ARCHITECTURE FOR A PORTABLE OPEN SOURCE REAL TIME KERNEL ENVIRONMENT

Paolo Gai, Enrico Bini, Giuseppe Lipari, Marco Di Natale, Luca Abeni
ReTiS Lab, Scuola Superiore di Studi e Perfezionamento S. Anna - Pisa,
pj@hartik.sssup.it,{bini,lipari,marco,luca}@sssup.it

Abstract

This paper describes a set of open source, integrated tools for developing embedded real-time applications, especially targeted to the automotive industry. The set of tools comprises a schedulability analyzer, a kernel library and a kernel configurator. The goal is to achieve portability of the application over different hardware platforms without sacrificing performance. With our tools, the programmer only need to write the application tasks using standard kernel primitives; then the optimization phase is carried out with the help of the kConfigurator tool, which automatically configures the kernel library to obtain maximum performance. So far, the tools have been provided for ST10/C166 microprocessor based boards, we are currently working on a version for ARM 7TDMI based boards.

1 Introduction

This paper describes a project currently under development at the Real Time Systems (RETIS) Laboratory of the Scuola Superiore S. Anna in the context of the MADESS Research Project. One of the goals of the MADESS project is the development of an Open Source Real Time Kernel for embedded automotive applications.

Developing a kernel for the automotive world is a critical issue. Many devices in a car are controlled by hardware/software components, and more will be controlled in the near future. All the applications that run on these devices are time-critical. The system performance must be guaranteed in all load conditions.

A real-time kernel must provide suitable real-time scheduling algorithms. Typically, these kernels implement a cyclic executive, or a fixed priority deadline monotonic scheduling algorithm[1] with a Non-Preemptive protocol for accessing mutually exclusive shared resources. Using the appropriate scheduling algorithm, the application can then be guaranteed off-line to meet all its time constraints. In addition, real-time kernels for embedded system have very demanding requirements on code size and interrupt handling response time. These kernels are usually very fast and small: they must fit in a few kbytes of memory, and require the least possible amount of RAM (RAM is more expensive than ROM).

In the last years there has been a strong commitment for a standardization of the interfaces that these kernels should provide (the OSEK/VDX standard[6] is a good example). In fact, in order to reduce the development costs, it is important to be able to port the same application on different hardware architectures with a minimum effort. The challenge is in providing a standard interface without sacrificing too much the overall performance.

2 System architecture

The goal of this work is to provide a set of integrated tools for developing embedded real-time applications, especially targeted to the automotive industry. The set of tools comprises a schedulability analyzer, a kernel library, and a kernel configurator. The schedulability analyzer tells the system designer if the application is schedulable on the given architecture with the given constraints. It also provides useful informations for helping the designer in choosing the right priority assignment: for example, it is possible to compute the minimum number of priority levels that makes the application schedulable. With this information, and using the appropriate scheduling strategy, it is possible to minimize the system stack footprint.

In many cases, the usage of (hardware and kernel) resources and the performance of the kernel can be greatly improved if off-line information about real-time application tasks is available and exploited. For example, it is possible to reduce the amount of memory allocated to the task stacks by optimizing the number of priority levels, or to reduce the task acti-

vation latency in cases where a number of periodic tasks must be activated on an hardware offering more than one hardware timer. In this case, knowledge of tasks periods can be used to cluster harmonic tasks in sets to be activated by a different timer. The kConfigurator tool analyzes the off-line configuration and tries to find an optimal mapping between the application requirements and the hardware resources. This tool automatically produces the kernel code and data that best suits the given application configuration.
These tools will be made available under the Gnu General Public License.

2.1 Kernel architecture.

The architecture of the our kernel is described in Figure 1, and 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 like Rate Monotonic[5][1], cyclic executive, etc.; resource management protocols like Priority Inheritance and Stack Resource Policy [2]; interrupt processing and error treatment. Basically, the kernel modules offer a common interface containing the basic kernel services to applications. These modules are developed using portable C code. They are built upon a common interface to the underlying hardware, so to enhance the portability of the kernel code between different hardware platforms.
The Hardware Abstraction Layer (HAL) contains some very basic services that are architecture-specific, like context-switch, interrupt handling, low-level timer and memory management initialization. This layered approach trades (very little) performance for the confinement of all hardware specific code in a very limited number of Hardware Abstraction Layer services. When porting the kernels on new architectures, only the HAL has to be rewritten.
Note that different applications can require different level of services. For example, in the automotive industry, one of the constraints is the RAM footprint to be as small as possible. To do this, a mono-stack kernel solution should be preferred against a multi-stack one. In a mono-stack kernel all tasks use the same stack; hence, the concurrency model used by the kernel must be constrained: it is not possible to interleave the execution of the tasks. For example, one such model could be a the Rate Monotonic fixed priority scheduler with the SRP[2] for mutual exclusive critical sections. Instead, the Priority Inheritance[7] algorithm allows interleaving in tasks' execution, and hence it is not suitable for a mono-stack model.
Other types of applications, that have less constraints on the RAM usage, can use a different scheduler that can be built on the multi-stack model. Of course, the HAL implementation, which provides the context switch between tasks, depends on the stack model. In our work, we provided two HAL for the target architecture, one suitable for mono-stack models, the other one for multi-stack models. Depending on the characteristics of the application, the system designer has the possibility to choose what to optimize.

2.2 The Kernel Configurator

One of the problems of designing and implementing embedded system is to optimize the performance of the system maintaining at the same time an architecture-independent approach. The limited availability of resources and the real-time nature of the applications impose an accurate optimization process. On the other hand, it is desirable to maintain the maximum portability and re-use the application code as much as it is possible. Unfortunately, these two goals are often in contrast.
Our approach to this problem consists in

- using a kernel with a standard interface; the hardware features are made available through standard system calls;
- writing the application tasks in a high level, hardware-independent language (like ANSI C), using the kernel primitives to access the hardware features.

The optimization is then carried out using an automatic configuration tool called kConfigurator which tries to map the application to the selected hardware platform. The kConfigurator takes as inputs the C files containing the application tasks and one or more *description files* that defines the hardware architecture, the application task structure, the peripherals, etc. The kConfigurator produces a set of kernel and application files, that can be compiled and linked to the final binary image. It tries to minimize the code size by linking only the useful kernel modules; moreover, it will configure each kernel module getting advantage of the application structure. For example, depending on the selected scheduling strategy, a mono-stack HAL or a multi-stack HAL must be selected.
In this way, it is possible to change the hardware architecture transparently, without modifying the application code. The user can still bypass the configuration tool and configure the kernel code by hand; however, this is often a tedious and error-prone job. Instead, the kConfigurator provides a safe and easy way to avoid errors; it is also possible to explore different solutions to find the configuration with better

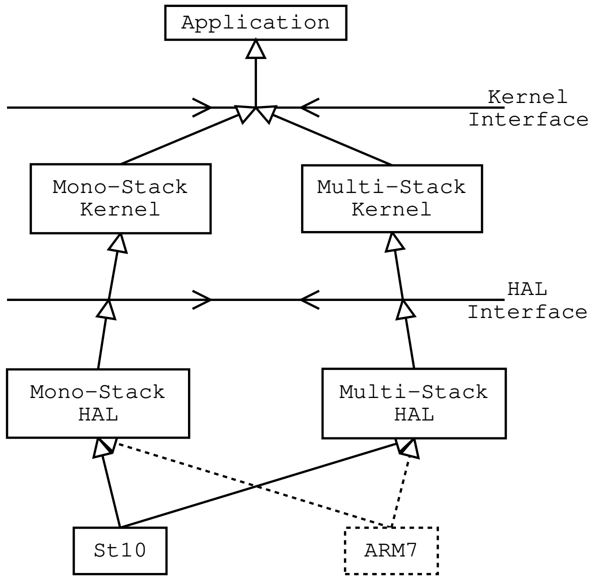

Figure 1: Architecture of the Kernel and HAL Layers.

performance without messing with the application code.
The configuration process is described in Figure 2:

- The box labelled with "Application Description" represents the application files. They consists in:
  1. one or more C files which contain the tasks' code;
  2. a task definition file containing the characteristics of each task, such as the periodicity (periodic, sporadic, aperiodic), the period length, the priority, etc. ;
  3. a peripheral description file, which describes the peripherals that are used by the application (interrupt level, corresponding task/routine to be activated, etc.);
  4. the scheduling policy.
- The box labelled "Hardware & Peripherals Description" represents all the platform dependent informations as:
  1. type of hardware architecture (by now it is only possible to select the ST10 architecture, we are currently working on an ARM 7TDMI board);
  2. number and type of available peripherals;
  3. the commands to load and configure a driver in a specific way;
  4. the syntax of the system calls used in the library.
- The rounded box "kConfigurator" is the main tool. In order to give an idea of its functionality, here's a sample of what it can do:
  1. set the system variables to the smallest size types according to the maximum possible value. This means, for example, that in case of less than 256 task, in order to minimize RAM usage, the task identifier type can be a byte instead of the larger word;
  2. configure the timers, trying to optimize their usage. For example, if the hardware platform provides two hardware timers, we must program them to account for all the periodic tasks in the application. The kConfigurator explores the possibility to cluster the tasks in two sets, each one containing only harmonic tasks. Then, it configures each timer using appropriate lookup tables.
- "Kernel Library" contains the kernel files implementing task management (preemption handling, mutexes etc.), and peripherals management (drivers).
- "Configuration" represents a set of files containing:
  1. *#define* directives, to optimize the modules usage (avoiding to load unused ones);

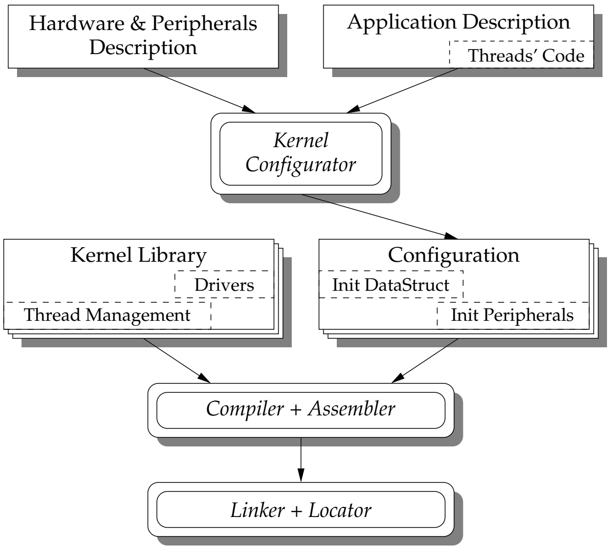

Figure 2: The kernel configuration process.

2. assembly macros to load only the strictly needed drivers;

3. initialization of data structures used by the kernel as the list of task descriptors, the periods of each timer, the first running task etc.;

4. initialization of code structures (as interrupt configuration data, task communication structures, etc.).

If a new architecture has to be introduced in the system, the following steps have to be done:

1. Implementing the new HALs for the selected microprocessor. This include writing the code for the context switch among tasks, the interrupt service subroutine, etc.

2. Implementing the external driver subroutines.

3. Modifying the Hardware & Peripheral Description box in order to let the configurator know about the new architecture.

Currently, the kConfigurator tool has been implemented and tested for the ST10 microprocessor architecture. We are currently writing a mono-stack and a multi-stack HAL for the ARM 7 TDMI multiprocessor. It is possible to download the kernel code and the kConfigurator code from the ReTiS web site (*http://retis.sssup.it*).

3 Current implementation of the kernel.

So far, we implemented our kernel architecture on ST10/C166 microprocessor based boards. We provided both a multi-stack and a mono-stack version of the kernel: in the mono-stack model it is possible to use Fixed Priority scheduling and the Stack Resource Policy with non-preemption groups [3]; in the multi-stack model it is possible to choose Earliest Deadline First [5][4], Cyclic Executive and Round Robin scheduling algorithms, Priority Inheritance [7], semaphores, mutex and condition variables. Both models support event and time handling routines.
In the following paragraphs we explain the internal structure of the mono-stack version of the kernel.

3.1 Data structures

Each task is represented by a normal C function. The execution model for a task in our kernel is the one-shot model: the body function implements only an instance. If the task is recurring (periodic or sporadic), then each time it is activated, the corresponding function is activated. Each task is statically created at system initialization and cannot be killed.
Each task has a descriptor containing the following informations:

- a pointer to the task body;
- the ready and the dispatch priority;
- the status of the task (can be IDLE, STACKED or READY);

- the number of pending activations;
- an index in the table used to queue the tasks.

The task descriptor must be initialized directly by the user (or by the kConfigurator) before the system starts.
The task priorities are coded using a 16 bit word, each priority level corresponds to a bit. Hence, only 16 different priority levels can be coded in the system (usually, this limit is enough for embedded applications). The most significant bit represents the highest priority (that is, priority `0x0010` is higher than `0x0001`).
The kernel keeps track of a task status through a three-state variable that can have these values:

IDLE When a task is in the idle state, it is not queued in any data structure, and it is waiting for an explicit activation through a call to thread_activate or through an interrupt request.

READY A task becomes READY from the IDLE state when it is activated, but has not yet started execution because the system ceiling has a value greater than or equal to its priority.

STACKED A task goes into this state when it becomes the running task: the state does not change if the task is preempted by another task.

Moreover, some data structures track the global system state:

- a *ready queue*, that manages the activated tasks that are waiting for execution;
- a *stacked queue*, that stores the task stack layout, queuing all the tasks that are preempted on the stack, starting from the running task;
- a *system ceiling*, that is a 16 bit word that stores the "busy" priority levels.

Also, a task can share mutually exclusive resources with other tasks using mutex semaphores. The Stack Resource Policy has been chosen as the concurrency control protocol: see Appendix A for a brief survey of the algorithm. The current implementation provides support for single unit resources; the only thing that the system needs to know about a mutex is the highest priority among all the tasks that use it.

Because the priorities are implemented through bits, it is possible to store the running task priority, the priorities of the stacked tasks and the preemption levels of every mutex locked by a task[1] in a single variable called system_ceiling. In this way, the preemption test can be done using a single *branch and jump* assembler instruction, and the system ceiling manipulation can be done using efficient bitwise operators such as `AND`, `NOT` and `OR`.

3.2 The primitives

The kernel provides three primitives to manage tasks activations and mutexes:

thread_activate This primitive can be called by the user in a task body to explicitly activate a task. If the target task is already active, a pending activation is saved; when the target task finishes the current instance, it will be activated again by the kernel;

mutex_lock This non-blocking primitive simply updates the system ceiling according to the preemption level of the locked mutex. On the ST10 micro controller it is implemented with 4 assembler opcodes protected by an ATOMIC instruction;

mutex_unlock This primitive implements the unlock of a mutex and also checks to see if a preemption have to be made.

Moreover, the kernel provides an end cycle primitive that is called automatically at the end of a task instance[2]. That killer function is used to wake up the next task and to check for preemptions. The kernel provides also a template that can be used to write interrupt handlers. This template is also used by the kernel configurator to support its optimizations.

3.3 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 task set depicted in Table 1 with the corresponding initialization values. The task set is composed of six tasks, each with different priority; there is a non-preemption group composed by task 5 and 6, and there is a shared resource used by tasks 2 and 4 through a mutex. The system ceiling[3] has a

[1] Note that the fact that a task has locked a mutex implies that that task is in the STACKED state. When a task locks a mutex, the system have to save the system ceiling value to restore it when the mutex will be unlocked, because the preemption level of a mutex can be the same of the priority of a task (no bitwise operators can be used).

[2] Never by the user...

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

Task Number	Ready Priority	Dispatch Priority	Initial Values		
			Status	Pend. Act.	next
1	0x01	0x01	STACKED	0	NIL
2	0x02	0x02	IDLE	0	NIL
3	0x04	0x04	IDLE	0	NIL
4	0x08	0x08	IDLE	0	NIL
5	0x10	0x20	IDLE	0	NIL
6	0x20	0x20	IDLE	0	NIL

Table 1: The example task set.

starting value of 0x01 (because the task 1 starts on the stack).
Then suppose that these events happen in the system:

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

After these events, the configuration of the kernel data structures is shown in Figure 3.

3.4 Performances

The performance of our mono-stack based kernel implementation are depicted in Table 2 and explained in more detail in the next subsections. The time duration of each primitive is measured from the instruction before the primitive call opcodes to the end of the return opcode.

3.4.1 ROM Footprint

The ROM footprint is 1020 bytes of kernel code, plus 8 bytes for each task (the task body function pointer, the ST10's context pointer, the dispatch and the ready priority), 2 bytes for each mutex (the mutex preemption level), plus a few bytes of ROM used to initialize the RAM data with the starting values.

3.4.2 RAM Footprint

The RAM footprint is 6 bytes of global variables (the system ceiling plus the ready and stacked queue), plus 4 bytes for each task (the task status, the pending activations plus the next pointer), plus 32 bytes for each priority level (one ST10 register bank for each priority level).

3.4.3 System stack usage

The system stack usage for a task is 16 bytes plus automatic variables. The stack footprint of a preempted task is shown in Figure 4 and is formed by the following items:

endcycle This is the return address which the CPU jumps to when a task instance ends. Obviously, it is the address of the first assembler instruction of the internal endcycle primitive. When it is called, the task does not use any byte of the stack anymore;

automatic variables It is the space used by the compiler to allocate the automatic variables of a task;

prim. call This is the return address that is pushed on the stack only if the task was preempted after a call to some primitive (i.e., thread_activate, mutex_unlock); it is not present if the task was preempted by an interrupt;

PSW, IP These values are pushed on the stack by a primitive or by the CPU interrupt response;

MDC, MDL, MDH These are a copy of the ST10's multiply registers, and they need to be saved every time a task is preempted;

CP This is the ST10's context pointer used by the preemption level to which the preempted task belongs to.

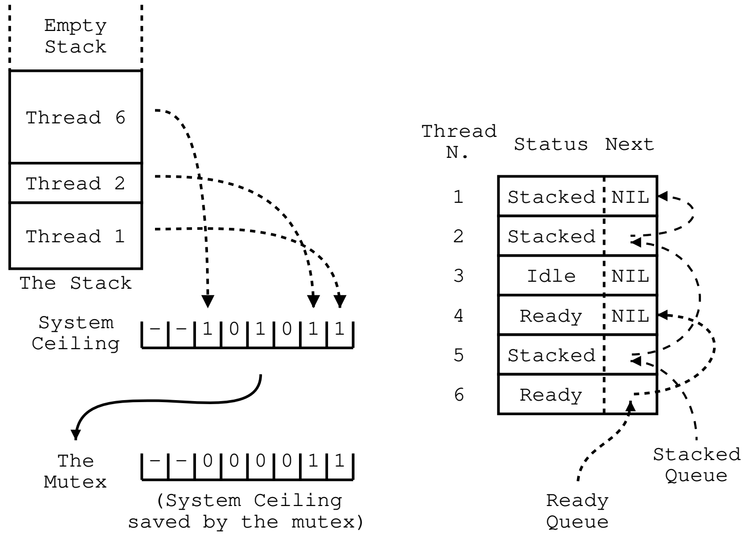

Figure 3: Kernel Data structures snapshot after the events listed in 3.3.

Index	Value
ROM Footprint	1020 bytes, + 8 bytes for each task + 2 bytes for each SRP Mutex
RAM Footprint	6 bytes global variables + 4 bytes for each task, + 32 for each priority level
System Stack usage	16 bytes for each priority level, + interrupts frames
mutexes	lock: 2.2 microseconds, unlock from 4 to 10 microseconds
activate	from 4.8 to 7.2 microseconds
end instance	from 7.6 to 12 microseconds
interrupt latency	3.2 microseconds

Table 2: Performance metrics of the mono-stack ST10 kernel.

3.4.4 Mutexes

The mutex_lock takes 2.2 microseconds, which is is the time the CPU executes the 5 assembler opcodes. The mutex_unlock primitive takes 4 microseconds when no context switch occurs and 10 microseconds when the context must be switched.

3.4.5 Activate

The thread_activate primitive takes 4.8 microseconds if the primitive simply increments the pending activation counter, 6.8 microseconds when the activated task is inserted in the *first* position of the ready queue, and 7.2 microseconds when the activated task becomes the running task; see Figure 5.

3.4.6 End instance

The end instance function takes 7.6 microseconds[4] when the new running task was already stacked (see Figure 5) and 12 microseconds[5] when the running task is woken up from the ready queue.

3.4.7 Interrupt Latency

The interrupt latency value of 3.2 microseconds is computed from when the interrupt fires to the first C instruction of the handler routine.

4 Conclusions and future works

We plan to port our kernel on an ARM 7TDMI platform and on a Dual ARM architecture. The Dual ARM is a novel architecture, consisting of two ARM 7TDMI processors communicating through a cross-

[4]Suppose that thread A activates thread B. When the B instance ends the endcycle is called. The time is measured from before the execution of the final } in B to before the first opcode in A after the thread_activate primitive. The CPU takes 1.2 microseconds from the first instruction of the thread_activate after the end of B to the thread_activate end.

[5]from before the execution of the final } of the ending thread to before the first opcode of the thread woken up from the ready queue.

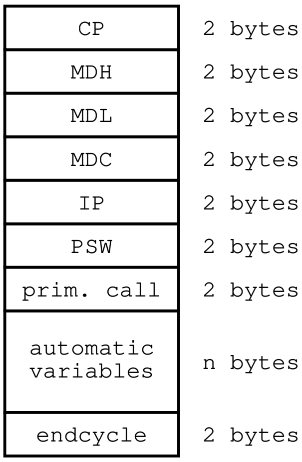

Figure 4: Typical Stack layout of a preempted task.

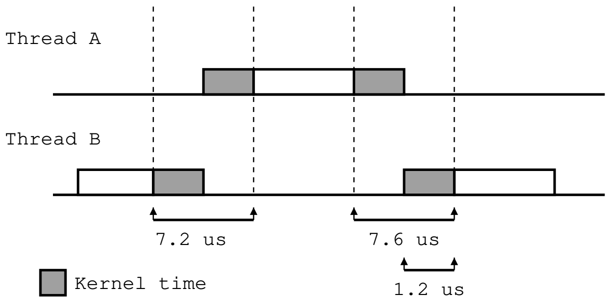

Figure 5: Time diagram for the thread_activate and the endcycle primitives.

bar switch. In this sense, we are investigating the possibility to develop an embedded multiprocessor real-time kernel that it is not simply "two kernel running on two CPUs", but that will include specific multiprocessor schedulers and protocols to get full advantage of the multiprocessor architecture.

A Stack Resource Policy (SRP) and non-preemption groups

According to the SRP protocol, 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, 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[6] *ceiling* defined as

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

Moreover, a dynamic *system ceiling* is defined as

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

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

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

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].

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 is the priority used to queue a task into the system ready queue;

dispatch priority is the priority that 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.

References

[1] N.C. Audsley, A. Burns, R. Davis, K. Tindell, and A.J. Wellings. Fixed priority pre-emptive scheduling: An historical prespective. *Real-Time Systems*, 8:173–198, 1995.

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

[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] M. L. Dertouzos. Control robotics: The procedural control of physical processes. *Information Processing*, 1974.

[5] 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.

[6] OSEK. *OSEK/VDX: Open systems and the corresponding interfaces for automotive electronics.* http://www.osek-vdx.org/mirror/os21.pdf.

[7] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE transaction on computers*, 39(9), September 1990.