface.

Every primitive of the Kernel Layer is coded by different kernels in different ways to implement the various scheduling algorithms. In practice, each kernel adds a prefix to the name of each primitive to say that it is *its* implementation. For example, the FP Kernel adds the prefix '`fp_`', so the primitive called `ERIKA_thread_activate()` can be found into `ERIKABASE/src/fp/thact.c` as `ERIKA_fp_thread_activate()`.

Note that applications never call directly `ERIKA_fp_thread_activate()`. They only call a redefinition that is obviously named `ERIKA_thread_activate()` that will then call, for example, `ERIKA_fp_thread_activate()`.

There are two reasons because a user can not call directly the implementation provided by a Kernel:

- Every architecture has its own methods for calling a primitive, and a Kernel implementation should only implement the *behavior* of a par-

ticular primitive, without relying on nothing else than the standard C HAL interface. For example, some architectures can implement a function call using a normal function call, whereas other architectures could use software interrupts to cope with MMU and memory protection issues.

- Some CPU architectures allow an efficient implementation of some primitives. The separation between the function the user calls and its implementation in the Kernel ease this kind of optimized implementation.

To ease the implementation of optimized versions of the kernel primitives, the source code contains some conditional defines that allow to remove the implementation independent definitions from the final objects. Table 9.1 on the next page contains that list: when such a `#define` is defined, the correspondent primitive is not compiled.

9.2 Thread queues

To track the evolution of the thread state, the Kernel handles some queue data structures, that groups all the threads that have the same thread state. These functions are not called directly by the user, but are instead called inside the implementation of each primitive.

Typical examples of thread queues are:

Ready queue This queue is used to contain all the ready thread in the system (or all the thread that are assigned to a CPU in a multiprocessor system). It is usually ordered by priority.

Stacked queue This queue is used to store the informations about the contexts that are currently allocated on each stack. In general, the first thread in that queue is the running thread, while the others are the preempted thread. The order in the queue reflects the real state on nesting of the contexts in the system.

Blocked queue This queue contains all the tasks that are blocked on a particular synchronization primitive. Note that this queue is not present in the kernels that allow stack sharing via the Stack Resource Policy (allowing the use of the monostack HAL), since there are not any kind of blocking primitive under the SRP.

Table 9.1 Symbols that exclude the compilation of the generic kernel primitives.

Symbol	*Primitive*
__PRIVATE_SYS_START__	ERIKA_sys_start()
__PRIVATE_SYS_PANIC__	ERIKA_sys_panic()
__PRIVATE_SYS_REBOOT__	ERIKA_sys_reboot()
__PRIVATE_SYS_IDLE__	ERIKA_sys_idle()
__PRIVATE_SYS_SCHEDULER__	ERIKA_sys_scheduler()
__PRIVATE_SYS_GETTIME__	ERIKA_sys_gettime()
__PRIVATE_THREAD_MAKE_READY__	ERIKA_thread_make_ready()
__PRIVATE_THREAD_ACTIVATE__	ERIKA_thread_activate()
__PRIVATE_THREAD_END_INSTANCE__	ERIKA_thread_end_instance()
__PRIVATE_THREAD_ENABLE__	ERIKA_thread_enable()
__PRIVATE_THREAD_DISABLE__	ERIKA_thread_disable()
__PRIVATE_THREAD_ISENABLED__	ERIKA_thread_isenabled()
__PRIVATE_MUTEX_LOCK__	ERIKA_mutex_lock()
__PRIVATE_MUTEX_UNLOCK__	ERIKA_mutex_unlock()
__PRIVATE_IRQ_MAKE_READY__	ERIKA_IRQ_make_ready()
__PRIVATE_IRQ_END_INSTANCE__	ERIKA_IRQ_end_instance()
__PRIVATE_SET_IRQ_HANDLER__	ERIKA_set_IRQ_handler()
__PRIVATE_RQ_QUERYFIRST__	ERIKA_rq_queryfirst()
__PRIVATE_STK_QUERYFIRST__	ERIKA_stk_queryfirst()
__PRIVATE_STK_GETFIRST__	ERIKA_stk_getfirst()
__PRIVATE_STK_INSERTFIRST__	ERIKA_stk_insertfirst()
__PRIVATE_RQ_INSERT__	ERIKA_rq_insert()
__PRIVATE_RQ2STK_EXCHANGE__	ERIKA_rq2stk_exchange()

Table 9.2 Symbols that exclude the compilation of the CAB primitives.

Symbol	*Primitive*
__PRIVATE_CAB_INIT__	ERIKA_DEFINE_CAB(), ERIKA_EXTERN_CAB(), ERIKA_cab_init()
__PRIVATE_CAB_RESERVE__	ERIKA_cab_reserve()
__PRIVATE_CAB_PUTMES__	ERIKA_cab_putmes()
__PRIVATE_CAB_GETMES__	ERIKA_cab_getmes()
__PRIVATE_CAB_UNGET__	ERIKA_cab_unget()

Table 9.3 Symbols that exclude the compilation of the semaphore primitives.

Symbol	*Primitive*
__PRIVATE_SEM_INIT__	ERIKA_sem_init()
__PRIVATE_SEM_WAIT__	ERIKA_sem_wait()
__PRIVATE_SEM_TRYWAIT__	ERIKA_sem_trywait()
__PRIVATE_SEM_POST__	ERIKA_sem_post()
__PRIVATE_SEM_GETVALUE__	ERIKA_sem_getvalue()

These queues are used by the following functions:

ERIKA_rq_queryfirst() It returns the ERIKA_TID of the first thread in the ready queue.

ERIKA_stk_queryfirst() It returns the ERIKA_TID of the first thread in the ready queue, that typically is the running thread.

ERIKA_stk_getfirst() It extracts the thread that is on the top of the ready queue.

ERIKA_stk_insertfirst() It inserts a thread on the top of the stacked queue.

ERIKA_rq_insert() It inserts a task into the ready queue, keeping the order imposed by the scheduling algorithm (for example, the threads are ordered by priority or by absolute deadline).

ERIKA_rq2stk_exchange() This function is used to activate a ready thread. It simply extracts the first thread from the ready queue and then it inserts it at the top of the stacked queue. The function returns the ERIKA_TID of the thread that has been moved.

9.3 Priority handling

The FP Kernel implements a particular priority encoding; in practice, a priority can only be coded as a power of 2. The priority binary representation

is a ERIKA_WORD with all the bits set to '0' but one that is set to '1'. Bigger numbers define higher priorities.

If we consider a ERIKA_WORD of 32 bits, both 0x0001 0000 and 0x0000 0800 are valid priority encodings, whereas 0x0000 3000 not.

The rationale of this choice is that this priority encoding ease the implementation of the SRP protocol. In particular, in that way the system variable ERIKA_sys_ceiling can store the maximum ceiling in the system and all its previous values simply using an OR instruction. Note that if we assign the MSB to the highest priority and so on, the checks on the system ceilings can be easily done using the '>' operator (only the MSB is significative in the comparison). In fact note that a task can make preemption only if its priority (or its preemption level) is strictly greater than the system ceiling.

An example of usage of this priority assignment can be found in Section 5.1.2 on page 37.

9.4 The HAL Interface

In Chapter 3 on page 9 we presented the HAL layer as a software layer that is used to abstract a particular implementation from the underlying hardware. In particular, the HAL interface offers the following services:

Thread abstraction A thread is identified by the HAL by a C function, and by the information that are stored in ROM/RAM, such as the stack space it occupies when it is stacked, the copy of the CPU registers if needed, and other implementation dependent informations.

Context handling The HAL implements a limited set of functions that can be used to implement context switching between tasks.

Interrupt handling The HAL implements all the interface needed to properly handle interrupts.

Utility functions Finally, some functions for time, idle time and reboot handling are provided.

All these services are not used directly by the application, but they are called through the services exported by the Kernel Layer. Table 9.4 on the following page summarizes the main services exported by the HAL.

Table 9.4 The basic HAL Interface

Context handling	`ERIKA_hal_endcycle_ready()`
	`ERIKA_hal_endcycle_stacked()`
	`ERIKA_hal_ready2stacked()`
	`ERIKA_hal_IRQ_ready()`
	`ERIKA_hal_IRQ_stacked()`
	`ERIKA_hal_stkchange()`
Primitive definition	`ERIKA_hal_begin_primitive()`
	`ERIKA_hal_end_primitive()`
	`ERIKA_hal_IRQ_begin_primitive()`
	`ERIKA_hal_IRQ_end_primitive()`
Interrupt handling	`ERIKA_hal_enableIRQ()`
	`ERIKA_hal_disableIRQ()`
	`ERIKA_hal_IRQ_enableIRQ()`
	`ERIKA_hal_IRQ_disableIRQ()`
Utility functions	`ERIKA_hal_gettime()`
	`ERIKA_hal_reboot()`
	`ERIKA_hal_panic()`
	`ERIKA_hal_idle()`

Table 9.5 Data types and constants defined by the HAL Layer

Constants	`ERIKA_NULL`	Invalid pointer value
Types	`ERIKA_TID`	Thread identifier
	`ERIKA_TIME`	Temporal reference
	`ERIKA_ADDR`	Memory address
	`ERIKA_WORD`	Word (architecture dependent)
	`ERIKA_BIT`	Single bit addressable value
	`ERIKA_INT8`	8 bit signed integer
	`ERIKA_UINT8`	8 bit unsigned integer
	`ERIKA_INT16`	16 bit signed integer
	`ERIKA_UINT16`	16 bit unsigned integer
	`ERIKA_INT32`	32 bit signed integer
	`ERIKA_UINT32`	32 bit unsigned integer

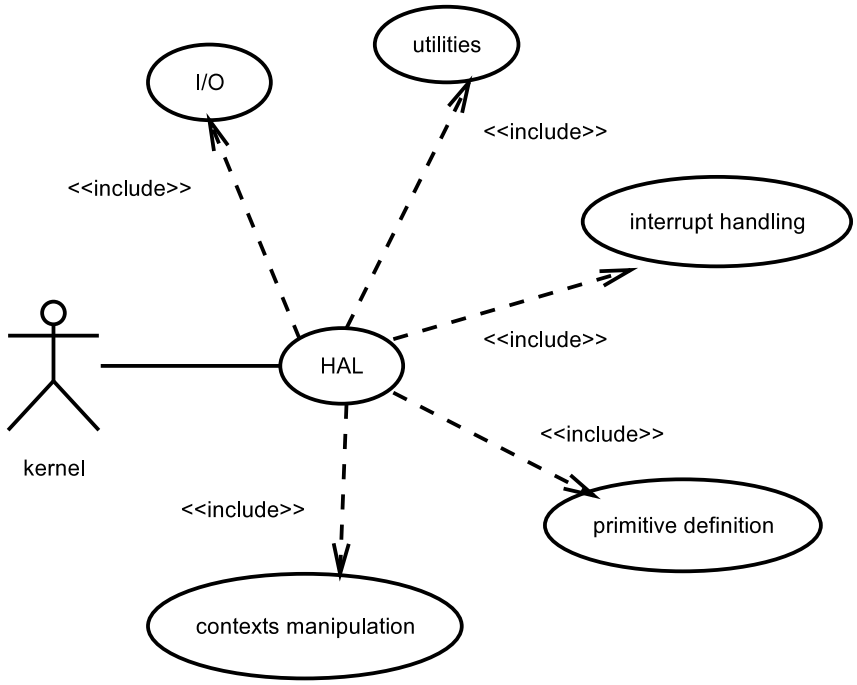


Figure 9.1 Services exported by the HAL layer.

The HAL interface is implemented by the HAL Layer. In general, the HAL layer can be developed following two different paradigms, that in the following paragraphs we highlighted as *monostack HAL* and *multistack HAL*.

Monostack HAL Following this model, all the threads share the same stack. This implicitly assume that a preempted thread can not execute again until all the threads and interrupts handlers that preempted it have finished.

Note Under this paradigm it is not possible to implement any kind of blocking primitive. Moreover, note that there are functions in the HAL that are used to wake up a preempted thread that is supposed at the top of a stack. As you will notice in the following sections, these functions need a parameter that is the `ERIKA_TID` that has to be woken up: since only one stack is supported, the thread to wake up can only be one. For this reason, the `ERIKA_TID` parameter is ignored.

Multistack HAL This model differs from the previous one because the HAL handles more than one stack. In particular, stack sharing is limited by assigning (at compile time) a set of thread to a particular stack.

As a limit, all the threads use a private stack, and no stack sharing is exploited at all. Note that the assignment of threads to stacks is usually done statically.

9.5 Interaction between HAL and Kernel

All the names of the functions exported by the HAL that are used for context handling follows the convention explained below:

- the word '`stacked`' is used to put in execution a preempted thread;
- the word '`ready`' is used to mean a thread that has been just extracted from the ready queue, and which has not any context allocated on its stack yet.

In the multistack HAL there is also the case of identifiers such as '`stkchange`': they refer to threads that have been preempted; in that case there will be certainly a stack change.

All the HAL implementations should guarantee a thread that has been interrupted will start again from the same point where it was interrupted.

As for the Kernel Layer, the application should define and initialize the HAL data structures. These structures heavily depend on the particular architecture, and are described in detail in Chapter **??** on page ?? and Chapter **??** on page ??. The initialization of these data structures can be statically done into the variable initializers or can be done into the `ERIKA_dummy()` function. The latter imposes some of the variables that could be stored in ROM (if the static initialization has been done) have to be stored in RAM due to their the on-line initialization.

The interrupt handlers, depending on the implementation, can execute on the same stack of the interrupted thread or in another stack.

The instruction needed to do the same thing can change whether a function is called into an interrupt handler or if it is called into an IRQ handler. For that reason some function has a double name, one that can be called into an interrupt and another that can be called into a thread. The identifiers with the '`IRQ_`' prefix identifies a primitive that can be used in the interrupt drivers.

All the HAL primitives that implements a context change suppose that the stack frame has a predefined form (that generally is imposed by C compilers). In particular this assumption can be done because only the HAL

functions manage contexts, and so they can be designed to coherently manage the stacks.

Finally, on a multistack kernel the HAL primitives that implement a context change do not always change the current stack.

9.6 Context Handling functions

```
void ERIKA_hal_endcycle_ready(ERIKA_TID t)
```

Description This function is called as the last instruction in primitives such as `ERIKA_thread_end_instance()`. It destroys the context allocated by the thread that is just ended, and creates the context of the thread '`t`' passed as parameter. Then the thread '`t`' is executed calling the thread body function.

Note that the thread passed as parameter is supposed to be removed from the ready queue (e.g. using `ERIKA_rq2stk_exchange()`, see Chapter 10 on page 97). This function can change the current stack if used with a multistack HAL.

See also `ERIKA_hal_IRQ_ready()`, `ERIKA_hal_ready2stacked()`, `ERIKA_hal_endcycle_stacked()`.

```
void ERIKA_hal_endcycle_stacked(ERIKA_TID t)
```

Description This function is called as the last instruction in primitives such as `ERIKA_thread_end_instance()`. It destroys the context allocated by the thread that is just ended, and then executes the thread '`t`' whose context is supposed to be on the top of a stack (that in a multistack HAL may not be the current stack).

The '`t`' parameter is used only with the multistack HALs because the HAL have to know on which stack the thread context is allocated. The '`t`' parameter is ignored by a monostack HAL, since it has only a single stack.

See also `ERIKA_hal_IRQ_stacked()`, `ERIKA_hal_ready2stacked()`, `ERIKA_hal_endcycle_ready()`.

`void ERIKA_hal_ready2stacked(ERIKA_TID t)`

Description This function is used to dispatch a ready task. For example it is called into a thread activation primitive or on a preemption check when the activated thread has to preempt the running thread. This function suspends the running thread (saving its context on its stack), then it creates the thread '`t`' on its stack, and finally it executes the body of the thread '`t`'.

If the HAL is multistack, this function can change the current stack.

See also `ERIKA_hal_endcycle_ready()`, `ERIKA_hal_IRQ_ready()`.

`void ERIKA_hal_IRQ_ready(ERIKA_TID t)`

Description It has the same meaning of `ERIKA_hal_endcycle_ready()`, but it is called only at the end of the last nested interrupt handler. It must be the last function call of the interrupt handler.

If the HAL is a multistack HAL, this function can change the current stack.

See also `ERIKA_hal_endcycle_ready()`, `ERIKA_hal_ready2stacked()`, `ERIKA_hal_IRQ_stacked()`.

`void ERIKA_hal_IRQ_stacked(ERIKA_TID t)`

Description It has the same meaning of `ERIKA_hal_IRQ_ready()`, but it is called only at the end of the last nested interrupt handler. It must be the last function call of the interrupt handler.

If the HAL is a multistack HAL, this function can change the current stack.

See also `ERIKA_hal_endcycle_stacked()`, `ERIKA_hal_ready2stacked()`, `ERIKA_hal_IRQ_ready()`.

`void ERIKA_hal_stkchange(ERIKA_TID t)`

Description This function is implemented *only* by the multistack HALs. It changes the running thread to another thread '`t`' that is currently on the top of *another* stack. This function always change the current context. It is a Kernel responsibility to ensure that '`t`' always indexes a thread that is on the top of the stack of another stack with respect to the running thread. If '`t`' is equal `ERIKA_NIL` then the `ERIKA_dummy` thread is considered.

This function is used by the Kernel to implement the synchronization points for the blocking primitives. In particular, a thread blocked on a synchronization primitive can be woken up by another thread. If a preemption has to be made, this function is called.

See also `ERIKA_hal_endcycle_ready()`, `ERIKA_hal_ready2stacked()`, `ERIKA_hal_IRQ_stacked()`.

9.7 Kernel Primitive Definition

The functions described in this section prepare the execution of a kernel primitive, and are composed by a prologue and an epilogue. This is needed because in general every architecture has a different way of handling the system calls (for example, some architectures simply disable/enable the interrupts in a kernel primitive, whereas other architectures use a software interrupt to bring the CPU in supervisor mode).

It is implementation defined whether the function `ERIKA_hal_end_primitive()` returns or whether it switches back to the thread that called the primitive. As a result, the C language '`return`' instruction cannot be used to return values to the caller. To return a value, a primitive should use a shared memory area specified passing a pointer to the kernel.

```
void ERIKA_hal_begin_primitive(void)
```

Description This function must be called as the *first* instruction of every primitive, to prepare the environment for the kernel.

See also `ERIKA_hal_end_primitive()`, `ERIKA_hal_IRQ_begin_primitive()`.

```
void ERIKA_hal_end_primitive(void)
```

Description This function must be called as the *last* instruction of every primitive, to restore the environment of the calling thread. Note that it is implementation defined if this function returns to the kernel primitive or to the calling thread.

See also `ERIKA_hal_begin_primitive()`, `ERIKA_hal_IRQ_end_primitive()`.

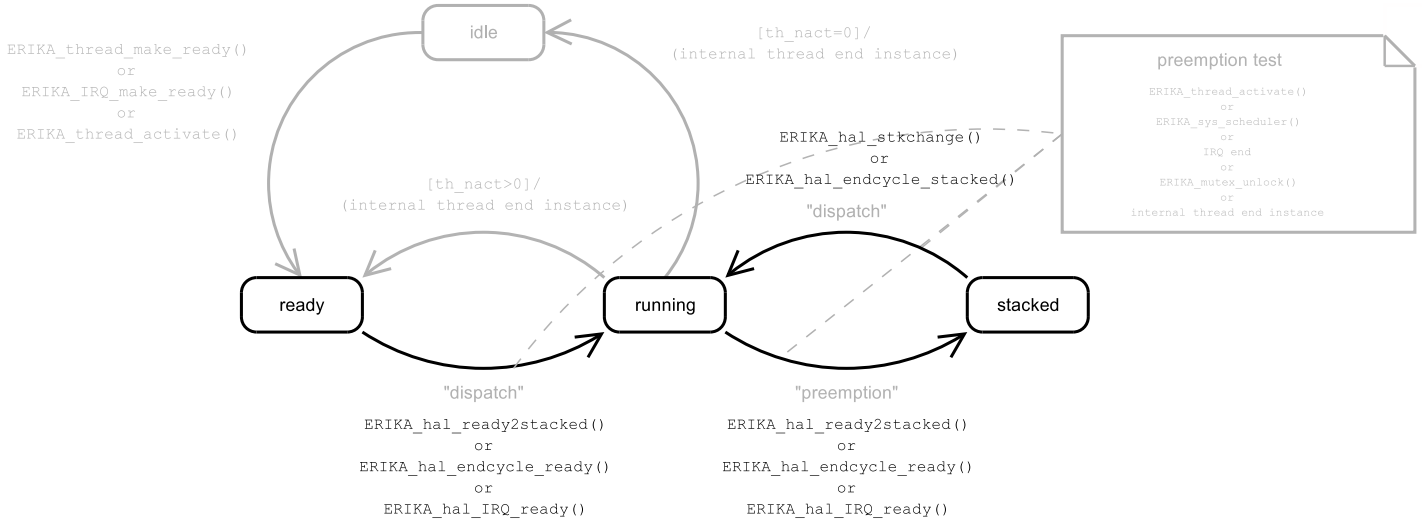


Figure 9.2 Kernel thread states and HAL functions that implement the transitions between states, without `ERIKA_BLOCKED` state. The gray part of the image is managed by the Kernel Layer (see Figure 4.2 on page 23). Note that when a thread moves from the stacked state to the running state, another thread ends (because `ERIKA_hal_endcycle_stacked()` is called by the `ERIKA_thread_end_instance()`) or blocks (only for Multistack HALs, see `ERIKA_hal_stkchange()`). When a thread moves from ready to running, the same primitive usually move the running thread from the running state to the stacked state.

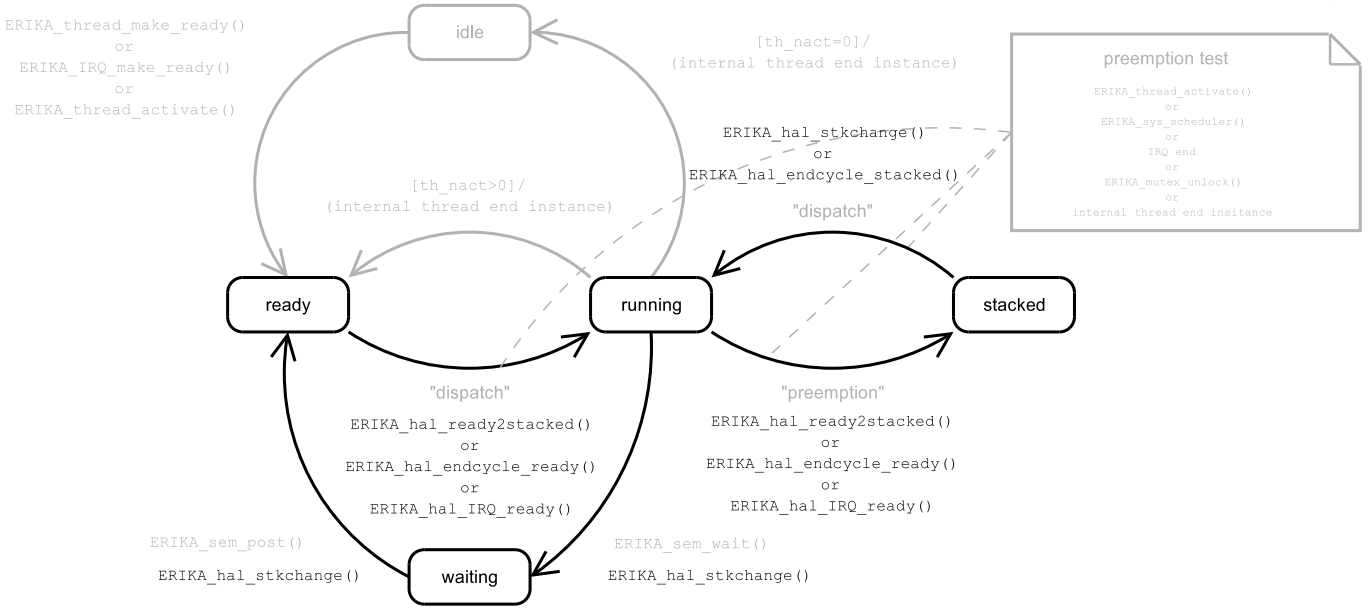
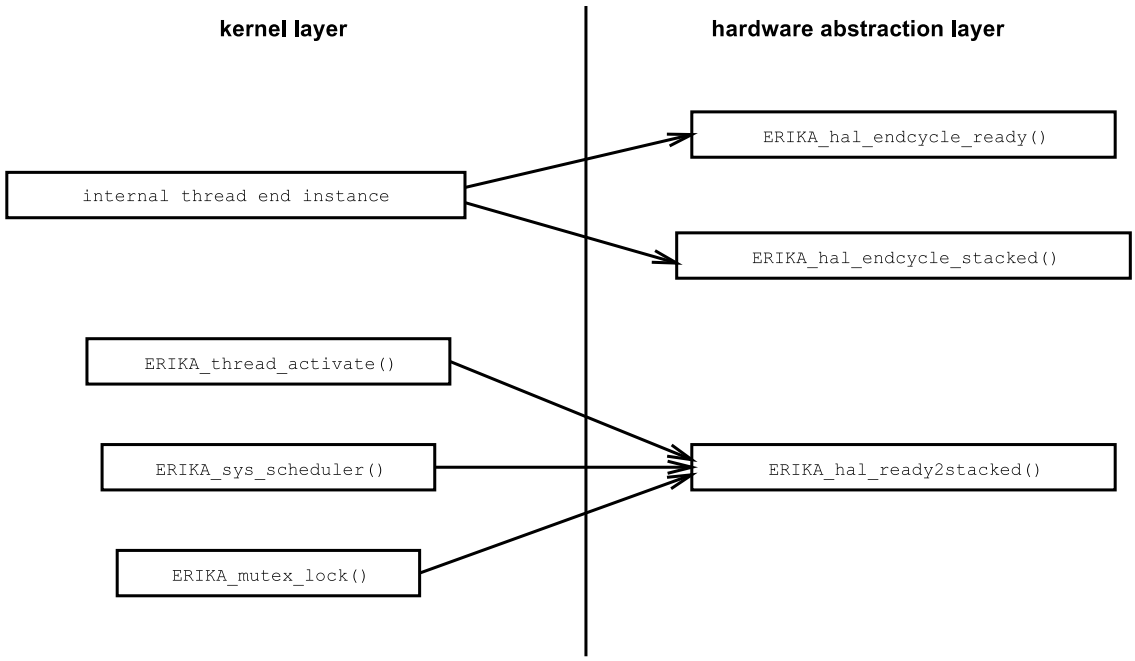
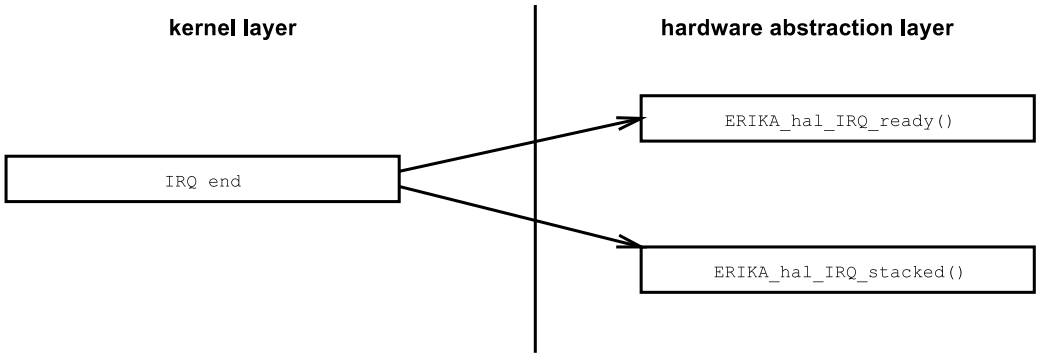


Figure 9.3 Kernel thread states and HAL functions that implement the transitions between states, with `ERIKA_BLOCKED` state. Please see Figure 9.2 on the preceding page.



(a) Primitive used into a thread



(b) Primitive used into a driver

Figure 9.4 Interaction between the kernel layer primitives and the HAL functions for context handling.

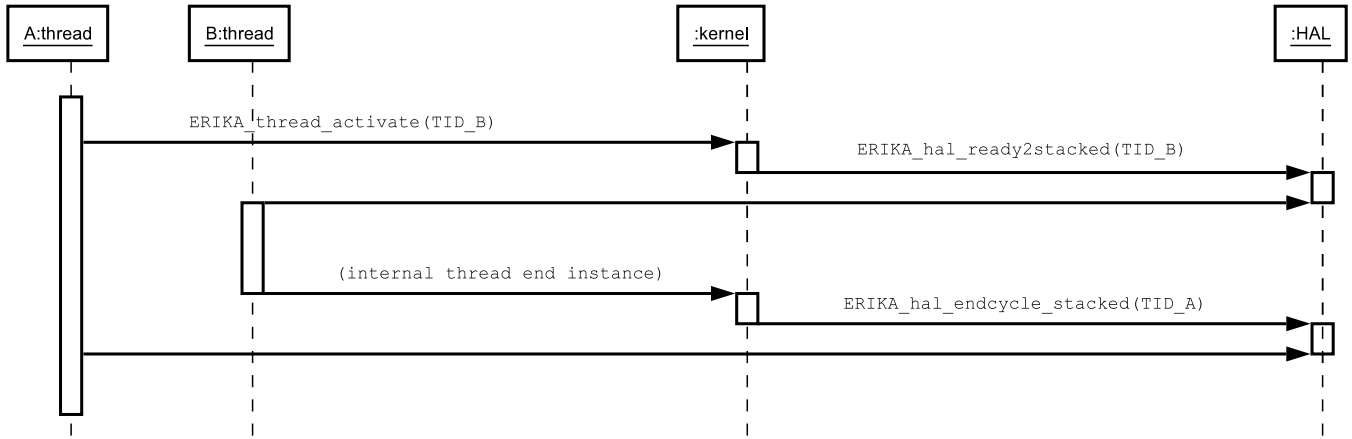


Figure 9.5 Context change due to the activation of a thread B at higher priority with respect to the running thread A.

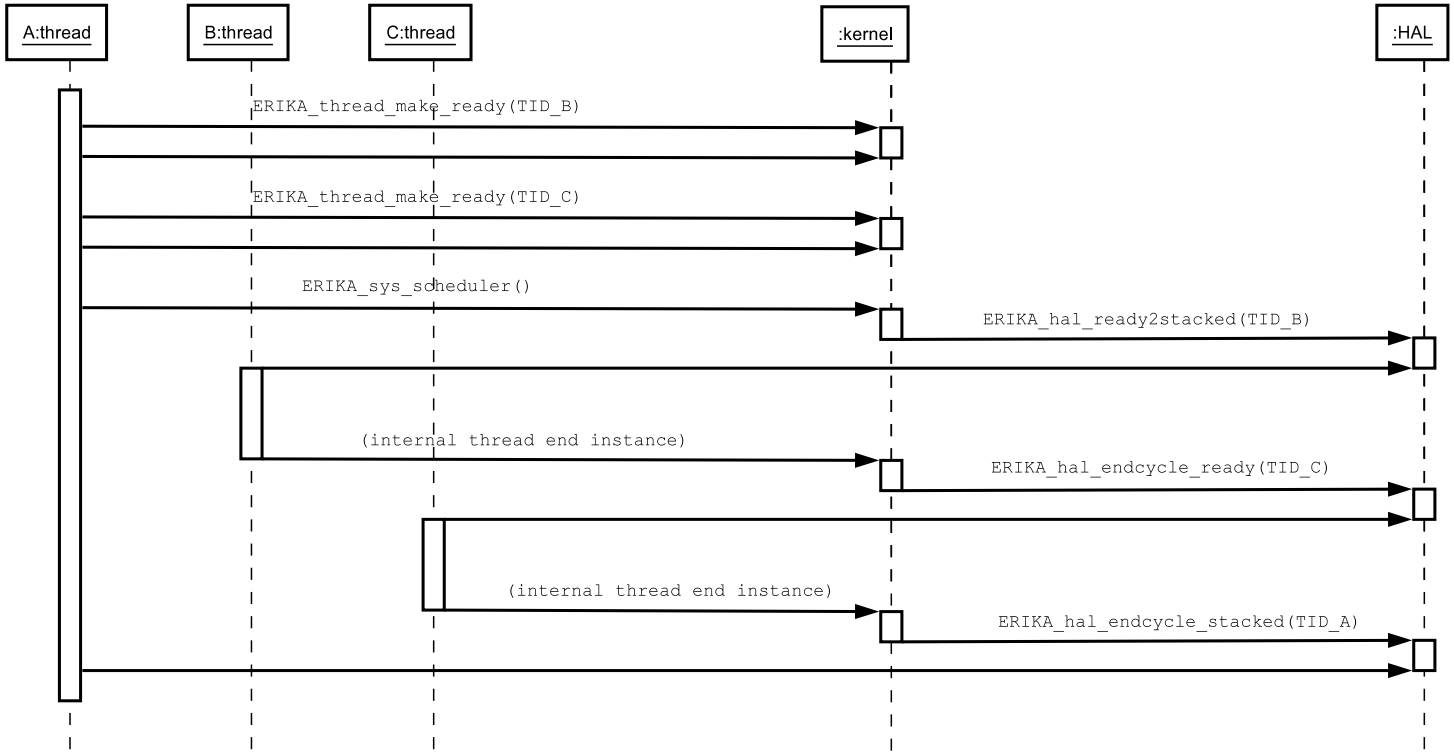


Figure 9.6 Context change caused by the activation of two higher priority threads with respect to the running thread.

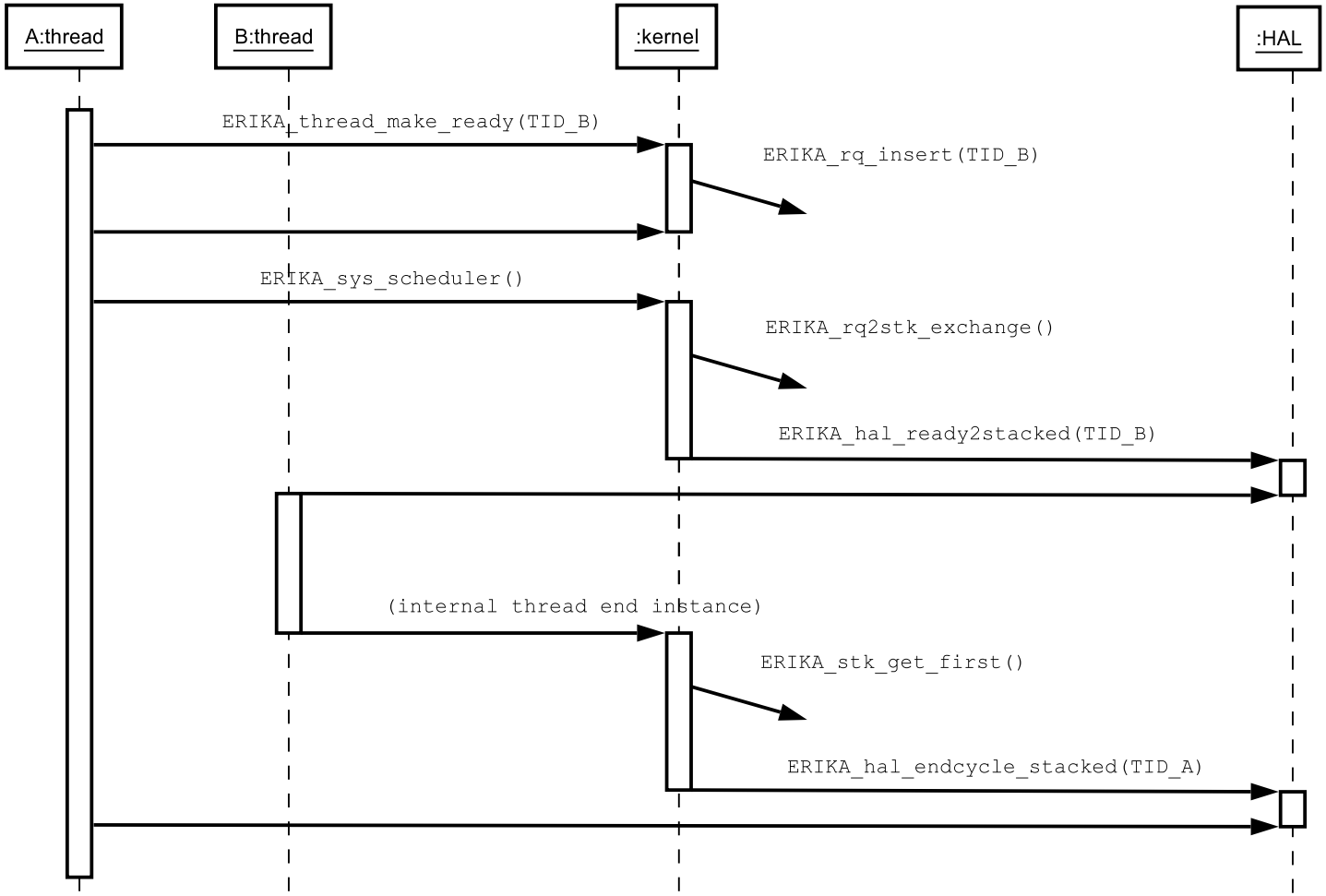


Figure 9.7 Context change due to the activation of an higher priority thread B with respect to the running thread A.

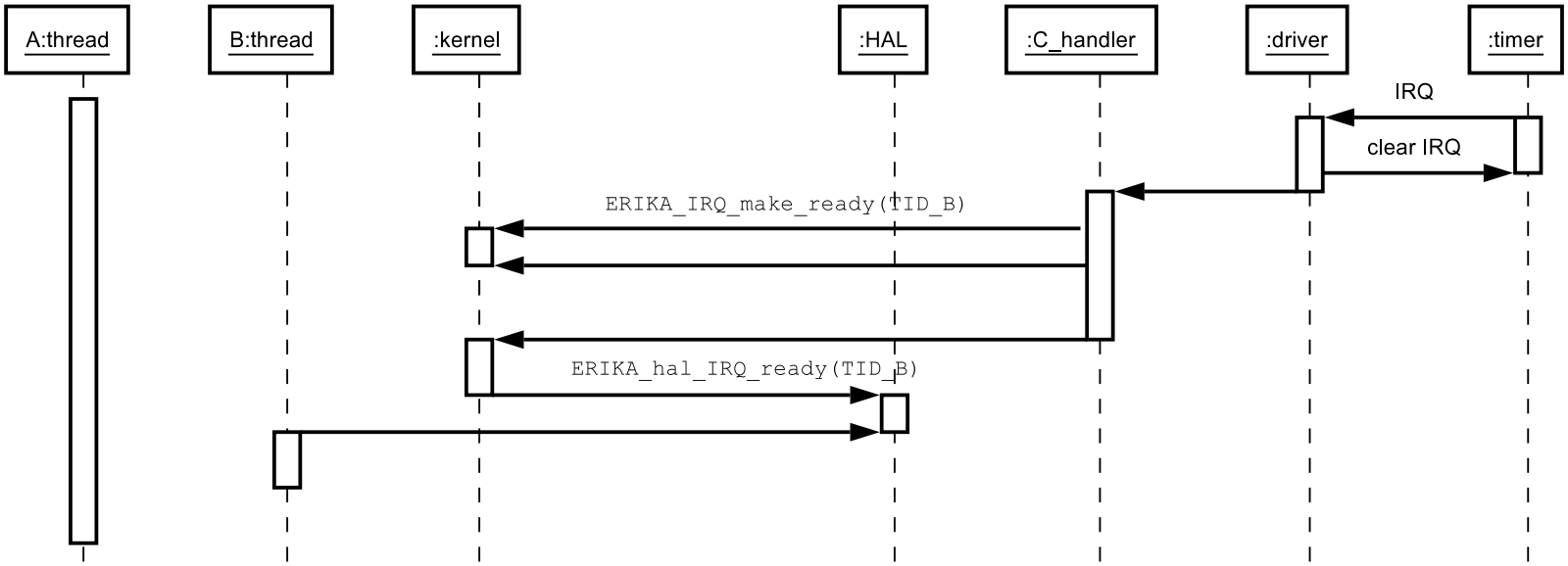


Figure 9.8 Context change due to the activation of an higher priority thread B with respect to the running thread A. The thread is activated by an interrupt handler (in the figure, a timer).

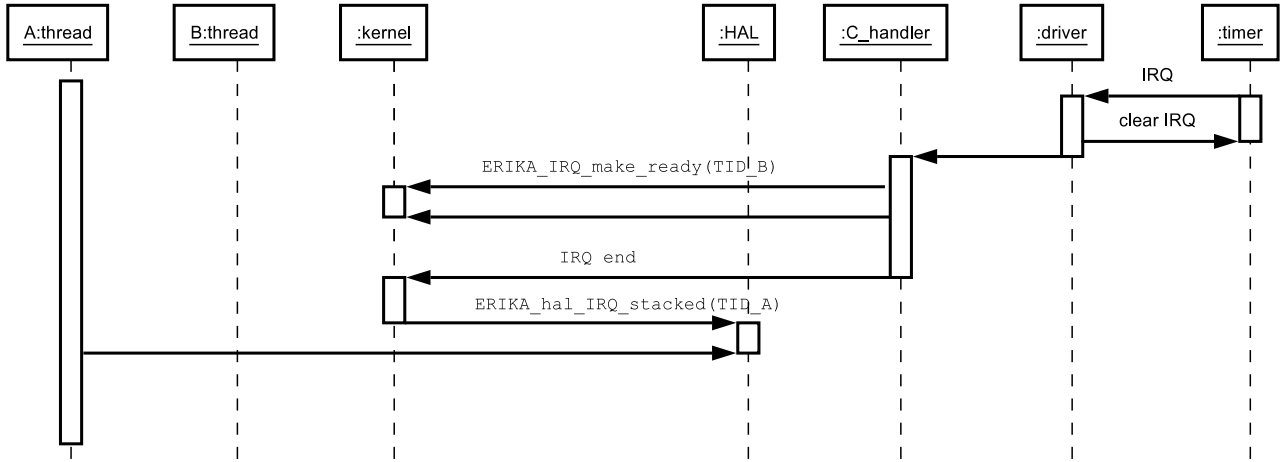


Figure 9.9 Lower priority thread activation without context switch The thread is activated by an interrupt handler (in the figure, a timer).

`void ERIKA_hal_IRQ_begin_primitive(void)`

Description This function must be called as the *first* instruction of every primitive used into an interrupt handler, to prepare the environment for the kernel.

See also `ERIKA_hal_IRQ_end_primitive()`, `ERIKA_hal_begin_primitive()`.

`void ERIKA_hal_IRQ_end_primitive(void)`

Description This function must be called as the *last* instruction of every primitive used into an interrupt handler, to restore the environment of the calling function. Note that it is implementation defined if this function returns to the kernel primitive or to the calling function.

See also `ERIKA_hal_IRQ_begin_primitive()`, `ERIKA_hal_end_primitive()`.

9.8 Interrupt handling

The functions described in this section are used to enable/disable interrupts into a thread[1] or into an interrupt handler. They are useful to implement short critical sections.

`void ERIKA_hal_enableIRQ(void)`

Description It enables all the interrupt sources. It can only be called into a thread.

See also `ERIKA_hal_disableIRQ()`, `ERIKA_hal_IRQ_enableIRQ()`.

`void ERIKA_hal_disableIRQ(void)`

Description It disables all the interrupt sources. It can only be called into a thread.

See also `ERIKA_hal_enableIRQ()`, `ERIKA_hal_IRQ_disableIRQ()`.

[1] This is the unique exception to the general rule that a thread can not call any function of the HAL.

`void ERIKA_hal_IRQenableIRQ(void)`

Description It enables all the interrupt sources. It can only be called into an interrupt driver.

Since some architectures can limit the number of interrupt drivers that can be nested, it is implementation defined if this function really disable the interrupts or not, depending on the fact that interrupts are enabled or disabled into a driver.

See also `ERIKA_hal_IRQ_disableIRQ()`, `ERIKA_hal_enableIRQ()`.

`void ERIKA_hal_IRQdisableIRQ(void)`

Description It disables all the interrupt sources. It can only be called into an interrupt drivers.

Since some architectures can limit the number of interrupt drivers that can be nested, it is implementation defined if this function really disable the interrupts or not, depending on the fact that interrupts are enabled or disabled into a driver.

See also `ERIKA_hal_IRQ_enableIRQ()`, `ERIKA_hal_disableIRQ()`.

9.9 Utility functions

We are going to describe some utility functions used for particular purpose.

`void ERIKA_hal_gettime(ERIKA_TIME *t)`

Description It returns the value of the system timer . Note that this function is only available if the user defines the `__TIME_SUPPORT__` symbol in the ERIKA Educational makefile.

`void ERIKA_hal_panic(void)`

Description It is called by `ERIKA_sys_panic()` to put the system to a meaningful state. It usually reset the system or it hangs the system signaling an abnormal termination to the external world.

`void ERIKA_hal_reboot(void)`

Description This function is called by `ERIKA_sys_reboot()` to reset the system.

`void ERIKA_hal_idle(void)`

Description This function is called by `ERIKA_sys_idle()` to put the CPU in an idle low power state. In that state, the CPU usually does nothing except waiting an interrupt. If the CPU does not support such a feature, the function simply does nothing.

10 Kernel Layer Internals

To be done!

11 H8 HALs Internals

The purpose of this short section is to show the most important features (in the ERIKA point of view) provide by the H8 processor. To have further details, please see the official Hitachi web site (http://semiconductor.hitachi.com/H8).

11.1 HALs general description

The ERIKA project provides two kinds of HALs for the H8 platform, that is a monostack HAL and a multistack HAL. Each of these have different requirements in terms of underlying memory models and in terms of memory usage. Moreover, each HAL provides different performances in terms of execution time of the primitives. Basically the main differences are:

Monostack HAL This is the simplest and fastest HAL available for H8. It supports only one stack shared by the whole application. It is so simple that most of the HAL primitive can be expanded inline.

Multistack HAL This is a bit more complex than the monostack HAL, because it supports multiple stacks allowing the usage of blocking primitives. In practice, there are some code stubs that implements stack changes, which cause some performance loss in terms of execution time and memory footprint.

11.2 Monostack HAL

Internal data structures that each application have to define:

- `ERIKA_ADDR ERIKA_h8_thread_body[THREAD_MAX]`: each element of the array contains the address of the thread. Note that each thread has an identifier which is `0, 1, 2, ...., (THREAD_MAX-1)`;

where `THREAD_MAX` is a symbol which is defined by application.

11.3 Multistack HAL

In addition to the monostack HAL structures, internal data structures defined by the application are:

- `STACKID ERIKA_h8_thread_stack_id[THREAD_MAX+1]`: each thread (plus dummy) is assigned to a stack identifier.
- `ERIKA_ADDR ERIKA_h8_sp[STACK_MAX]`: it stores the SP (Stack Pointer) of each stack.
- `STACKID ERIKA_h8_active_stack_id`: stack identifier of the running thread.

Where STACKID is a symbol defined as `ERIKA_UINT8` (Note that your application cannot use more than 256 stack: I hope that is not a strong limitation).

Let's see an example:

- 4 threads (`THREAD_MAX=4`);
- 3 different stack (`STACK_MAX=3` and 0, 1, 2 are the relative ***stack_id***);
- running thread uses the stack whose identifier is *stack_id*=2;

11.4 Interrupt handling

To install your own handler, you have to follow these steps:

- call this macro **`ERIKA_declare_stub(handler_name)`**, where handler_name is the name of the handler you want to be installed;
- add this line to the **`set_hand.s`** file which must be present in your application directory:

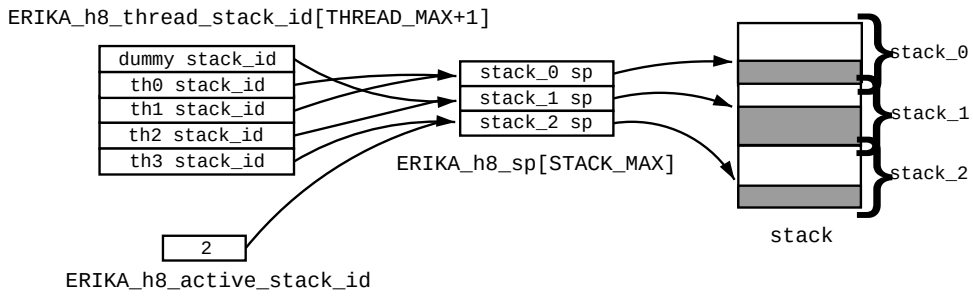


Figure 11.1 A typical stack configuration.

M_set_handler handler_name

- call this function (typically in the dummy) **ERIKA_set_handler(irq_type,handler_name)**, where *irq_ type* is one of the constants specified in section 7.1.1 on page 45;
- call the function **ERIKA_enable_irq(irq_type)** to enable device to produce interrupt requests.

11.5 How to write an application

This sections explains all the application configuration details which are characteristic of the H8 ERIKA implementation.

11.5.1 Dummy thread

As first instruction of dummy, you have to execute **ERIKA_init()** which will provide to initialize all the resources that your application needs and, eventually, it provides to install system timers.

11.5.2 Structure of the Makefile

11.5.3 The ERIKAOPT variable

__H8__ It specifies the H8 development environment used to compile ERIKA code.

__DEBUG__ If defined it enables some debugging facilities providing the assembler of each C file code.

__MULTI__ and __MONO__ These symbols specify if we are using a monostack or multistack HAL. If __MULTI__ is defined then the user must define also **ERIKA_thread_stack_id[]**, **ERIKA_h8_sp[]** and **ERIKA_h8_active_stack_id** in the inithal.c file.

__FP__ These symbols decides what is the wanted scheduler policy. In particular, __FP__ selects the RM with SRP and preemption threshold combination, while __SRPT__ selects the EDF with SRP and preemption threshold combination.

__TIME_SUPPORT__ It provide the sys_getime() primitive to the user

__LCD_USED__ It produces code to manage the LCD display correctly (see 7.1.3 on page 48).

__PLAY_NOTE_SUPPORT__ It produces code to play single notes (see 7.1.6.1 on page 53)

__PLAY_BUFFER_SUPPORT__ It produces code to play melodies (see 7.1.6.2 on page 55)

__MOTOR_x_FP__ If defined, motor x can be used with fixed power control, where x can be A, B, C (see 7.1.4 on page 48).

__MOTOR_x_VP__ If defined, motor x can be used with variable power control, where x can be A, B, C (see 7.1.4 on page 48).

__MOTOR_x_PWM__ If defined, motor x can be used with pulse width modulation control, where x can be A or C (see 7.1.4 on page 48).

__AD_SUPPORT__ It produces code to manage the a/d converter correctly (see 7.1.5 on page 51)

__UNSIGNED_FIXED_USED__ and

__SIGNED_FIXED_USED__ and

__DOUBLE_FIXED_USED__ These symbols decides what is the wanted fixed point math types.

__SF_TRIG_USED__ and

__DF_TRIG_USED__ These symbols have to be defined whether application needs trigonometric functions.

__SF_HIGH_SPEED_TRIG_USED__ and

__DF_HIGH_SPEED_TRIG_USED__ If defined, these symbols optimize trigonometric functions by reducing execution time. Note that this optimization must be used only when you can be sure that every time you call a trigonometric function, the argument passed is an angle whose value is between 0 and 360 degree. If you cannot sure that, this optimization will produce wrong results.

__TIMER_A_USER_FUNCTION_USED__ If defined, application have to define the function

```
void ERIKA_timerA_user_function(void)
```

which will be called by that system driver which is executed every (2 * **TIMER_A_PERIOD**) us. Note that interarrival driver period should be greater than about 100 μs, so application should define the symbol TIMER_A_PERIOD with a value at least equals to 50. Note that this function is executed inside a driver.

__TIMER_B_USER_FUNCTION_USED__ If define, application have to define the function

```
void ERIKA_timerB_user_function(void)
```

which will be called by the system driver which is executed every 2ms. Note that this function is executed inside a driver.

__NO_ROM_DATA__ If you don't define this symbol, it will be reserved an amount of ROM necessary to contain a copy of the initialized RAM data: during the system bootstrap data will be copied from ROM to RAM, so that every time you reboot the system, you will get RAM data correctly initialized just like the linker would make. Obviously that operation increase the quantity of needed memory (data RAM size will be double and some more instructions in the bootstrap code), so if your application does not need to reboot, you should define this symbol. This option does not work yet when using GCC 3 (see __GCC3__ below).

__GCC3__ current ERIKA makefile assume you are using gcc 2.95, if you want to use gcc 3 you can use option, this way you will switch to a modified makefile.

__NO_NEWLIB__ Newlib support is enabled by default when using gcc 3, if you do not want it you cand use this option.

11.5.4 Other variables

There are other variables that the user has to define:

PROJ It is the name of the final object file, without the extension (so the name of the object file will be PROJ.srec)

APP_SRCS It represents the list of the files that form the application, both C and ASM, including the relative path compared to the current directory.

III

Appendices

A

List of Symbols, Types, and Primitives

THIS CHAPTER contains a (hopefully) complete list of the symbols that can be used into the ERIKAOPT environment variable in the makefile, and that can be used in the code to configure the system.

A.1 Symbols and makefiles

These are the symbols that can be inserted into the ERIKAOPT environment variable. Note that the makefile pass these symbols to the target compiler tool-chain using the various command-line options.

So, if a line like the following one is put in the makefile,

```
ERIKAOPT = __FP__ __MONO__
```

a side effect is produced in all the compiled files, in the same way as if the following lines

```
#define __FP__

#define __MONO__
```

would be inserted into the config.h file.

A.1.1 Predefined environment variables

ERIKABASE This is the directory where the ERIKA Educational project is installed; on our PC, we set this in the Cygwin .bashrc:

```
ERIKABASE=/cygdrive/d/erika
```

A.1.2 Target architecture selection

__H8__ The target architecture is a Lego Mindstorms RCX.

A.1.3 Stack Model selection

__MONO__ The Mono Stack HAL for the chosen architecture is selected.

__MULTI__ The Multi Stack HAL for the chosen architecture is selected.

A.1.4 Kernel selection

__FP__ The FP Kernel is selected.

A.1.4.1 Additional options

__ENABLE_THREAD__ The following symbols are defined when this flag is used: `ERIKA_TYPEENABLE`,

`ERIKA_thread_enable()`,

`ERIKA_thread_disable()`,

`ERIKA_thread_isenabled()`.

The user should also define the array `ERIKA_th_enabled[]`.

A.1.5 Timer Support

__TIME_SUPPORT__ This flag enables the timer support. That means that there are available some primitives for read a system-wide timing reference.

A.1.6 Libraries selection

__CABS__ CABs are used in the application. That means that also the CABs code will be linked to the final image.

__SEM__ Semaphores are used in the application. That means that also the Semaphores code will be linked to the final image.

A.1.7 H8 Options

This sections will be updated soon... Please look at Section 11.5.3.

B

Files and directories organization

In this chapter we are going to show the hierarchical structure of ERIKA Educational directories and a short description of their files, addressed to programmers who wants to navigate through the kernel source code. The naming convention is also explained.

B.1 Main organization

As you can see in Figure B.1 on the following page, the ERIKA Educational base directory contains three main directories:

include that contains all the C definition for Kernels and HALs layers;

src that contains all the C and assembly code for Kernels and HALs;

config that contains all the makefiles used by the `make` tool.

Within these directories there are a set of subdirectories that rely on only one of the following naming rules:

- a directory has the form '`hal+stack`' when it contains the code belonging to HAL layers (e.g., `h8`, ...);
- a directory has the form '`sched`' when it contains the code belonging to Kernel layers (e.g., `fp`, `srpt`, ...).

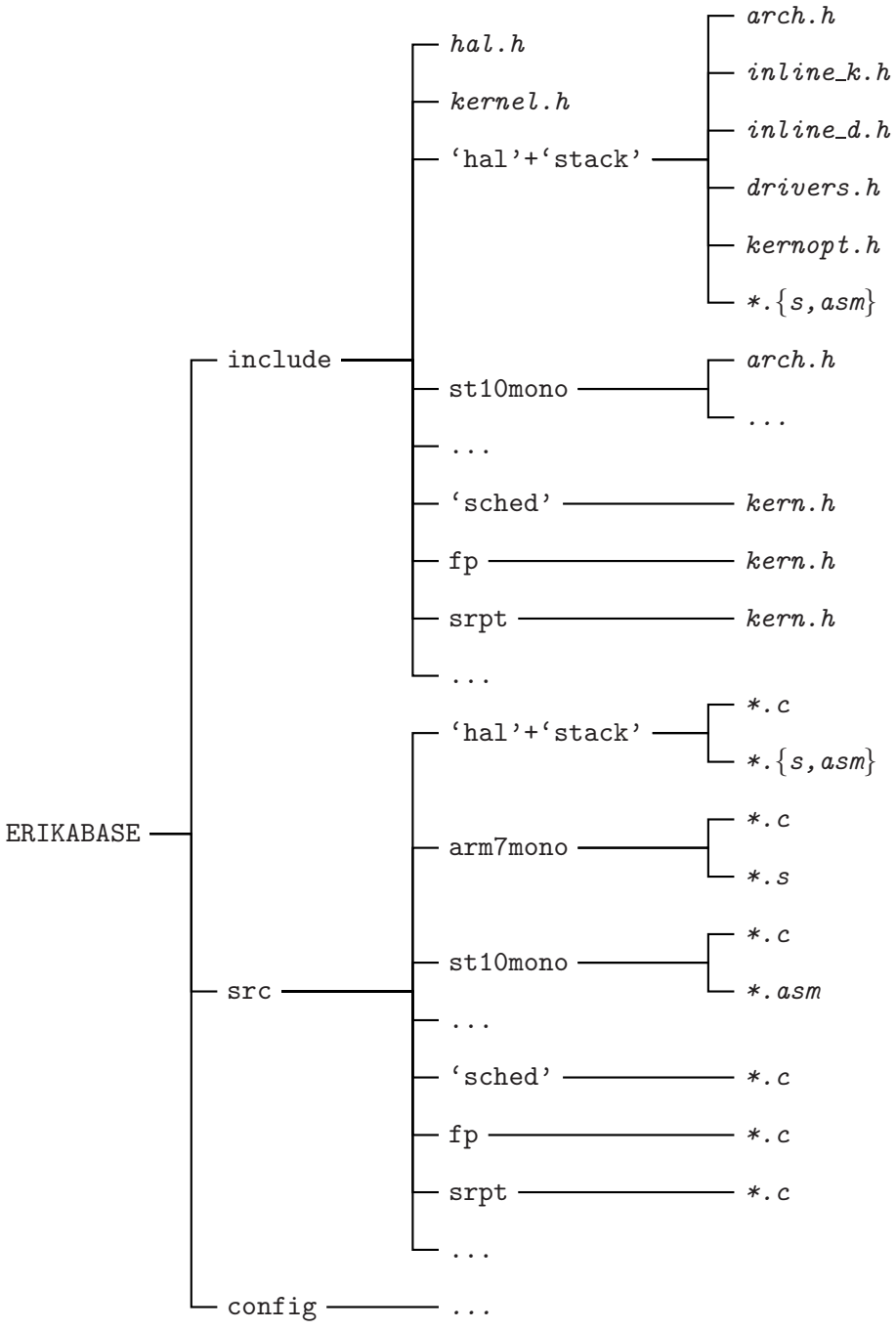


Figure B.1 Kernel directories structure

B.2 The `include` directory

It contains two important files:

`erikahal.h` Functions, data, and symbols declarations of HAL layers.

`erika.h` Functions, data, and symbols declarations of Kernel layers.

B.2.1 HAL layer subdirectory

It contains:

`arch.h` Declarations about hardware/software platform, like HAL types, function, and variables;

`drivers.h` Declarations about hardware peripherals;

`inline_k.h` *inline* definition of some function declared in `arch.h`;

`inline_d.h` *inline* definition of some function declared in `drivers.h`;

`kernopt.h` in this file it is decided whether exists an optimized version for each Kernel layer primitive (see Table 9.1 on page 77);

`*.{s,asm}` typically these files contains some optimized version of Kernel or HAL primitives.

B.2.2 Kernel layer subdirectory

It contains just `kern.h`, in which is defined an hardware independent interface for the Kernel layer. It contains the declaration of all the constants, types, and functions used by the Kernel. It also contains the `dummy()` thread declaration.

B.3 The `src` directory

Here are the bodies of the functions which signature are declared in '`include/*`' subdirectories. For this reason, `src` has the same subdirectory structure as `include`. The code is written in C and ASM, with no particular file naming convention.

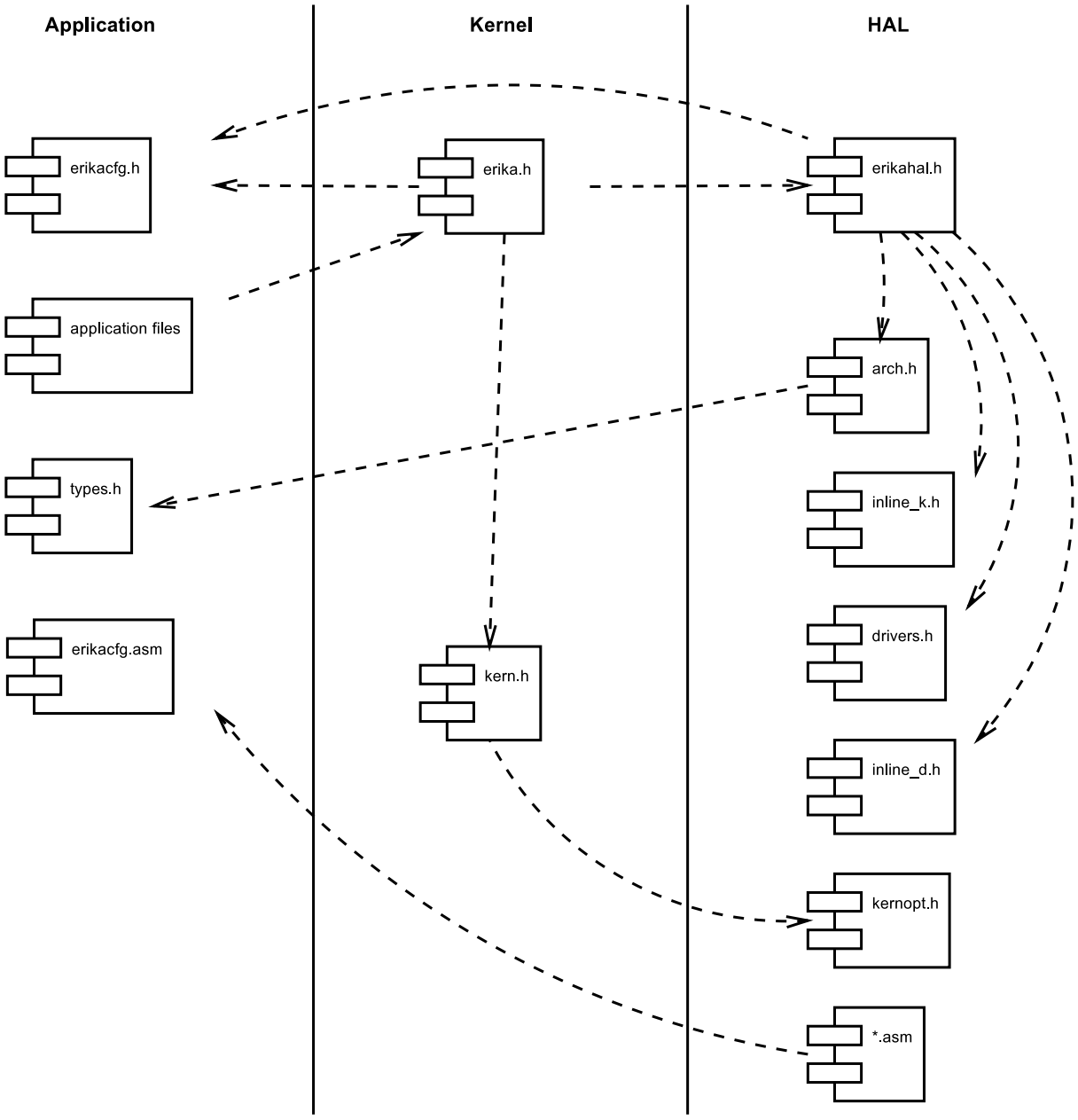


Figure B.2 Inclusion relationship among kernel and application C headers

B.4 The `tests` directory

There are some demonstration applications, tested with various platforms. You can use these examples to learn how to write a new application or to do some profiling measures.

C

Scheduling Algorithm Overview

C.1 Stack Resource Policy (SRP) and non - preemption groups

According to the SRP protocol [1], every hard (periodic and sporadic) task τ_i is assigned a priority p_i and a static preemption level π_i, such that the following essential property holds:

Task τ_i is not allowed to preempt task τ_j, unless $\pi_i > \pi_j$.

Under EDF and Rate Monotonic [4], the previous property is verified if periodic task τ_i is assigned the following preemption level:

$$\pi_i = \frac{1}{T_i}.$$

where T_i is the task period.

In addition, every resource R_k is assigned a static[1] *ceiling* defined as

$$\text{ceil}(R_k) = \max_i \{\pi_i \mid \tau_i \text{ needs } R_k\}.$$

[1] In the case of multi-units resources, the ceiling of each resource is dynamic as it depends on the number of units actually free.

Moreover, a dynamic *system ceiling* is defined as

$$\Pi_s(t) = \max\Big[\{\text{ceil}(R_k) \mid R_k \text{ is currently busy}\} \cup \{0\}\Big].$$

Then, the SRP scheduling rule states that

> *"a job is not allowed to start executing until its priority is the highest among the active jobs and its preemption level is greater than the system ceiling."*

The SRP ensures that once a task is started, it will never block until completion; it can only be preempted by higher priority tasks.

This protocol has several interesting properties. For example, it applies to both static and dynamic scheduling algorithms, prevents deadlocks, bounds the maximum blocking times of tasks, reduces the number of context switches, can be easily extended to multi-unit resources, allows tasks to share stack-based resources, and its implementation is straightforward.

Under the SRP there is no need to implement waiting queues. In fact, a task never blocks its execution: it simply cannot start executing if its preemption level is not high enough.

The SRP protocol allows all the tasks to share the same stack, reducing the memory requirements of the application. However, in some application environments this may not be sufficient. To cope with these memory requirements the SRP protocol can be enhanced using non-preemption groups[3][7][6].

A *non-preemption group* is a set of tasks that shares the property that only one task of the set can be executed or can be stacked at one time. In other words, when a task becomes the running task, it acquires the highest priority of the tasks that belongs to its non-preemption group.

In this way, with an off-line schedulability analysis, the tasks can be grouped into non-preemption groups to limit the number of tasks that can be on the stack at a specified time. Also, the total number of different priorities in the system can be reduced.

The non-preemption groups implementation can be done simply decoupling the task priority in two values:

- *ready priority*, that is the priority used to queue a task into the system ready queue;

- *dispatch priority*, that is the priority a task acquires when it is executed.

This way of setting task priorities can also be viewed as if the tasks belonging to a non-preemption group implicitly acquires a shared resource just before beginning execution.

D

GNU General Public License

Version 2, June 1991

59 Temple Place, Suite 330, Boston, MA

02111-1307 USA

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we

want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute

the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who

have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software

Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

No warranty

11. Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.

12. In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

End of terms and conditions

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) 19yy  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.
```

```
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

E

GNU Free Documentation License

Version 1.1, March 2000

0 Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, TEXinfo input format, LATEX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3 Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these

Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

(a) Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

(b) List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

(c) State on the Title page the name of the publisher of the Modified Version, as the publisher.

(d) Preserve all the copyright notices of the Document.

(e) Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

(f) Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

(g) Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

(h) Include an unaltered copy of this License.

(i) Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

(j) Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

(k) In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

(l) Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

(m) Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

(n) Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same

name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7 Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will

automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10 Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliography

[1] T.P. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3, 1991.

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

[3] Robert Davis, Nick Merriam, and Nigel Tracey. How embedded applications using an rtos can stay within on-chip memory limits. In *Proceedings of the Work in Progress and Industrial Experience Session, Euromicro Conference on Real-Time Systems*, June 2000.

[4] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[5] Kekoa Proudfoot. http://graphics.stanford.edu/ kekoa/rcx/. available on Internet.

[6] Manas Saksena and Yun Wang. Scalable real-time system design using preemption thresholds. In *Proceedings of the Real Time Systems Symposium*, December 2000.

[7] Yun Wang and Manas Saksena. Fixed priority scheduling with preemption threshold. In *Proceedings of the IEEE International Conference on Real-Time Computing Systems and Applications*, December 1999.