UNIVERSITÀ DI PISA

Facoltà di Ingegneria

corso di studi in Ingegneria Informatica

master thesis

Real-time kernel support for engine control applications

Academic Year 2014-2015

Supervisor

Prof. Ing. Giorgio Buttazzo

correlator

Dott. Mauro Marinoni

candidate

Vincenzo Apuzzo - mat. 488443

If something can go wrong, it will go wrong.

Murphy's Law

Abstract

Engine control applications typically include computational activities consisting of periodic tasks, activated by timers, and engine-triggered tasks, activated at specific angular positions of the crankshaft. Such tasks are typically managed by a OSEK-compliant real-time kernel using a fixed-priority scheduler, as specified in the AUTOSAR standard adopted by most automotive industries. Recent theoretical results, however, have highlighted significant limitations of fixed-priority scheduling in managing engine-triggered tasks that could be solved by a dynamic scheduling policy.

This master thesis proposes a new kernel implementation within the ERIKA Enterprise operating system, providing EDF scheduling for both periodic and engine-triggered tasks. The proposed kernel has been conceived to have an API similar to the AUTOSAR/OSEK standard one, limiting the effort needed to use the new kernel with an existing legacy application.

A simulation framework is presented, showing a powerful environment for studying the execution of tasks under the proposed kernel. Such framework is based on Lauterbach Trace32 Cortex simulator and it was extended with custom plugins for testing the proposed kernel.

Performance tests are designed and executed in order to evaluate the proposed kernel in terms of run-time overhead and footprint, that represent the main drawbacks of the earliest deadline first kernel with respect to the fixed-priority scheduling.

The thesis is organized as follows: the first chapter is an introduction about the engine control and related problems; then a related works and studies are presented, moreover the theoretical model of the engine control is reported. The second chapter shows the system architecture, with a description of the software tools and hardware devices adopted. Chapter four describes the design of the simulation framework with a special attention to the developed plugins, needed for simulating the proposed kernel. Then the experimental environment and result are shown and discussed.

Contents

List of Figures

List of listing

List of Tables

Chapter 1

Introduction

The automotive industry is one of the world's most important economic sectors. It is characterized by high competitiveness which results in a particular attention to innovation and in the continuous research of new solutions. Providing a vehicle with new features is a must in order to make it stand out from the others.
The aspects which are considered when developing new car models are the ecological impact, the performances of the vehicle, the safety and the costs. Often the final product is the result of a balancing process between the provided features. It is not only a matter of design of the look, there are also many constraints coming from safety requirements and pollution restrictions.
The modern trend to deal with this challenging environment has been to include the electronics components and software inside the vehicles. Nowadays vehicles parts are handled and controlled by many microcontrollers (Fig. 1.1), and these embedded systems are becoming an important part in the design of new cars.
The microntrollers in the vehicles provides some features, for instance car temperatures with performance controls for air conditioning maintenance, braking mechanisms, engine control and so on.

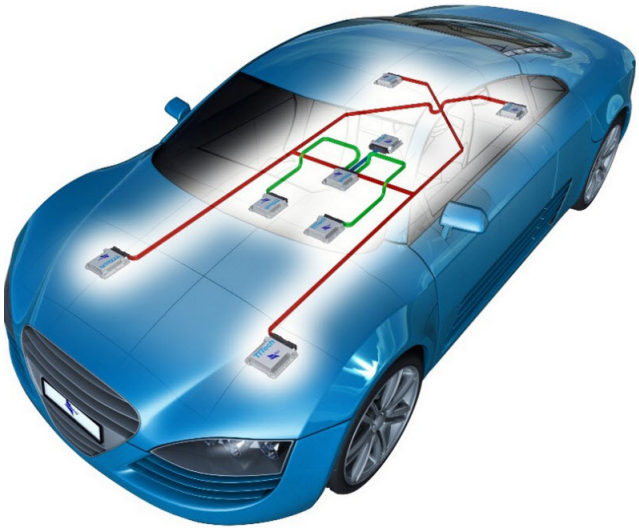

Figure 1.1: Microcontrollers used in a car.

In this scenario, the engine control, represents an important field of research, in order to meet performance and dependability requirements, imposed by the market. Generally engine control applications are characterized by two types of tasks: the periodic tasks, activated at constant time intervals and the aperiodic tasks, activated at specific crankshaft rotation angles. This kind of activation involves many problems related to the angular velocity, in fact both the activation rate and the computation time are proportional to the engine speed: the higher the engine speed, the higher the activation rate and the lower the available computation time.
This aspect can generate an overload condition (Fig. 1.2) on the engine control unit (ECU) processor, in fact for high activation rate the system utilization can increase beyond a limit, with disruptive effects on the controlled system, introducing unbounded delays on the computational activities, or even leading to a complete functionality loss.

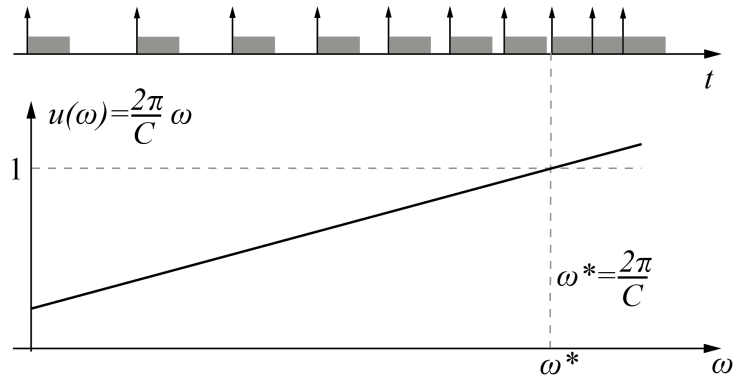

Figure 1.2: Overload condition.

To prevent such problems, a common practice adopted in automotive applications, is to implement tasks in such way they adapt the computational requirements and functionality to engine speed (Fig. 1.3). To do this, the task functionality are implemented as a sequence of conditional *if* statements, each execute a specific subset of function [1]. For this reason they are called *Adaptive Variable-Rate Task* (AVR Task).

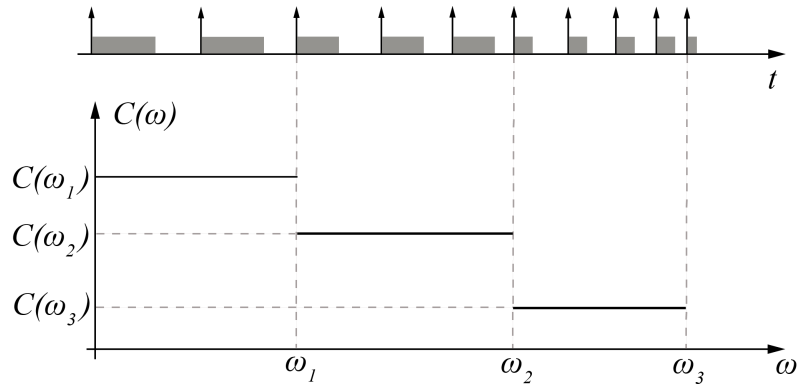

Figure 1.3: AVR Task utilization.

In engine control systems, the AVR Task are used to handle many operations, such as the fuel injection, and they are characterized by an angular position, the crankshaft absolutely position, and by an angular deadline, namely the angular deviation within task must end.
In a four cylinder engine, the pistons are paired in phase opposition, hence when a couple is in *Top Dead Center* (TDC) position, the other one is in *Bottom Dead Center* (BDC), as shown in Fig. 1.4. These points represents, respectively, the

position of a piston in which it is faster from, and nearest to, the crankshaft. The TDC is a typical reference point, in the controller activities, for the functions that are rotation dependent. These functions include the control of the combustion, the computation of time for injecting and the quantity of fuel to be injected.

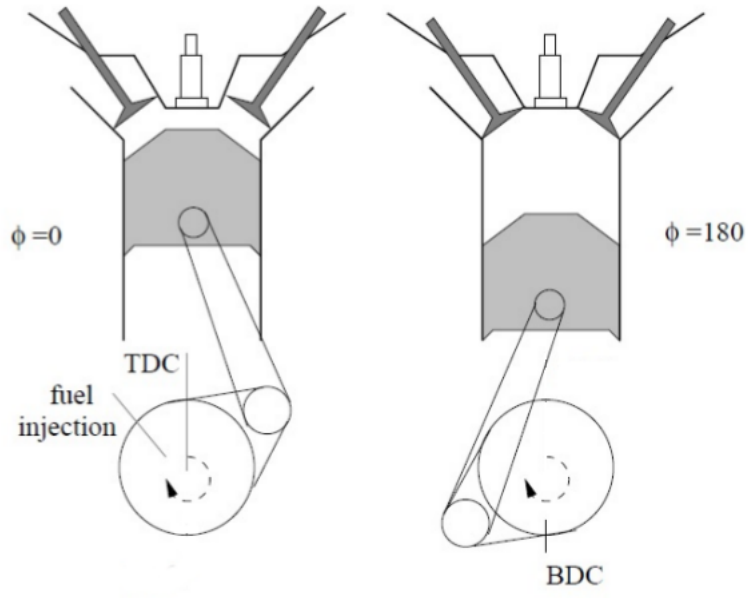

Figure 1.4: Position and phase of paired cylinders.

The case study of fuel injection analyzed by Guzzella et el. [2] can be taken like exhaustive example of a control process. The goal of a fuel injection system is to determine the point in time and the quantity of fuel to be injected in the cylinders, relative to the position of each piston, which is a function of the angular position of the crankshaft. The Fig. 1.5 shows the timing diagram for injection for two cycle.

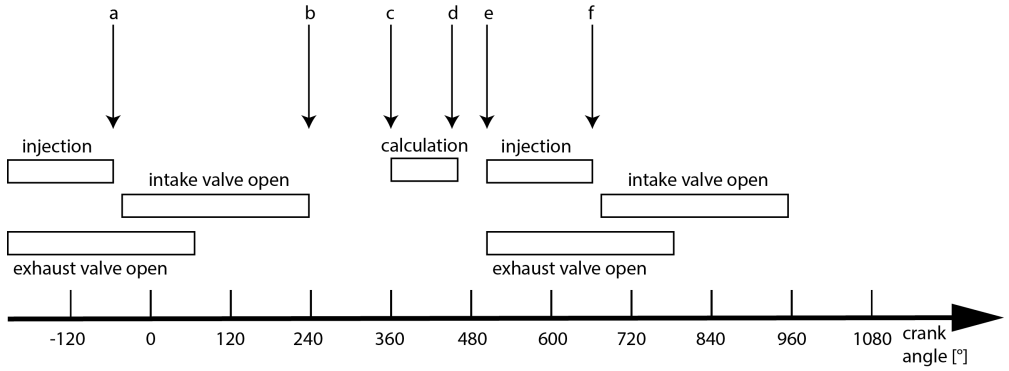

Figure 1.5: Timing diagram for the injection.

a) Injection completed. In the ECU, this event is fixed. The start of injection is set at the necessary crank angle, which is computed using the air mass flow information obtained at an earlier crank angle.

b) Start of injection for the next cycle.

c) Measurements for the calculation of the duration of the injection are taken.

d) Calculation ended.

e) The injection with the most recent air mass information starts.

f) Injection pulse is finished.

Given these characteristics of the AVR Tasks, the classical real-time approaches to schedulability analysis cannot be done, so several analysis methods have been proposed by several authors.
Kim, Lakshmanan and Rajkumar [3] derived a preliminary schedulability result under simple assumptions. In particular, they analyzed a task set with a single AVR task under Fixed Priority algorithm, with the priority levels assigned by the Rate Monotonic algorithm, moreover the period of the AVR task was always smaller than the period of the other task, hence it runs always at the highest priority level.
Pollex [4] presented a sufficient schedulability analysis under fixed priorities, assuming a constant angular velocity. The analysis is formulated using continuous interval, hence it cannot be translated into a practical schedulability test, whose

complexity has not been evaluated.
The dynamic behavior of AVR task under fixed priorities has been analyzed by Davis [5], who proposed a sufficient test based on the quantization of the instantaneous crankshaft rotation speed, which may introduce additional pessimism in the analysis to guarantee the safety of the test.
A complete analysis of the interference of an AVR Task under fixed priorities has been done by Biondi et al. [6]. He analyzed the interference using a search approach in the speed domain, where the complexity is contained by deriving a set of dominant speeds, which also avoid quantizing the instantaneous speed considered in the analysis.
In previous papers the analysis has always been done under the fixed priority scheduling, which is the de facto standard in industry and it is the scheduling algorithm used in OSEK/AUTOSAR real time operating system, because it is very easy to implement, it introduced less runtime overhead and it is more predictable in overload conditions.
On the other hand, as exhaustively analyzed by Buttazzo [7], where the behavior of *Fixed Priority* (FP) and *Earliest Deadline First* (EDF) scheduling has been compared under many different conditions such as runtime overhead and overload condition, the real advantage of FP with respect to EDF is its simpler implementation in commercial kernels that do not provide explicit support for timing constraints, such as periods and deadlines. Other properties typically claimed for FP, such as predictability during overload conditions, or better jitter control, only apply for the highest priority task, and do not hold in general. On the other hand, EDF allows a full processor utilization, which implies a more efficient exploitation of computational resources and a much better responsiveness of aperiodic activities. For this reason Buttazzo and Gai et el. [8] proposed an efficient implementation of the EDF scheduler, based on the implicit circular timer's overflow handler (ICTOH) to better represent the deadline in a circular time model. Moreover they proposed an integration in the OSEK standard, adding few keywords in the standard OIL specification language, illustrating how EDF can be easily integrated with existing operating systems.
The analysis of a mixed set of classical and AVR tasks under Earliest Deadline First (EDF) scheduling has been addressed by Buttazzo, Bini, and Buttle [9],

but for AVR tasks related to independent rotation sources. They also provided a design method that allows computing the set of switching speeds at which modes have to be changed to keep the overall utilization below a desired bound. Although the results produced in the previous papers represents important milestones for the analysis of engine control systems, the task models used for the analysis are not always able to capture features that are currently adopted by the automotive industry in the implementation of AVR tasks. For example, in some work [9], tasks are considered to be linked to independent rotation sources, while in reality all the angular tasks related to engine control are linked to the same rotation speed and may be triggered at different rotation angles. A typical engine control application includes both classical periodic tasks (with period ranging from a few milliseconds up to 100 ms) and a number of angular tasks activated every single, half, and quarter engine revolution (given the range of values in which the engine speed can vary, typically from 500 to 6500 rpm, the interarrival time of an AVR task varies from 9 to 120 ms with a single activation per cycle). The assumption of independency clearly simplifies the analysis, but introduces an additional source of pessimism, considering situations that cannot actually occur when tasks are related to the same rotation source. The EDF schedulability of AVR tasks linked to a common rotation source has been analyzed by Biondi and Buttazzo [10], but their test is only sufficient, since derived from a utilization upper bound. Recently, Guo and Baruah [11] studied the EDF scheduling of AVR tasks proposing a speedup factor analysis and sufficient schedulability tests.

An exact feasibility analysis under EDF scheduling for engine control applications including classic periodic tasks and an AVR task and extensive experiments performed by Biondi, Buttazzo and Simoncelli et al. [12] confirmed that for this type of applications, fixed priorities scheduling is not the best choice, due to the large range in which the interarrival time of an AVR task can vary. Under FP scheduling, this means that there are several engine speeds at which any fixed priority assignment is far from being optimal, significantly penalizing the system schedulability (Fig. 1.6).

On the other hand, as shown in Fig. 1.7, under FP, the AVR tardiness is kept equal to zero even for U=1.5, because the overload does not affect the AVR task

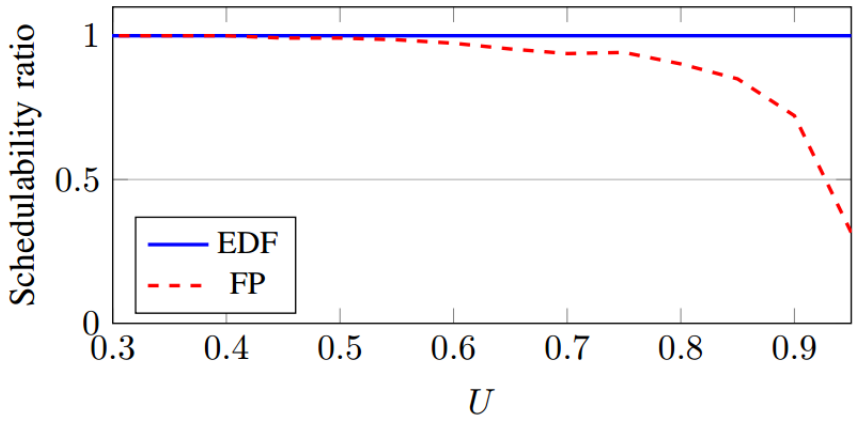

Figure 1.6: Schedulability ratio as a function of U.

(always scheduled at the highest priority), but only the lower priority tasks (the tardiness of the lowest priority task is indicated by the FP-LP curve). Otherwise, under EDF, the AVR tardiness increases with the overload, reaching a value higher than 60 times the deadline for utilizations greater than 1.4. This happens because EDF tends to automatically distribute the exceeding workload to all the tasks, hence the lowest priority task (EDF-LP curve) is not so penalized as in the case of FP scheduling.

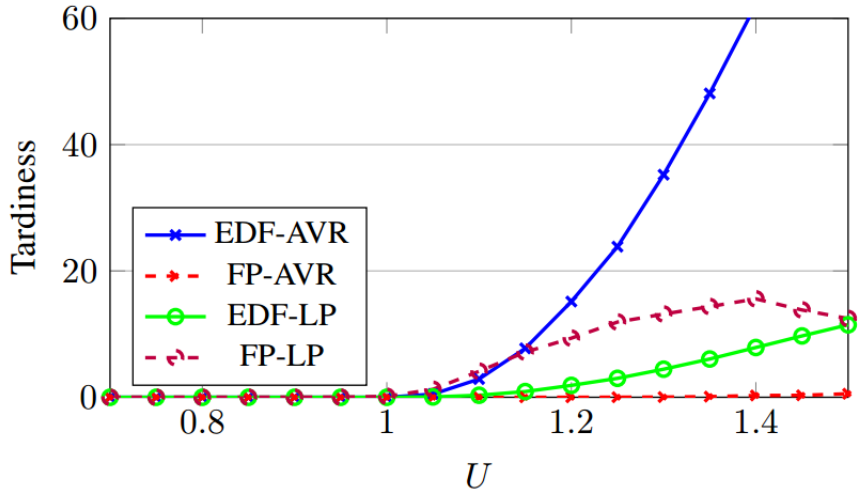

Figure 1.7: Tardiness as a function of the system load U.

The higher schedulability provided by EDF is the main reason that convinced

to develop an efficient support for AVR tasks in a real-time kernel with an EDF scheduler. The kernel selected is ERIKA Enterprise [13], because offers EDF scheduling with an OSEK-like API and a static configuration of the kernel, as mandated by OSEK. However, the native EDF support in ERIKA does not include some features needed to manage engine-triggered tasks.
In this work, the engine is considered as a rotation source that triggers the execution of the AVR tasks at specific angles, in particular it is characterized by the following parameters:
θ the current rotation angle of the crankshaft;
ω the current angular speed of the crankshaft;
α the current angular acceleration of the crankshaft.
The speed ω and the acceleration α are assumed to be limited within a range $[\omega^{min}, \omega^{max}]$ and $[\alpha^{-}, \alpha^{+}]$; in Table 1.1 are reported typical realistic value of such parameters. The software composing an engine control application is modeled

Parameter	min	max
ω (RPM)	500	6500
α (RPM/s)	$-97.2 \cdot 10^2$	$97.2 \cdot 10^2$

Table 1.1: Typical ranges for the engine speed and acceleration

as a set of n real-time preemptive task $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$. Each task can be a standard *periodic* task, or an AVR task, activated at specific crankshaft angles. Both periodic and AVR tasks are characterized by a worst-case execution time (WCET) C_i, an interarrival time (or period) T_i, and a relative deadline D_i. However, while for regular periodic tasks such parameters are fixed, for angular tasks they depend on the engine rotation speed ω. An AVR task is characterized by an angular period Θ_i and an angular phase Φ_i, so that it is activated at the following angles:

$$\theta_i = \Phi_i + k\Theta_i, \qquad for \quad k = 0, 1, 2, ...$$

All angular phases Φ_i are relative to a reference crankshaft position called *Top Dead Center* (TDC) corresponding to the angle for which at least one piston is at the highest position in its cylinder, by convention the TDC position is assumed to be at $\theta = 0$. An angular task is also characterized by a relative *angular deadline*

Δ_i expressed as a fraction of the angular period.
The AVR tasks are typically implemented as a set $\mathcal{M}_i$ of M_i execution modes. Each mode m has a different WCET C_i^m and operates in a predetermined range of engine speeds $(\omega_i^{m+1}, \omega_i^m]$, where $\omega_i^{M_i+1} = \omega^{min}$ and $\omega_i^1 = \omega^{max}$. Hence, the set of modes of a AVR task can be expressed as

$$\mathcal{M}_i = \{(C_i^m, \omega_i^m), m = 1, 2, \ldots, M_i\}$$

We assume that the computation time of a generic AVR task can be expressed as a non-increasing step function C_i of the instantaneous speed ω at its release, that is,

$$C_i(\omega) \in \{C_i^1, \ldots, C_i^{M_i}\}.$$

An example of $C(\omega)$ function is illustrated in Figure 1.8. The worst case analysis

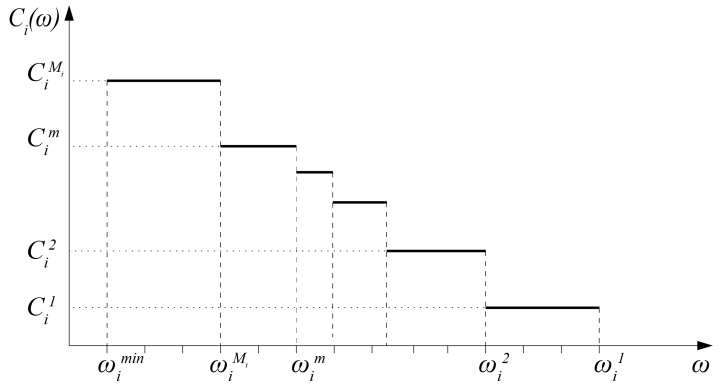

Figure 1.8: Worst-case execution time of an AVR task as a function of the speed at its activation.

performed in [9] shows the case in which the rotation speed ω_i can change over the time, but its acceleration is limited by a maximum value α_i^+. Let t_0 the activation time of a generic AVR task, when the crankshaft rotation angle is equal to θ_0 and its rotation speed is ω_0, the next angle at which the task is triggered again is equal to

$$\theta_1 = \theta_0 + \Delta\theta$$

where $\Delta\theta$ is the angular period.
If the crankshaft rotation speed is constant and equal to ω_0, the rotation angle

$\theta(t)$ will increase linearly as a function of a time as

$$\theta(t) = \theta_0 + \omega_0(t - t_0)$$

Therefor, the activation angle θ_1 will be reached at time

$$t_1 = t_0 + \frac{\Delta\theta}{\omega_0}$$

obtaining an activation period equal to

$$T_i(\omega_0) = t_1 - t_0 = \frac{\Delta\theta}{\omega_0}.$$

However, if α is not equal to zero, the activation period will be shorter than $T_i(\omega_0)$, like is depicted in Figure 1.9. To compute the shortest period $T_i'(\omega_0)$ it

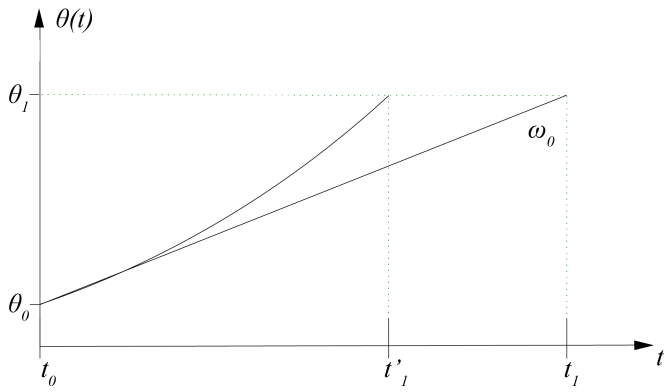

Figure 1.9: Shaft angle as a function of time.

was assumed that $\omega(t)$ can increase at most with a maximum acceleration α^+, therefore by the kinematics equation is obtained the following expression:

$$\omega(t) = \omega_0 + \alpha^+(t - t_0)$$

and the angle $\theta(t)$ will increase as

$$\theta(t) = \theta_0 + \int_{t_0}^{t} \omega(t)dt = \theta_0 + \omega_0(t - t_0) + \frac{\alpha^+}{2}(t - t_0)^2$$

and the value θ_1 will be reached at time t'_1 such that

$$\Delta\theta = \omega_0(t'_1 - t_0) + \frac{\alpha^+}{2}(t'_1 - t_0)^2$$

Hence the shortest activation period can be computed as $T'_i = t'_1 - t_0$ and it is such that:

$$\omega T'_i + \frac{\alpha^+}{2}(T'_i)^2 = \Delta\theta$$

Discarding the negative solution of the equation above, the following formulation of an AVR activation period is found:

$$T'_i(\omega, \alpha^+) = \frac{\sqrt{\omega^2 + 2\Delta\theta\alpha^+} - \omega}{\alpha^+}$$

Figure 1.10 illustrates the activation period T'_1 as a function of ω for different values of α (rad/sec^2) and $\Delta\theta = 2\pi$. This analysis underline that the relative

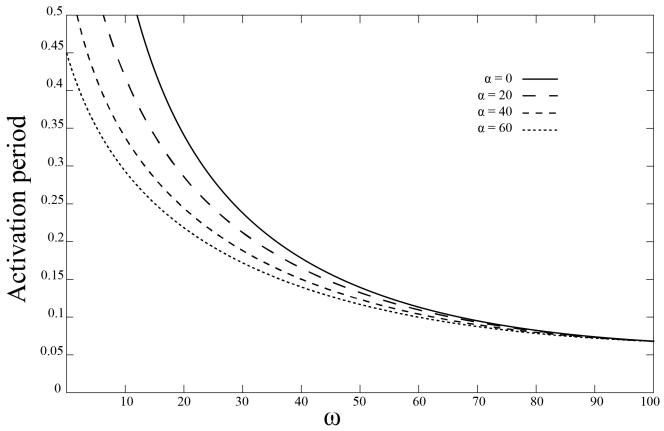

Figure 1.10: Activation period as a function of ω for different values of α (rad/sec^2) and $\Delta\theta = 2\pi$.

deadline of an AVR task is a variable parameter, which is a function of engine state at the release of a job and the future evolution of the rotation source in terms of acceleration. This feature cause to be useless the exact computation of the relative deadline, in fact, all possible values $T(\omega) \in [T(\omega, \alpha^+), T(\omega, \alpha^-)]$ can be interarrival times to the next job (Fig. 1.11). Besides the identified issue, it is anyhow possible to achieve a safe schedule assigning each job the earliest possible deadline among those compatible with the speed at its activation, that is, the one derived assuming the maximum acceleration α^+ from the task release

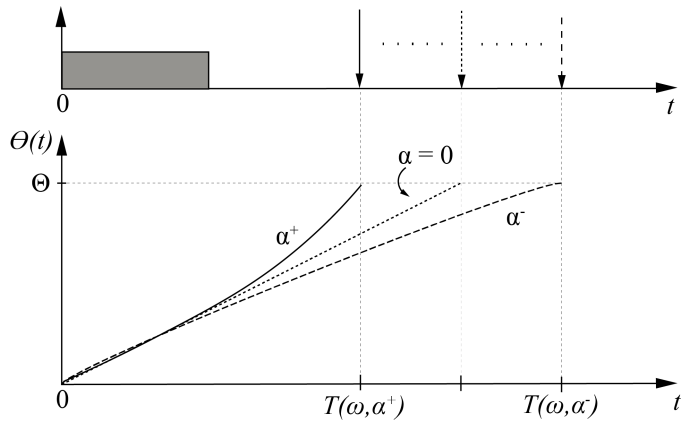

Figure 1.11: Possible deadline of an AVR job activated at speed ω.

on. Using this approximation, the relative deadline of an AVR task released at the instantaneous speed ω results

$$D_i(\omega) = \frac{\sqrt{\omega^2 + 2\Delta_i\alpha^+} - \omega}{\alpha^+}. \tag{1.1}$$

This thesis describes the support implementation to manage the AVR tasks under EDF scheduling in the ERIKA kernel, keeping the standard OSEK API, hence providing the possibility of making a simple integration with existing OSEK applications. To achieve this goal I added some new keywords to OIL configuration language to manage the specific AVR parameters and the new kernel APIs. To test the implemented kernel I used both a hardware device, in particular the STM32F4 microcontroller, and an Cortex instruction set simulator provided by Lauterbach Trace32. In both case I had to implement the necessary plugins; for the microcontroller I implemented the Erika EDF support, whereas for the instruction set simulator I implemented a circular timer, for EDF support, and a rotation source simulator to generate an external interrupt for AVR tasks activation. In order to efficiently implement the relative deadline formula had to be an exhaustive study of a computation time.

Chapter 2

System architecture

This chapter explains both software environments and the hardware devices adopted for developing the kernel support for the AVR tasks.

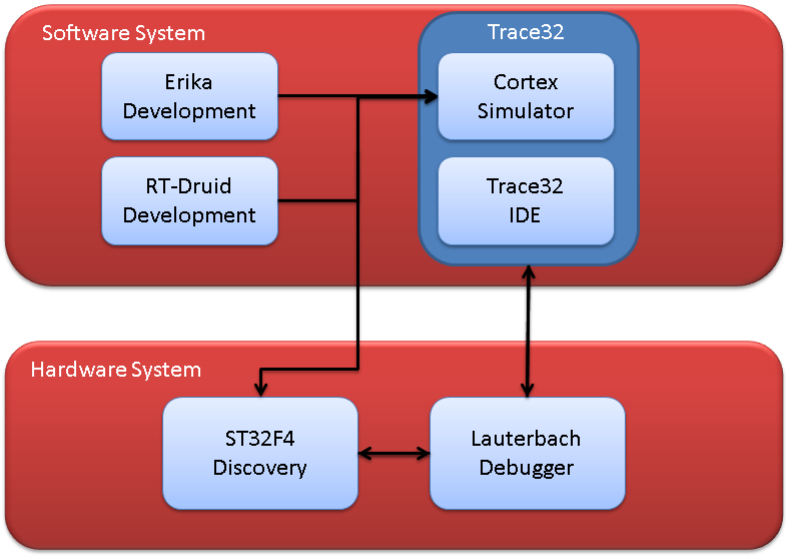

Figure 2.1: System block diagram.

2.1 Software system

2.1.1 Erika

Erika Enterprise [13] is a free of charge open-source OSEK/VDX Hard Real Time Operating System (RTOS), certified as OSEK-compliant, that support a large set of microcontrollers and multi-core platforms. An important feature of this kernel is that allows achieving high predictable timing behavior with a very small run-time overhead and a memory footprint in the order of few kilobytes. Erika supports two typologies of kernel: the OSEK standard [14], with its conformance classes, and other non-standard conformance classes.
The OSEK standard conformance classes are:

- BCC1: supports only basic tasks, which are tasks that release the processor only if they terminate, the RTOS switches to a higher-priority task or an interrupt occurs which causes an interrupt service routine. Moreover the BCC1 supports only one task per priority without multiple requesting of task activation.
- BCC2: extends the BCC1 with the support for multiple requesting of task activation and more than one task per priority.
- ECC1: like BCC1, with the support for the extended tasks, which are distinguished from basic tasks by being allowed to use the operating system call WaitEvent, which may result in a waiting state that allows the processor to be released and to be reassigned to a lower-priority task without the need to terminate the running extended task.
- ECC2: extends the ECC1 with more than one task per priority and supports multiple requesting of task activation only for the basic tasks.

The non-standard conformance classes are the following:

- FP, a minimal implementation of the fixed-priorities scheduling with preemption thresholds [15];
- EDF, that implement the Earliest Deadline First algorithm and the Stack Resource Policy (SRP);

- FRSH, implementing the IRIS [16] scheduling algorithm for resource reservation.
- HR, offering a two-level hierarchical scheduling framework through the M-BROE algorithm [17].

The EDF conformance class is obviously the baseline for the development of the support for the AVR tasks, in fact, in the other OSEK standard conformance classes there is no support to deadline-based scheduling. The implementation relies on a circular timer [8] for managing the internal time representation. The Implicit Circular Timer's Overflow Handler (ICTOH) allows an efficient representation of the absolute deadline with 32-bit variables, containing the runtime overhead and footprint, and provided an infinite system lifetime.
In Erika implementation, the access to mutual exclusive resources is regulated by the the SRPT algorithm, which combine the SRP protocol with Preemption Thresholds to reduce the number of preemptions (and hence reduce the switch context overhead) and save stack space.
Moreover Erika provided a mechanism to handle the interrupt, specified by the OSEK/VDX standard, that are divided in two type:

- Type 1, which not use an operating system service. After the ISR is finished, processing continues exactly at the instruction where the interrupt has occurred, i.e. I/O operations.
- Type 2 that calls the scheduler at the end of the service routine, hence this type of routine can be used to interact with kernel object (i.e. to active a task).

2.1.2 RT-Druid

As specified in OSEK/VDX standard, all the RTOS objects, like tasks, alarms and semaphores, are *static*, this means that all RTOS configuration are predefined at compile time and cannot be changed at run time. This static approach is necessary in order to reduce both footprint and run-time overhead, obtaining a customized RTOS image that is optimized for a specific application-dependent kernel configuration.

In Erika, the objects composing a particular application are specified in the OSEK Implementation Language (OIL) and stored in a proper configuration file. The Erika development environment also includes RT-Druid[18], which is a tool in charge to processing the OIL configuration file to generate the proper Erika code for the requested kernel configuration.
RT-Druid is provided as a plugun of Eclipse IDE, and its main feature is analyze the OIL file and generate three files:

- Makefile, in which are defined the Erika options and the compilation rules and hierarchy;
- eecfg.c, in which are defined all the RTOS objects and the data structures;
- eecfg.h, in which are defined the Erika options and the system configuration constants.

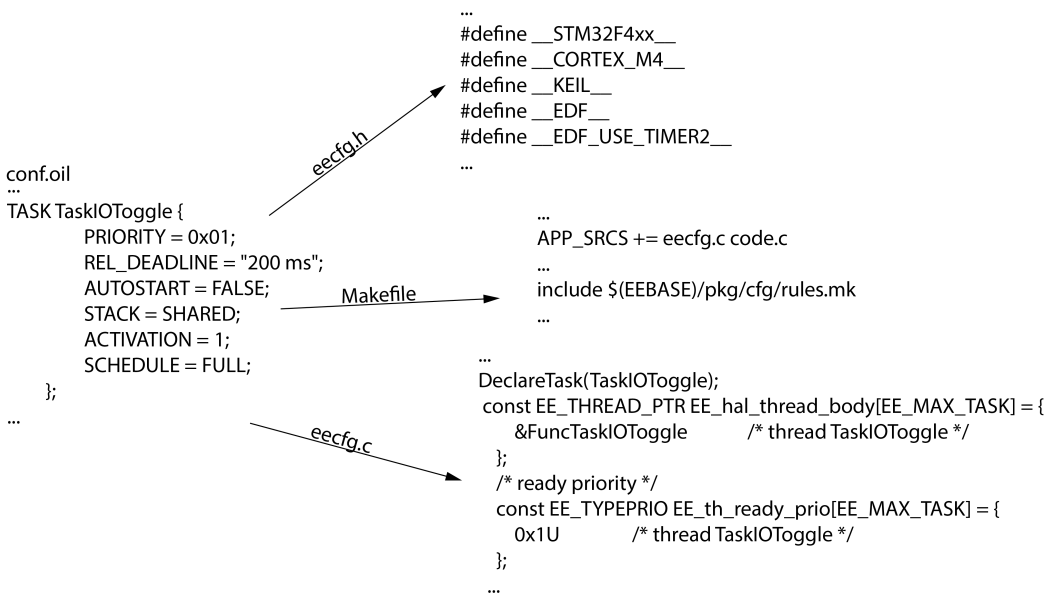

Figure 2.2: Rt-Druid code generator example.

Moreover RT-druid provided other important features for real-time system development:

- **Scheduling analyzer module**: used to perform a worst case timing analysis in according to the selected kernel;

- **Scheduling analyzer module**: for those systems where the evaluation of the worst case response time of the threads needs to be supplemented by simulations of average as well as critical runs highlighting the expected time behavior of the system.

- **Trace viewer/analyzer**: providing support for estimating the execution time attributes of software components. A graphical front-end is able to display the timing properties of the system, letting the user understand the run-time application behavior and enabling useful back-annotation that will help in a better system characterization. Input can be taken from a simulated trace and from real execution on the target microcontroller.

2.1.3 Cortex simulator

The Instruction Set simulator supports a large number of microcontrollers and it is included in the Trace32 suite, distributed for free by Lauterbach GmbH, the world's larger producer of hardware assisted debug tools for microprocessors. This simulator allows to execute real code collecting a large set of debug and trace information without having any hardware devices, enabling the possibility to build very powerful testing and development environment.
The main limitation of the Trace32 suite is that it only offers the instruction set simulator for the CPU of a microcontroller and not the simulation of its peripherals devices (e.g. timers and interrupt controller). However, Trace32 offers a full memory map of the microcontrollers and a standard interface, called Peripherals Simulation Model (PSM)[19], that allows developing custom simulated peripherals. PSM is a software overlay for memory area occupied by peripheral modules and it provides functions for the interaction between the cpu and the other modules. Through the PSM a developer can implement a custom software that reacts and access to specific registers located in physical memory of the simulated microcontroller.
The simulator, every clock cycle, records all bit's changes, both in cpu registers and in the physical memory, in this way Trace32 can collect all tracing and statistical data such as function time computation. This mean that to run approximately six seconds of simulation are needed a significant amount of memory

(up to 25 GBs), meanwhile to store, only the trace data, are needed up to 4 GBs of hard disk.
The simulator is managed through the Trace32 PowerView IDE, an universal user interface for all microprocessor development tools from Lauterbach, which offers intuitive, consistent and fast access to debug and trace information.
This IDE is fully user-adjustable through the PRACTICE scripting language, that allows develop both interactive and automatic programs to set up the IDE and simulator environment and to support the debugging process.

2.2 Hardware devices

2.2.1 STM32

The STM32 is a is a family of 32-bit microcontroller integrated circuits by STMicroelectronics. In particular it has been used the Discovery Board, with the STM32F407 microcontroller which have an 32-bit ARM Cortex-M4 CPU with FPU up to 168HMz, 1 MB flash and 192 KB of RAM, including an ST-LINK/V2 embedded debug tool, two digital accelerometer and digital microphone, one audio DAC with integrated class D speaker driver, leds and push buttons and an USB OTG micro-AB connector.

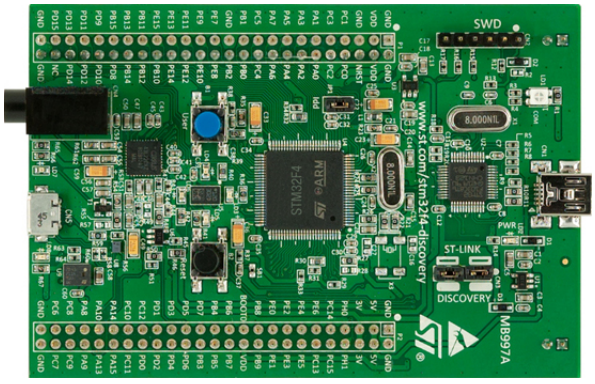

Figure 2.3: STM32F4 Discovery.

The Erika kernel EDF scheduler implementation, as specified in 2.1.1, required a free runner timer, the Discovery board provided three kind of timers:

- General-purpose timer:

  - 16-bit or 32-bit auto-reload counter in up, down or up/down mode.
  - 16-bit programmable prescaler used to divide the counter clock frequency.
  - synchronization circuit to control the timer with external signals and to interconnect several timers.
  - interrupt/DMA generation on some event (e.g. counter overflow/underflow, counter initialization, trigger event, Output compare)
  - supports incremental encoder and hall-sensor circuitry for positioning purpose
  - trigger input for external clock or cycle-by-cycle current management

- Advanced-control timer,only in 16-bit version, that in addition to the general purpose feature, provide:
  - complementary outputs
  - repetition counter to update the timer registers only after a given number of cycles of the counter
  - break input to put the timer's output signals in reset state or in a know state.

- Basic timer
  - 16-bit auto-reload upcounter
  - 16-bit prescaler
  - synchronization circuit to trigger the Digital Analogic Converter
  - interrupt/DMA generation on the counter overflow event.

The type chosen for the implementation is a 32-bit general purpose timer, in this way with a clock frequency set to 168 MHz (hence the timer clock at 84 MHz by hardware constraints), the maximum difference between two different tasks is 25,5 seconds, that is an acceptable value for the common automotive applications.

2.2.2 Lauterbach CombiProbe

The CombiProbe (Fig. 2.4) is a hardware tool provided by Lauterbach for debugging microcontroller applications. Moreover it provides particular con-

Figure 2.4: CombiProbe Hardware.

figuration commands for displaying and analyzing recorded trace information. System information can be recorded into the trace memory of the CombiProbe or it can be sent directly to the host, using the CombiProbe memory as a buffer. The trace information can be analyzed with the Trace32 PowerView IDE. Actually the CombiProbe provides the standard JTAG and the JTAG support for ARM cores including the Serial Wire Debug Port, it also provided the proper cable adapters for supporting a widest range of microcontrollers. The STM32 does not provide a standard debug connector, hence a homemade cable has been done to adapt the MIPI20 connector (that is the standard ARM debug connector) to the pins of the STM32. CombiProbe is connected to the host through an ethernet cable and they communicate over the TCP/IP protocol.

Chapter 3

Support for EDF scheduling for AVR tasks

This chapter reports the detailed explanation of the both extension on the Erika is kernel and on RT-Druid tool for supporting the engine-triggered tasks. Moreover there is a special focus on the implementation of the relative deadline formula.

3.1 Erika support for STM32

To execute the EDF scheduler on the STM32F4 Discovery board it was necessary to implement two functions to initialize and to read the free runner timer. According to the Erika tree structure these two functions are implemented in the ee_mcu.h file included in the STM32F4 folder of the Erika repository. The added functions are:

- Timer initialization function, shown in Lst.3.1, used to set the timer in up mode with prescaler value set to zero. It is used once in the Erika initialization section of the application main method.
- Get time function, shown in Lst.3.2, used to retrieve the timer counter value. This function is used in the activation task method to compute the absolute deadline, that it is compute by adding the counter value to the relative deadline.

In both functions, it was used the *volatile* type, in this way it is possible to access directly to the registry address, avoiding a compiler optimization which may compromise the correct register set up. The STM32F4 board provided two 32-bit timers, whose selection is made by user setting a keyword in the microcontroller section of the OIL file. The keyword is translated in a C macro that distinguishes which version should be used.

```
#ifdef __EDF_USE_TIMER2__
__INLINE__ void __ALWAYS_INLINE__ EE_time_init(void){
        volatile uint32_t* apb1enr = (uint32_t *)0x40023840;
        volatile uint16_t* cr1 =     (uint16_t *)0x40000000;
        volatile uint32_t* arr =     (uint32_t *)0x4000002C;
        volatile uint16_t* psc =     (uint16_t *)0x40000028;
        *apb1enr |= 0x00000001;
        *cr1 &= 0xFC8F;
        *arr = 0xFFFFFFFF;
        *psc = 0x0;
        *cr1 |= 0x0001;
}
#endif
```

Listing 3.1: Timer 2 initialization function

```
#ifdef __EDF_USE_TIMER2__
__INLINE__ EE_TIME __ALWAYS_INLINE__ EE_hal_gettime(void)
{
        volatile uint32_t* cnt = (uint32_t *)0x40000024;
        return *cnt;
}
#endif
```

Listing 3.2: Get time function for timer 2

3.2 Support for AVR tasks in the Erika kernel

In the standard AUTOSAR/OSEK API the task activation is performed with the following system call

`ActivateTask (TaskType TaskID)`,

where `TaskID` represents the identifier of the task for which a job has to be

activated. To keep the API OSEK-compliant, the same system call has been held in the Erika EDF kernel. Such a call computes the absolute deadline for each upcoming job by using the constant relative deadline.
As specified in the Equation 1.1, however, the relative deadline of an AVR task depends on the current engine speed ω, hence determining the need for creating a new system call to activate this kind of tasks:

`ActivateTask (TaskType TaskID,SpeedType w)`,

where `w` represent the current engine speed at the time at which the task is activated.
Regarding the *SpeedType* two measurement units are considered:

- *revolutions/ticks* (RPTicks), expressed by a floating point value.
- RPM, expressed by an integer value;

This choice was made considering that the notion of current engine speed, also used to trigger the mode change in an AVR task, is typically maintained in the engine control applications through some technique that can be summed in two scenarios. The first one is a typical case of use of an estimation technique such as an average speed in some time is intervals [5], in which the engine speed is measured by using a temporal reference of the microcontroller, like an internal timer, producing a floating point number. The second one, covers the situation in which the engine speed is provided by an external device, know as Time Processing Unit (TPU), commonly provided by microcontrollers designed for automotive applications. An example is described in [20], where the microcontroller manufacturer provides a firmware library for the TPU returning the engine speed in RPM as 32-bit integer value.
The new `ActviteTask` system call is in charge of computing the current relative deadline $D_i(\omega)$ expressed in Eq. 1.1, so obtaining the absolute deadline $d_i = t + D_i(\omega)$, where t is the current time at which `ActviteTask` is invoked. Once the absolute deadline is computed, all the existing code in ERIKA for managing task activations under the EDF conformance class can be reused. Hence, it is obvious that the key challenge for supporting AVR tasks is to provide an efficient implementation of Eq. 1.1 and manage the additional data structures needed to store the AVR task parameters.

3.2.1 Deadline computation

Considering the unnecessary run-time overhead at each task activation produced by the inefficient implementation originated by the use of the native square root function available in the standard C mathematical library, two alternative approaches are investigated and compared for implementing the deadline formula:

1. The first approach, aspire to reduce the impact in term of footprint, uses an iterative computation method to approximate the value of the square root, but on the other hand the run-time overhead of the `ActivationTask` system call is incremented (resulting in any case less than the standard C square root);

2. The second approach, aspire to reduce the run-time overhead, uses a lookup table to store a set of deadline values with a given speed step, but it significantly increases the footprint, which becomes dependent on the required resolution and the number of AVR tasks.

Fast square root approach: The OSEK RTOSes are static, this mean that no tasks can be created at run-time, hence all the tasks properties are well-know at compile time. For this reason a part of the Eq. 1.1 can be pre-computed at compile time, thus reducing the amount of computation needed at run-time. Therefore, the following parameters are defined for each AVR task τ_i^*:

$$K_i^{(a)} = \frac{1}{\alpha^+}$$

and

$$K_i^{(b)} = 2\Delta_i\alpha^+$$

Using $K_i^{(a)}$ and $K_i^{(b)}$, the Eq 1.1 can be rewritten as

$$D_i(\omega) = (\sqrt{\omega^2 + K_i^{(b)}} - \omega)K_i^{(a)}, \tag{3.1}$$

thus avoiding one division and one multiplication at run-time.
To store this two parameters are needed two additional data structures

```
TypeAVRParams K_A MAX_AVR_TASKS;
TypeAVRParams K_B MAX_AVR_TASKS;
```

where `MAX_AVR_TASK` represents the number of AVR tasks in the task set and `TypeAVRParams` is a floating point data type.
The most critical part of Eq.3.1, in term of overhead time, is the computation of the square root. Therefore it was implemented following the FastSQRT algorithm [21], that provides a fast computation of an approximation of the square root (Lst.3.3)

```
float  fast_sqrt(float x){
        float xhalf = 0.5f*x;
        union
        {
                float x;
                int i;
        } u;
        u.x = x;
        u.i = 0x5f3759df - (u.i >> 1);
        u.x=u.x*(1.5f-(u.x*u.x*xhalf));
        return x*u.x*(1.5f-(u.x*u.x*xhalf));
}
```

Listing 3.3: FastSqrt algorithm implementation

Such an algorithm relies on the Newton is method for iteratively computing the square root of a number, but improves on its original formulation by selecting an initial value for the iteration that is able to considerably reduce the error with only two iterations. The initial value for the iteration is computed by means of a particular constant (called Magic Number) which as been formally studied in [21].

A numerical study it was performed to evaluate the error of the deadline computation by using the FastSQRT algorithm with the respect to the exact square root function. Such a study was performed considering all the speeds in a typical range for a production car engine, that is from 500 to 6500 RPM, and implementing the functions using both integer and floating point values in according with the measurement units of the engine speed considered in this work. The results, reported in Fig. 3.1, shows that the FastSQRT algorithm present an error always lower than 0.04%. In particular the error is lower for rev/tick

because the values of the engine speed, expressed in this measurement unit, are always less than one (e.g. 6500 RPM = 0.000001289 rev/ticks for a tick time equals to 11.9ns) and the error function, defined as *estimation values - exact values*, have an increasing oscillation trend (3.2); moreover it is always negative, this means that the deadline computed by the FastSQRT is early in respect the exact one, hence this feature adding an other kind of pessimism resulting that the system is always in safe state.

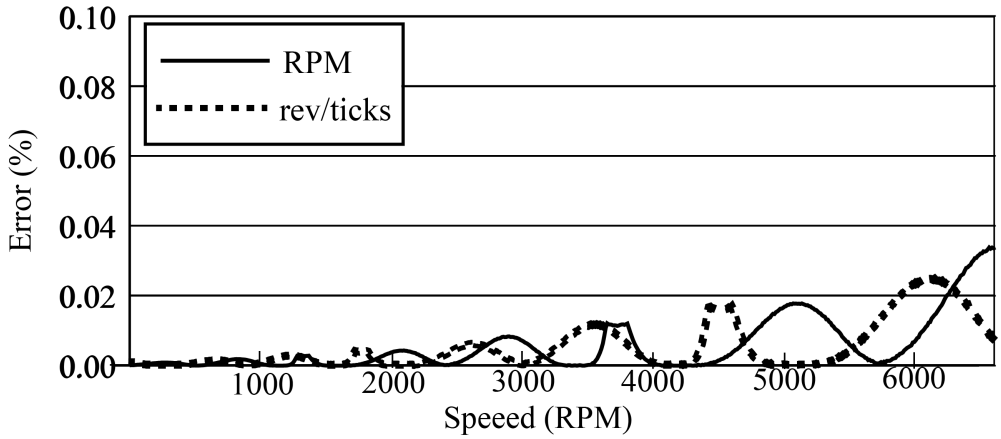

Figure 3.1: Error of the FastSQRT algorithm.

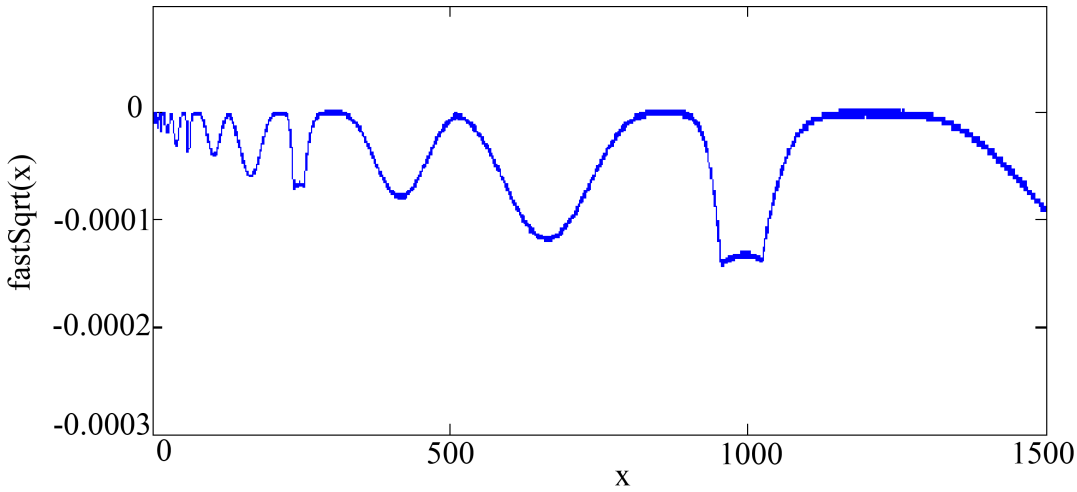

Figure 3.2: Error of the FastSQRT.

Lookup Table approach: The second approach considered in this work for computing the deadline of AVR tasks relies on an off-line pre-computation of Equation

1.1 for the whole range of engine speeds. The values of Equation 1.1 are computed off-line by using a quantization step $\Delta\omega$ and stored as constants in an array whose elements are denoted as $W_j, j = 0, 1, \ldots, \left\lceil \frac{\omega^{max} - \omega^{min}}{\Delta\omega} \right\rceil$, so obtaining a lookup table. At run-time, a weighted average is then used to estimate the value of the deadline. More specifically, suppose to have a current engine speed ω and let $Y = \frac{\omega - \omega^{min}}{\Delta\omega}$. An index j for the look table is computed as $j = \lfloor Y \rfloor$ and the value $D(\omega)$ is obtained as

$$D(\omega) = \frac{qW_j + lW_{j+1}}{q + l}, \tag{3.2}$$

where q and l represent the weights of the weighted average and their definition depends of the measurement unit used to represent the engine speed ω. When the speed is represented in RPM (as an integer value) we defined $q = \Delta\omega - l$ and $l = (\omega - \omega^{min}) - j\Delta\omega$, leading to $q + l = \Delta\omega$, where l can be computed by means of a modulo operation. In this way, if $\Delta\omega$ is defined as a power of two, the value of $D(\omega)$ can be efficiently computed without involving any floating point operation implementing the ratio as a bit shift operation. On the other hand, when the speed ω is represented in revolutions/ticks (as a floating value), different formulations for q and l can be defined to optimize the computation. In fact, defining $q = 1 - l$ and $l = Y - j$, we obtain $q + l = 1$, so avoiding the division in Equation 3.2. In addition, the value of l can be computed by a truncation operation by a casting to an integer value. Under this conditions, the value of $\Delta\omega$ results crucial for determining the precision and the additional footprint imposed by this approach. Table 3.1 reports a numerical evaluation of both error and footprint for different values of $\Delta\omega$. Since deadlines in ERIKA are represented in ticks, each element of the lookup table can be represented as a 32-bit integer value leading to a cost of 4 bytes each, independently from the measurement units specified. Overall, although a considerably small footprint is required for achieving a good precision (e.g. with $\Delta\omega = 256RPM$), the main drawback of this approach is that (considering that in the general case of AVR tasks having all different angular deadlines) a lookup table has to be generated for each tasks, so limiting the scaling for high number of tasks on memory-constrained platforms.

$\Delta\omega$ (RPM)	AVG err (%)	MAX err (%)	Footprint (bytes)
32	0.002	0.013	752
64	0.009	0.05	376
128	0.036	0.2	188
256	0.145	0.79	96
512	0.58	2.99	48
1024	2.36	10.493	24

Table 3.1: Percentage of error and footprint for the lookup table approach under different values of $\Delta\omega$.

RPM			
	FastSQRT	*Lookup Table*	*SQRT*
MAX	2.23 μs 188 cycles	0.3 μs 26 cycles	5.42 μs 456 cycles
AVG	2.08 μs 176 cycles	0.25 μs 22 cycles	5.09 μs 428 cycles

revolutions/ticks			
	FastSQRT	*Lookup Table*	*SQRT*
MAX	1.69 μs 143 cycles	1.9 μs 164 cycles	5.21 μs 438 cycles
AVG	1.51 μs 127 cycles	1.55 μs 131 cycles	4.89 μs 411 cycles

Table 3.2: Run-time comparison of the FastSQRT and Lookup table approaches for computing the deadline of AVR tasks.

Moreover, a comparative study it was performed on the time overhead for both approaches, together with one of using the SQRT function of the standard C library. The experiments results, obtained on a STM32F4 microcontroller running at 168Mhz with FPU enabled and reported in the Table 3.2, shows a significant improvement of the FastSQRT on the standard SQRT function, halving the run-time in case of representation in RPM and being one-third in case of using revolution/ticks. This time difference can be better understood by analyzing the implementation of the functions. If the RPM is used as representation of the engine speed (Lst.3.4), the measurement unit of the computed deadline is *minute*, hence an other multiplication is necessary to convert *minute* in *tick* (the system constant value RATIO represent the scalar conversion factor); on the other hand, in the case where it is used the rev/ticks unit (Lst.3.5), the deadline is already expressed in *tick*, hence it is enough to return the computed

value without any conversions.

```
__INLINE__ EE_TYPERELDLINE __ALWAYS_INLINE__ computeAvrDeadline(EE_TID t,
   SpeedType omega){
        float dl;
        dl = ((fast_sqrt((omega*omega)+(EE_th_deltatimesalpha[EE_th_avrindex[t
           ]]))-omega)*EE_th_alphamaxinverse[EE_th_avrindex[t]]);
        return dl * RATIO;
}
```

Listing 3.4: Deadline computation using FastSQRT and RPM unit

```
__INLINE__ EE_TYPERELDLINE __ALWAYS_INLINE__ computeAvrDeadline(EE_TID t,
   SpeedType omega){
        return ((fast_sqrt((omega*omega)+(EE_th_deltatimesalpha[EE_th_avrindex[t
           ]]))-omega)*EE_th_alphamaxinverse[EE_th_avrindex[t]]);
}
```

Listing 3.5: Deadline computation using FastSQRT and rev/ticks unit

Moreover, observing Table 3.2, the lookup table approach resulted very efficient for the RPM case (Lst.3.6), where no floating point operations are involved, while resulted comparable to the FastSQRT in case of speed representation in revolutions/tick (Lst.3.7), due to the floating point operation.

```
__INLINE__ EE_TYPERELDLINE __ALWAYS_INLINE__ computeAvrDeadline(EE_TID t,
   SpeedType omega){
        int index;
        int alpha;
        int beta;
        int base=(1 << STEP);
        index=(omega-OMEGA_MIN) >> STEP;
        beta=(omega-OMEGA_MIN) % base;
        alpha=base-beta;
        return (alpha*(*(EE_tab_address+EE_th_avrindex[t]))[index])+(beta*(*(
           EE_tab_address+EE_th_avrindex[t]))[index+1]) >> STEP;
}
```

Listing 3.6: Deadline computation using lookup table and RPM unit

```
__INLINE__ EE_TYPERELDLINE __ALWAYS_INLINE__ computeAvrDeadline(EE_TID t,
   SpeedType omega){
        float beta;
```

```
        int index;
        float alpha;
        beta = ((omega-OMEGA_MIN)/STEP);
        index=(int)beta;
        beta=beta-index;
        alpha = 1-beta;
        return ((alpha * (*(EE_tab_address+EE_th_avrindex[t]))[index]) +(beta *
           (*(EE_tab_address+EE_th_avrindex[t]))[index+1]));
}
```

Listing 3.7: Deadline computation using lookup table and rev/ticks unit

From the above analysis, it is impossible to determine which solution is the best since it is depend on which measurement unit is adopted and which constraints are critical for the application (the timing constraints or the footprint), hence it was decided to implement both versions. The choice of which one to use can be made or explicitly by the users or automatically by RT-Druid, in both case new keywords are added to the standard OIL language for supporting this features. Some features of the C language are used for improving the kernel performances. For instance, the functions were implemented as *inline*, to speed up the execution of the task activation function, avoiding a function call. Moreover, each function is defined inside the preprocessor statement `#ifdef`. The latter design solution allows to achieve two important goals:

- In according with the kernel configuration, only one function is defined at compile time, in this way the footprint does not increase.
- The same function signature (function name and input arguments) can be used for all methods for computing the deadline.

With this approach, the new system call for activating AVR tasks is implemented in very simple way. In fat, the activate task function takes as input arguments the task id and the engine speed, computes the relative deadline and calls the common Erika activation task function (Lst.3.8). In this way, the existing code can be reused.

```
void EE_edf_ActivateTask_AVR(EE_TID t,SpeedType omega){
        EE_TYPERELDLINE dline=computeAvrDeadline(t,omega);
        EE_edf_ActivateTask_Dline(t,dline);
}
```

Listing 3.8: ActivateTaskAvr system call

To support the AVR tasks, new data structures were defined, depending on the chosen method. The data structures defined when the lookup table method is selected are shown in Lst.3.9

- An array of integers, for each distinct task, to represent the lookup table, whose dimension depends on quantization step;
- An array of address, which contains the lookup tables address;
- The system's constant to compute the deadline (e.g. quantization step).

```
...
const EE_TYPERELDLINE EE_th_lookuptable0[25] = {
3772843,
2887725,
...
};
...
const EE_TYPERELDLINE *EE_tab_address[2]={
EE_th_lookuptable0,
EE_th_lookuptable1
};
...
```

Listing 3.9: Data structures defined for the lookup table

The array of address is needed because, as shown in the fragment code 3.7 and 3.6, to access the lookup table are used its address. The data structures defined when the FastSQRT method is selected are shown in Lst.3.10):

- An array of floating points, represent the precomputed value of $2\Delta\alpha^{+}$, with dimension equals to the number of distinct task;
- An array of floating points, represent the value of $\frac{1}{\alpha^{+}}$, with dimension equals to the number of distinct task.

```
...
 const float EE_th_deltatimesalpha[2] = {
        0.00000000000002294082,
        0.00000000000002039184
    };
...
 const float EE_th_alphamaxinverse[2] = {
        43590420917822.470000,
        43590420917822.470000
    };
...
```

Listing 3.10: Data structures defined for the fastSQRT

In any case, an array of 8-bit integer was defined to allow accessing the AVR data structures (Lst.3.11).

```
    const uint8_t EE_th_avrindex[EE_MAX_TASK] = {
        0,      /* thread Task_AVR1 */
        -1,     /* thread Task_Periodic */
        1       /* thread Task_AVR2 */
    };
```

Listing 3.11: Array of index for AVR tasks

3.3 Support for OIL language

AUTOSAR/OSEK RTOSes, and so on Erika, are configured at compile time through a specific language named OIL. RT-Druid, as illustrated in 2.1.2, is a Java tool, provided together with Erika, to configure the Erika kernel starting form an OIL file.
To support and configure the kernel, some new keyword were added to the standard OIL in order to support the definition of the AVR tasks, standard OIL specifications used for the tasks has been extended by adding a special construct, called `AVR_TASK`, which contain the following field:

- `ALPHA_MAX`, containing the maximum angular acceleration for the rotation source triggering the AVR task;

- `ANG_DEADLINE`, containing the angular deadline Δ of the task.

An example of use is shown in the Lst.3.12, where it was used the rev/ms^2 to represent the maximum acceleration and *degrees* to represent the angular deadline, moreover other different measurement units are available for representing the values of such fields.

```
TASK sampleAvrTask{
            AVR_TASK=TRUE{
                    ALPHA_MAX = 0.000162 RPms2;
                    ANG_DEADLINE = 180 degrees;
            };
            ...
};
```

Listing 3.12: AVR Task definition in the OIL file

Since the kernel implementation is able to handle the speed representation both in RPM and in rev/tick, an OIL field named `SPEED_TYPE`, is provided in the kernel section for selecting among these two options, this field can assume only two values:

- RPM: the kernel is configured, as default option, for using the lookup tables to compute AVR task deadlines with a default quantization step equals to 256 RPM.
- RPTICK: in this case the FastSQRT is used as default option.

Moreover, an additional kernel configuration OIL struct is added for enforcing the use of the FastSQRT algorithm or lookup tables with different quantization steps $\Delta\omega$. To avoid any user error in selecting the proper quantization step, this field has been configured with some predefined quantization values to guarantee the proper behavior of the lookup tables for both speed representations. So, all the predefined quantization values are in the powers of two. In this way the IDE shows only the available values, as show in Fig. 3.3.

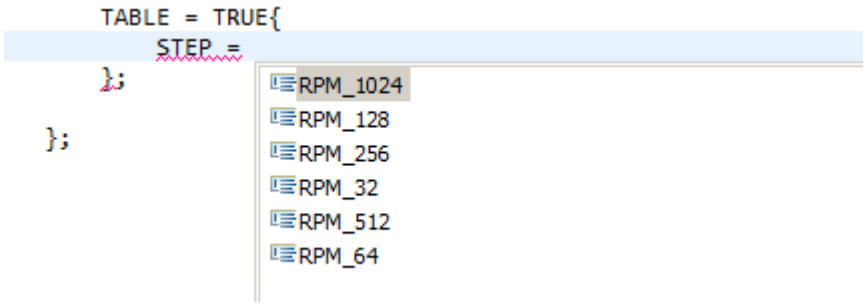

Figure 3.3: Predefined quantization step values.

The RT-Druid tool has been modified to support the extended specification. In particular, the EDF scheduler package has been modified by adding new features to properly configure the options and the system calls of such a kernel. Hence, supposed that the EDF scheduler has been selected, in the OIL file, the first step of RT-Druid is to check the definition of the `SPEED_TYPE` and `TABLE` fields, and define the proper macro in the Makefile and in the eecfg.h file. RT-Druid provides default values for these fields, in this way there are four possible scenarios for configuring the kernel:

1. No fields are defined: RT-Druid automatically configures the kernel for using the FastSQRT algorithm and the *rev/tick* as angular speed representation.

2. SpeedType field defined: if RPM is selected, the tool set at TRUE the value of the TABLE field and set the quantization step equals to 256 RPM, otherwise, if RPTICK is selected the TABLE field is set to FALSE, for using the FastSQRT algorithm;

3. Table field defined: if TRUE is selected the kernel is configured to use the RPM for representing the angular speed, otherwise, the kernel is configured for using *rev/tick* measurement unit;

4. Both field are defined: in this case the user can customize the kernel selecting among the possible combinations.

Subsequently RT-Druid checks if at least one AVR task is specified, and, in this case, defines the macro `__EDF_AVR__`, in the Makefile and in the eecfg.h file.

With this macro the compiler includes the files that defines the activation task function for the AVR tasks. If no AVR task is defined, `SPEED_TYPE` and `TABLE` fields are ignored. For each AVR task, the tool extracts the values specified in the `ANG_DEADLINE` and `ALPHA_MAX` fields and check if their measurement units are correct, reporting an error state if they are not. These values are used by RT-Druid to generate the proper data structures in according with the kernel configuration, therefore if FastSQRT is selected, such values are used to precompute the parameters $K_i^{(a)}$ and $K_i^{(b)}$, of the Eq. 3.1, and to fill the data structures `EE_th_deltatimesalpha` and `EE_th_alphamaxinverse`, in eecfg.c file, with their corresponding values. The Lst.3.13 shows a fragment of the RT-Druid code for reading the OIL fields and for creating the data structures.

```
...
String a=currTask.getString(EDF_TASK_AVR_OIL_ANG_DEADLINE);
double ang=currTask.getDouble(EDF_TASK_AVR_ANG_DEADLINE_REVOLUTION_DOUBLE);
double alphaMax=currTask.getDouble(EDF_TASK_AVR_ALPHA_MAX_RPMTICK2_DOUBLE);
float k_a = 2*ang*alphaMax;
float k_b = (1/alphaMax);
sbAngDLThread.append(preavr + postavr + indent2 + String.format("%.20f",k_a));
sbInverseAlphaMax.append(preavr + postavr + indent2 + String.format("%f", k_b));
...
```

Listing 3.13: Creation of the data structures

On the other hand, if the kernel is configured to create the lookup tables, the `ANG_DEADLINE` and `ALPHA_MAX` fields of each task are used to compute the values of the corresponding lookup table. A lookup table is filled considering that the *k-th* element of the table is equals to

$$D(\omega_k) \quad k = 0, 1, ..., k_{max}$$

where the function $D(\cdot)$ is the Eq. 1.1, $\omega_k = \omega_{k-1} + \Delta\omega$, $\omega_0 = \omega_{min}$, k_{max} is the index for which $\omega_{k_{max}-1} + \Delta\omega \geqslant \omega_{max}$ and $\Delta\omega$ is the quantization step. The Java code for filling the lookup table is shows in Lst.3.14

```
...
for(int omega=omega_min;omega<=omega_max+step;omega=omega+step){
        int dl=(int)Math.ceil((((Math.sqrt((omega*omega)+2*alpha*angDL))-omega)/
           alpha));
```

```
        buffer.append(pre+dl+post);
        pre=",\n";
}
...
```

Listing 3.14: Computation of the lookup table elements

The main drawback of this approach is that the footprint may considerably increase, since for each task, in according with the selected kernel configuration, or a lookup table or an element in both array is created. On the other hand, considering that some AVR tasks might have the same angular deadline and same angular acceleration (e.g. the scenario in which two or more tasks are triggered by the same rotation source), a special java class has been added to RT-Druid package, for recognizing this type of tasks, and for creating only one data structure for all of them. The Lst.3.15 shows the part of the java code for recognizing if a task with the same angular properties has already been processed.

```
...
boolean found=false;
for (int i = 0; i < taskSet.size(); i++) {
        if(taskSet.elementAt(i).checkParam(currentTask)){
                currentTask.index=taskSet.elementAt(i).getIndex();
                currentTask.duplicate=true;
                found=true;
                taskSet.add(currentTask);
                break;
                }
        }
        if(!found){
                currentTask.index=avr_index;
                avr_index++;
                taskSet.add(currentTask);
        }
...
```

Listing 3.15: Found duplicate properties

The `taskSet` object is an array of AvrTask objects, a special data type created for storing some task information such as the angular properties. The Lst.3.16 shows the implementation of the AvrTask class.

```
public class AvrTask{
```

```
        private String oil_ang_deadline;
        private String oil_alpha_max;
        private double alpha_max;
        private double ang_dl;
        private String task_name;
        private int index;
        private boolean duplicate;
        public AvrTask(String ang_deadline,String alpha_max,String name){
                this.oil_alpha_max=alpha_max;
                this.oil_ang_deadline=ang_deadline;
                this.task_name=name;
                this.duplicate=false;
        }
        public AvrTask(String name){
                this.task_name=name;
        }
        public boolean checkParam(AvrTAsk t){
                return (this.oil_ang_deadline.equals(t.oil_ang_deadline) && this
                   .oil_alpha_max.equals(t.oil_alpha_max));
        }
}
```

Listing 3.16: AvrTask class

3.3.1 Stack Resource Policy support

Erika kernel provides the SRP protocol for accessing shared resources when using EDF. This is a priority-based protocol that assigns each task a priority level. These priorities are assigned in the OIL file through the `PRIORITY` field. This field is used for filling two operating system data structures, namely `EE_th_ready_prio` and `EE_th_dispatch_prio`. With this approach, the responsibility to set the proper priorities, is assigned to the user, so it might be wrong assignments causing a malfunction of EDF, for instance a task with a longest deadline may be activated before a task with a shortest deadline even if no shared resources are defined.
To avoid this behavior, a new OIL keyword, `TASK_PRIORITY_ASSIGNMENT`, has been added in the kernel configuration section. This field can be set with two values:

- `MANUAL`: priorities set by the user.

- `DEADLINE_MONOTONIC`: priorities redefined following the *deadline monotonic* priority assignment policy. With this approach the priority is inversely proportional to the relative deadline length (the higher priority task is the task with the early deadline).

`PRIORITY` is a required field, so exploiting this feature, the RT-Druid was modified to redefine the priorities values for reusing the existing code to create the data structures. In this way RT-Druid is able to handle two scenarios under `DEADLINE_MONOTONIC` kernel configuration:

1. No resource defined: all task priorities are set to zero, in this way the proper behavior of EDF is guaranteed.

2. One or more resources defined: RT-Druid analyzed the relative deadline of each task; for the AVR tasks, it computes the worst case deadline, corresponding to the case of maximum angular speed; tasks are sorted by deadline and the proper priority is assigned to each task.

```
Arrays.sort(taskDL_sorted);
for(int i=0;i<tasksDL.size();i++){
        int prio=-1;
        for(int j=0;j<taskDL_sorted.length;j++){
                if(taskDL_sorted[taskDL_sorted.length-1-j]==taskDL[i]){
                        prio=j;
                        break;
                }
        }
        prio= 1 << prio;
        ISimpleGenRes currTask=tasks.get(i);
        currTask.removeAProperty(ISimpleGenResKeywords.TASK_READY_PRIORITY);
        currTask.setProperty(ISimpleGenResKeywords.TASK_READY_PRIORITY,""+prio);
}
```

Listing 3.17: Priorities reconfiguration

The listing 3.17 implements the operations for assigning the proper priority to the each tasks. The `taskDL` array contains the task relative deadline without any order (in the order as they are defined in the OIL file). The `taskDL_sorted` array contains, through the java static method `Arrays.sort(int[])`, the task relative deadlines into ascending numerical order (the earliest one is the first

element of the array, hence the last element of the array represent the lower priority task). In this way the priorities are represented by the position in the array. The outer `for loop` scans the `taskDL` array and the inner one scans backwards the sorted array, for finding the counterpart element and store the corresponding index *j*. This index is used to set the proper bit in the bitmask that represent the priority level value. The last instruction set the `ISimpleGenResKeywords.TASK_READY_PRIORITY`, an RT-Druid internal property. In this way the existing RT-Druid code, for creating the system data structures relative to the SRP protocol, can be reused.

Chapter 4

Simulation environment for EDF-based RTOS in Lauterbach Trace32

A simulation framework has been used for testing the kernel support developed and for studying the execution of tasks under this kernel. The framework is based on the Lauterbach Trace32 PowerView IDE.

Such a IDE provides only the instruction set simulator, hence, the framework has been extended with two custom plugins to support the execution of both periodic and engine-triggered tasks with the proposed kernel for Erika.

- Free running timer, used by the EDF kernel to handle the time representation in the system;
- Crankshaft simulator, used to generate interrupts related to the rotation of a crankshaft, hence generating the activation of the AVR tasks.

The Peripherals Simulation Model (PSM) allows to write a custom software that reacts and accesses to the microcontroller registers. Moreover, it provides functions for interacting with the simulated processor and other modules of the Trace32 simulator.

The custom plugins are developed in C language and compiled as dynamic linked libraries that can be loaded into Trace32 PowerView by using PRACTICE

commands. Moreover two files, simul.c and simul.h, provided by Lauterbach, implement the API functions for communicating with the PSM.

4.1 Timer implementation

The free running timer module simulates a 32-bit timer available in the STM32F4 MCU, using the PSM interface for reacting to specific registers of the microcontroller that are used for configuring the timer and reading its value. The value of such a timer is the temporal reference for the kernel needed to define absolute tasks deadlines and it is configured with the resolution specified in the `TICK_TIME` field in the OIL configuration file.
For the timer implementation, the first step was to define a struct for representing the timer registers (Lst.4.1). The PSM provides a special variable type, called `simulWord32`, to represent a 32-bit memory area. In this way, the timer registers can be accessed as a simple variables.

```
#define TIMER2_BASE 0x40000000
#define TIMER5_BASE 0x40000C00
#define CR1_OFFSET  0x00
#define CNT_OFFSET      0x24
#define PSC_OFFSET      0x28
#define ARR_OFFSET      0x2C
typedef struct
{
        simulWord32 CR1;         /* Address offset: 0x00 */
        simulWord32 CNT;         /* Address offset: 0x24 */
        simulWord32 PSC;         /* Address offset: 0x28 */
        simulWord32 ARR;         /* Address offset: 0x2C */
} TIM_Register;
```

Listing 4.1: Timer registers struct for library development.

The STM32F4 MCU provides two 32-bit timers and the struct definition is the same for both of them. The `TIMER2_BASE` and `TIMER5_BASE` macros represent the base address of each timer.
The Lst.4.2 defines a particular structure used for describing the timer properties, in particular it is used for accessing to the timer registers. Moreover, the `bustype, reset, work, intport` and `chport` variables are used by the PSM

to allow the communication between the modules; `regs` represents the timer register struct and the `freqDivider` represents the frequency divider. This last variable is needed because the update frequency of the timer is always less than processor frequency (also in case the timer prescaler value is set to zero).

```
typedef struct {
        simulWord32 startaddress;
        int bustype, reset, work, intport, chport;
        TIM_Register regs;
        simulTime ctimestamp;
        int freqDivider;
        void * ptimer;
} Timer;
```

Listing 4.2: Timer simulator properties.

When the dynamic linked library is loaded from the Trace32 PowerView, the PSM executes the `SIMUL_Init` function. Such a function must be used to initialize the timer simulation structure and to register the callback functions for the timer registers.

```
int SIMULAPI SIMUL_Init(simulProcessor processor, simulCallbackStruct * cbs) {
        Timer* timer;
        int i;
        char tim5[] = "timer5";
        strcpy(cbs->x.init.modelname, __DATE__ " Timer Model");
        timer = (Timer*) SIMUL_Alloc(processor, sizeof(Timer));
        timer->freqDivider = 2;
        timer->bustype = 0;
        timer->intport = -2;
        timer->startaddress = TIMER2_BASE;
        if(cbs->x.init.argc == 2)
        {
                if(strcmp (cbs->x.init.argp,tim5) == 0)
                        timer->startaddress = TIMER5_BASE;
        }
        Regs_Init(processor, timer);
        SIMUL_RegisterResetCallback(processor, TIMER_Reset, (simulPtr) timer);
        timer->ptimer = SIMUL_RegisterTimerCallback(processor, IntReqTimer, (
           simulPtr) timer);
        TIMER_Reset(processor, cbs, timer);
        SIMUL_Printf(processor, "%s \n", "Timer library loaded");
        return SIMUL_INIT_OK;
```

Listing 4.3: Initialization function for simulator plugin.

The initialization function (Lst.4.3) has two input parameters:

```
typedef struct
{
        int type;
        simulTime time;
        union {
                simulParamCallbackStruct init;
                simulParamCallbackStruct command;
                simulBusCallbackStruct bus;
                simulBusCallbackStruct32 bus32;
                simulBusCallbackStruct64 bus64;
                simulPortCallbackStruct port;
                simulPortCallbackStruct32 port32;
                simulPortCallbackStruct64 port64;
                simulTerminalCallbackStruct terminal;
        } x;
}
simulCallbackStruct;
```

Listing 4.4: Simul callback structure definition.

- `simulProcessor processor`, that represents the simulated processor, it can be used to retrieve some information such as the clock frequency;
- `simulCallbackStruct * cbs` that contains information about the simulation (for instance the time elapsed from the beginning of the simulation) and information about the simulated system status, (Lst.4.4).

For instance the initialization function uses the `simulCallbackStruct` for setting the simulated model name. Moreover, the `simulCallbackStruct` contains a char array that store the potential parameters specified in the library load command. In this way the user can select the proper timer.
The `Regs_Init` function is used to set the register callbacks. These callbacks are called when the simulator tries to access to memory address specified in the callback register function (Lst.4.5).

```
static void Regs_Init(simulProcessor processor, Timer * timer)
{
        simulWord from, to;
        int i;
        for (i = 0; i < NUM_OF_REGS; i++)
```

```
        {
                from = timer->startaddress + regs_offset[i];
                to = from + 3;
                SIMUL_RegisterBusWriteCallback(processor, writeFunc[i], (
                   simulPtr) timer, timer->bustype, &from, &to);
                SIMUL_RegisterBusReadCallback(processor, readFunc[i], (simulPtr)
                    timer, timer->bustype, &from, &to);
        }
}
```

Listing 4.5: Register callback function example.

The callback functions take the memory addresses range as input arguments (the variable `from` and `to`), in this way the the plugin detects when the simulator tries to access in that range.
The `writeFunc` and `readFunc` arrays contain the functions for handling memory accesses (Lst.4.6).

```
static int SIMULAPI CNT_Read(simulProcessor processor, simulCallbackStruct * cbs
   , simulPtr private)
{
        Timer * timer = (Timer*) private;
        SIMUL_ExtractWord(processor, &timer->regs.CNT, 32, &cbs->x.bus.address,
           cbs->x.bus.width, &cbs->x.bus.data);
        return SIMUL_MEMORY_OK;
}

static int SIMULAPI CNT_Write(simulProcessor processor, simulCallbackStruct *
   cbs, simulPtr private) {
        Timer * timer = (Timer*)private;
        SIMUL_InsertWord(processor, &timer->regs.CNT, 32, &cbs->x.bus.address,
           cbs->x.bus.width, &cbs->x.bus.data);
        return SIMUL_MEMORY_OK;
}
```

Listing 4.6: Read and write register functions.

Lst.4.7 shows the core of the plugin. This function, called every clock cycle, implements the timer behavior. For testing the proposed kernel, only the free runner timer in up mode has been implemented.

```
int SIMULAPI IntReqTimer(simulProcessor processor, simulCallbackStruct *cbs,
   simulPtr private) {
        static int divider = 0;
```

```
        Timer * timer = (Timer*) private;
        simulWord data;
        if (divider == timer->freqDivider - 1)
        {
                if (timer->regs.CR1 & 0x1)
                {
                        timer->regs.CNT++;
                }
        }
        divider = (divider + 1) % timer->freqDivider;
        return SIMUL_TIMER_OK;
}
```

Listing 4.7: Timer behavior implementation.

Before updating the counter value (`timer->regs.CNT++;`), the function checks if the timer has been activated by the software using the `timer->regs.CR1` variable.

4.2 Crankshaft simulator

The main goal of the crankshaft simulator is to generate an interrupt signal, at a specific angular position, for AVR task activation. The signal is sent to the simulated Nested Vector Interrupt Controller (provided by Lauterbach) through a simulated internal bus. The time instants at which such angular events occur are directly dependent on the speed trend of the engine. Hence, the model of the crankshaft requires some physical laws to be implemented in order to update the activation time and the angular speed at each cycle:

$$ActivationTime = \frac{\sqrt{\omega^2 + 2 \cdot \alpha \cdot \theta} - \omega}{\alpha} \tag{4.1}$$

$$AnugularSpeed = \sqrt{\omega^2 + 2 \cdot \alpha \cdot \theta} \tag{4.2}$$

where the variables represent:

- ω : the angular speed at the current time instant t;
- θ : the crankshaft angular displacement from the time instant t, after which the time elapsed from t and the new angular speed must be computed;

α : the angular acceleration that is assumed to be constant for the angular displacement θ.

Both equations have been implemented using the standard C mathematical library, and the listings are shown in the Lst.4.8 and Lst.4.9.

```
float getActivaionTime(float _omega){
        float first,second;
        float time;
        if(alphaCurr==0){
                time=theta / _omega;
                return time;
        }
        first= _omega * _omega+2*alphaCurr*theta;
        if(first<0){
                time = theta/omegaMinus;
                return time;
        }
        second=sqrt(first);
        if(second>omegaPlus){
                time = theta/omegaPlus;
                return time;
        }
        time = (second - _omega)/alphaCurr;
        return time;
}
```

Listing 4.8: Activation time function for the crankshaft simulator.

The first *if* condition of the Lst.4.8 covers the case in which the angular acceleration is equal to zero. This is the case of the uniform circular motion and the Eq.4.1 becomes:

$$ActivationtTime = \frac{\theta}{\omega}.$$

The other two *if* cover two cases:

- The ω is too low and the angular acceleration is less than zero. This case represents the scenario of uniform motion with the minimum angular speed;
- On the other hand, a saturation value is used to represent an uniform motion with the maximum angular speed.

```
float getOmega(float _omega){ //omega = rad/s
        float first,second;
        first= _omega * _omega+2*alphaCurr*theta;
        if(first<0)
                return omegaMinus;
        second=sqrt(first);
        if(second>omegaPlus)
                return omegaPlus;
        return second;
}
```

Listing 4.9: Angular speed function for the crankshaft simulator.

The above consideration are also used for implementing the function to compute the angular velocity (Lst.4.9).

The saturation values are defined as:

```
static const double omegaMinus=52.359; // rad/s 500 RPM
static const double omegaPlus=680.678; // rad/s 6500 RPM
```

The interrupt signal rate depends also from the acceleration. The function in charge to update the acceleration is shown in Lst.4.10.

```
void updateAcceleration(simulProcessor processor){
        simulTime now;
        double timeElapsed;
        SIMUL_GetTime(processor,&now);
        timeElapsed= (now - lastUpdate)/(double)1000000000000; //porto il tempo
           da picosecondi a secondi
        alphaCurr=RandomDouble(alphaCurr+(jerkMinus*timeElapsed),alphaCurr+(
           jerkPlus*timeElapsed));
        if(alphaCurr>alphaPlus)
                alphaCurr=alphaPlus;
        else if(alphaCurr<alphaMinus)
                alphaCurr=alphaMinus;
        SIMUL_GetTime(processor,&lastUpdate);
}
```

Listing 4.10: Function to update the acceleration for the crankshaft simulator.

The new acceleration value is generated by the `RandomDouble(double a, double b)` function that generates a random value in the specified range, following the uniform distribution. To obtain a realistic acceleration evolution, the jerks values, `jerkMinus` and `jerkPlus` are taken in the account. Moreover the

`timeElapsed` (time between two updates) is needed to compute the delta of the acceleration. Then a saturation to `alphaPlus` and `alphaMinus` is performed.

Moreover, the values of the physical law are expressed in seconds but the simulator is a discrete-time system. Hence, special functions have been created to convert the time measurement units from seconds to processor ticks.

```
float getOmegaInTick(float omega){
        return (omega/(((float)1000000000*2*3.141592)/11.9));
}
float getActivaionTimeTick(float _omega){
        float time;
        float ret;
        time=getActivaionTime(_omega);
        ret = (1000000000*time)/5.95;
        return floor(ret);
}
```

Listing 4.11: Function to convert values from second to tick.

An important issue in the update acceleration function, is to set `JerkMinus`, `JerkPlus`, `AlphaMinus`, `AlphaPlus` in order to obtain velocity and acceleration realistic profiles.
The simulator can be configured to use two different ways for generating the interrupt signals:

- File, which loads a predetermined speed pattern from a file;
- Random Speed Pattern, which generates a random speed evolution, given a set of configurations parameters (for instance maximum and minimum acceleration).

The initialization function (Lst.4.12) checks the input arguments; some possibilities are allowed:

- No input argument: the simulator forces the acceleration parameters to default values to achieve the most realistic speed trend:
  - `alphaPlus = 50`
  - `alphaMinus = -50`
  - `jerkPlus = 120`

  - `jerkMinus = -120`

- One input argument: the simulator assumes that the input argument is the file name that contains the angular speed profile; in this case the simulator fills two arrays representing the activation time (the time instant, expressed in ticks, for generating interrupt signal) and the angular speed.
- Four input arguments: these values represents the angular acceleration and angular jerk bounds. These values are used to generate, at run time, the speed profile.
- Five input arguments: the first four values represents the acceleration properties, as the previous case; the last argument represents the seed for random number generator initialization.

```
int SIMULAPI SIMUL_Init(simulProcessor processor,simulCallbackStruct * cbs){
  InterruptStruct* is;
  int i=0;
  int inputFile;
  char pdata[256];
  int ret;
  char** tokens;
  char* tok;
  const char delim[2]=";";
  double actTime;
  char* filename;
  int seed;
  interruptNum=0;
  dinamic=1;
  switch(cbs->x.init.argc-1){
    case 0:
      SIMUL_Printf(processor,"%s","externalInterrupt: no input argument");
      SIMUL_Printf(processor,"default parameter:");
      SIMUL_Printf(processor,"jerkMinus=-120");
      SIMUL_Printf(processor,"jerkPlus=120");
      SIMUL_Printf(processor,"alphaMinus=-50");
      SIMUL_Printf(processor,"alphaPlus=50");
      SIMUL_Printf(processor,"");
      SIMUL_Printf(processor,"");
      SIMUL_Printf(processor,"");
      jerkMinus=-120;
      jerkPlus=120;
      alphaMinus=-50;
      alphaPlus=50;
      alphaCurr=RandomDouble(alphaMinus,alphaPlus);
```

```
    omegaCurrent=getOmega(0);
    omega[0]=getOmegaInTick(omegaCurrent);
    activationTime[0]=getActivaionTimeTick(omegaCurrent);
    break;
  case 5:
    seed=atof(cbs->x.init.argp[5]);
    RandomInitialise(seed,9337);
  case 4:
    jerkMinus=atof(cbs->x.init.argp[1]);
    jerkPlus=atof(cbs->x.init.argp[2]);
    alphaMinus=atof(cbs->x.init.argp[3]);
    alphaPlus=atof(cbs->x.init.argp[4]);
    SIMUL_Printf(processor,"insert parameter: ");
    SIMUL_Printf(processor,"jerkMinus=%f",jerkMinus);
    SIMUL_Printf(processor,"jerkPlus=%f",jerkPlus);
    SIMUL_Printf(processor,"alphaMinus=%f",alphaMinus);
    SIMUL_Printf(processor,"alphaPlus=%f",alphaPlus);
    SIMUL_Printf(processor,"");
    SIMUL_Printf(processor,"");
    SIMUL_Printf(processor,"");
    alphaCurr=RandomDouble(alphaMinus,alphaPlus);
    omegaCurrent=RandomDouble(omegaMinus,omegaPlus);
    omega[0]=getOmegaInTick(omegaCurrent);
    activationTime[0]=getActivaionTimeTick(omegaCurrent);
    break;
  case 1:
    filename=(char*)malloc((strlen(cbs->x.init.argp[1])-1) * sizeof(char));
   memcpy(filename,&(cbs->x.init.argp[1])[1],(strlen(cbs->x.init.argp[1])-2));
    filename[(strlen(cbs->x.init.argp[1])-2)]='\0';
    SIMUL_Printf(processor,"%s",filename);
    inputFile=SIMUL_OpenFile(processor,filename,SIMUL_FILE_READ);
    if(inputFile==NULL){
      SIMUL_Warning(processor, "inputFile not found");
      return SIMUL_INIT_FAIL;
      break;
    }
    dinamic=0;
    ret=SIMUL_ReadlineFile(processor,inputFile,pdata,sizeof(pdata));
    ret=SIMUL_ReadlineFile(processor,inputFile,pdata,sizeof(pdata));
    ret=SIMUL_ReadlineFile(processor,inputFile,pdata,sizeof(pdata));
    i=0;
    while(ret!=0){
      tok=strtok(pdata,delim);
      omega[i]=getOmegaInTick(atof(tok));
      tok = strtok(NULL, delim);
      tok = strtok(NULL, delim);
      tok = strtok(NULL, delim);
      actTime=atof(tok)*1000000000.0;
      activationTime[i]=(int)(actTime/((double)5.95));
      SIMUL_Printf(processor,"%.20f;%f",omega[i],actTime);
```

```
        i=i+1;
        ret=SIMUL_ReadlineFile(processor,inputFile,pdata,sizeof(pdata));
      }
      interruptNum=i-1;
      break;
    default:
      SIMUL_Warning(processor,"Usage parameters: [<filename>] [jerkMinus
         jerkPlus alphaMinus alphaPlus]");
      return SIMUL_INIT_FAIL;
      break;
  }
  strcpy(cbs->x.init.modelname, __DATE__ "External Interrupt");
  is = (InterruptStruct*) SIMUL_Alloc(processor, sizeof(InterruptStruct));
  is->bustype = 0;
  is->intport = -2;
  Regs_Init(processor, is);
  SIMUL_RegisterResetCallback(processor, TIMER_Reset, (simulPtr) is);
  is->is_pointer = SIMUL_RegisterTimerCallback(processor, IntReqTimer,(simulPtr)
      is);
  SIMUL_RegisterBreakCallback(processor,(void*)exitCallBack,is);
  TIMER_Reset(processor, cbs, is);
  SIMUL_Printf(processor,"%s","externalInterrupt library loaded");
  if(dinamic==1){
    SIMUL_Printf(processor,"omega(rad/s) ; alpha(rad/s^2) ; activation time(s) ;
        time elapsed(s)");
    SIMUL_Printf(processor,"%.20f;%f;%f,%f",omega[0],alphaCurr,getActivaionTime(
       omegaCurrent),getActivaionTime(omegaCurrent));
  }
  else
    SIMUL_Printf(processor,"%s","File loaded");

  return SIMUL_INIT_OK;
}
```

Listing 4.12: Initialization function of the crankshaft simulator.

In the case of dynamic speed trend the initialization function computes:

- `alphaCurr`: initial angular acceleration value, generated as a random value between `alphaMinus` and `alphaPlus`;
- `omegaCurr`: initial angular velocity value, generated as a random value between `omegaMinus` and `omegaPlus`;
- `activationTime`: first time instant, expressed in ticks for generating an interrupt signal.

Moreover, the initialization function sets the callback function for a special register. This register is used to store the current speed value, when an interrupt signal is generated, thus simulating the presence of a TPU[20].

The `IntReqTimer` function (Lst.4.13) implements the behavior of the simulator.

```
int SIMULAPI IntReqTimer(simulProcessor processor, simulCallbackStruct *cbs,
   simulPtr private){
  static int angInterrupt=0;
  InterruptStruct * is = (InterruptStruct*) private;
  simulWord data;
  simulWord temp;
  static double countTick=0;
  temp=0;
  if(countTick==activationTime[currentExp]){
    is->omegaRegs.OMEGA_REG=omega[currentExp];
    is->regs.PR=1;
    temp=1;
    SIMUL_SetPort(processor,118,1,&temp);
    }
  currentExp=currentExp+1;
  if(dinamic==1){
      updateAcceleration(processor);
      omegaCurrent = getOmega(omegaCurrent);
      omega[currentExp]=getOmegaInTick(omegaCurrent);
      activationTime[currentExp]=activationTime[currentExp-1]+
         getActivaionTimeTick(omegaCurrent);
  }
  else{
    if(currentExp==interruptNum){
      SIMUL_Printf(processor,"End of simulation");
      SIMUL_Stop(processor);
      }
    }
  }
countTick=countTick+1;
return SIMUL_TIMER_OK;
}
```

Listing 4.13: Behavior of the crankshaft simulator.

The instruction set simulator calls this function every clock cycle. The function compares the `countTick` variable, that represents the current cycle counter, and the `activationTime` variable. If the values are equal, the function calls `SIMUL_SetPort` function for generating an interrupt signal and store the angular

speed value in a register. The interrupt signal, using the *Nested Vector Interrupt Controller*, actives an Interrupt Service Routine of the Erika operating system, running on the simulated processor. The ISR retrieves the angular speed value from the register and calls the activate AVR task function. If a dynamic speed computation is selected, the plugin updates the angular acceleration value and computes the next angular speed and activation time values, on the other hand, in case the speed profile is loaded from a file, the index of the activation time and angular speed are incremented.

4.3 Simulation environment configuration

The timer and crankshaft simulators have been integrated with the Cortex instruction set simulator. The resulting framework (Fig. 4.1) is able to collect a full trace related to the execution of an application. Trace32 PowerView IDE allows to analyze, explore, process and partially re-execute the trace data.

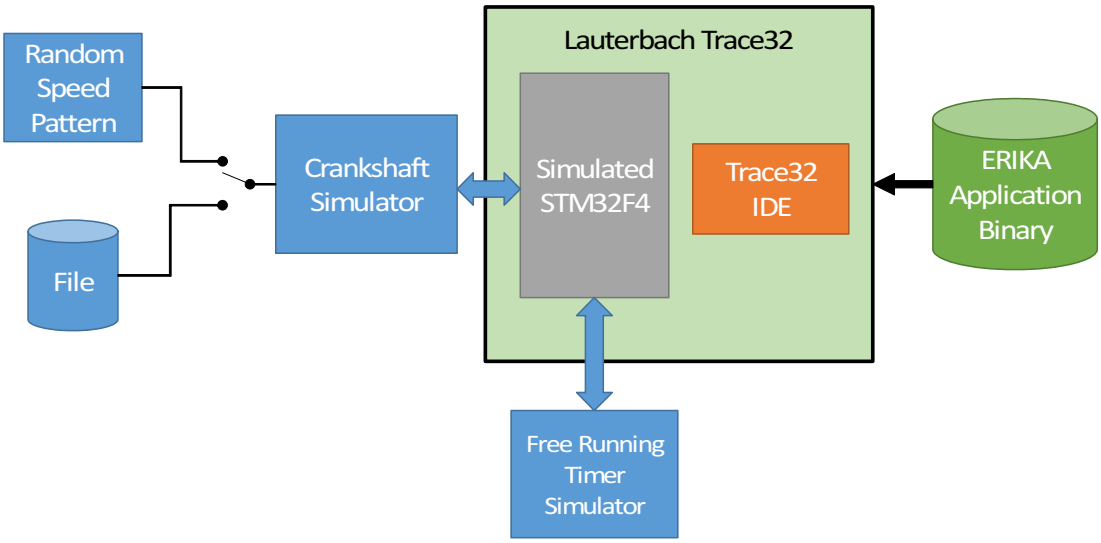

Figure 4.1: Simulator framework block diagram.

The framework can be configured either manually, using graphic interfaces (Fig. 4.2), or automatically, using a set of scripts (Lst.4.14).

```
SYStem.Down
SIM.UNLOAD
```

```
SYStem.CPU STM32F407VG
vco.FREQUENCY 168.0MHz
area
SYStem.UP
SIM.Load nvic.dll
SIM.Load timerSTM32.dll
SIM.Load interruptGenerator.dll
PER
trace.size 1073741823
D.load ..\workspace_erika\edf_avrTask_test\Debug\c_mX.elf
List
enddo
```

Listing 4.14: Trace32 PowerView configuration script.

The scripts are written in PRACTICE language, a Lauterbach script language. PRACTICE is a line-oriented test language which can be used to solve all usual problems of digital measurement engineering. PRACTICE-II is an enhanced version of this test language, first developed in 1984 for in-circuit emulators.

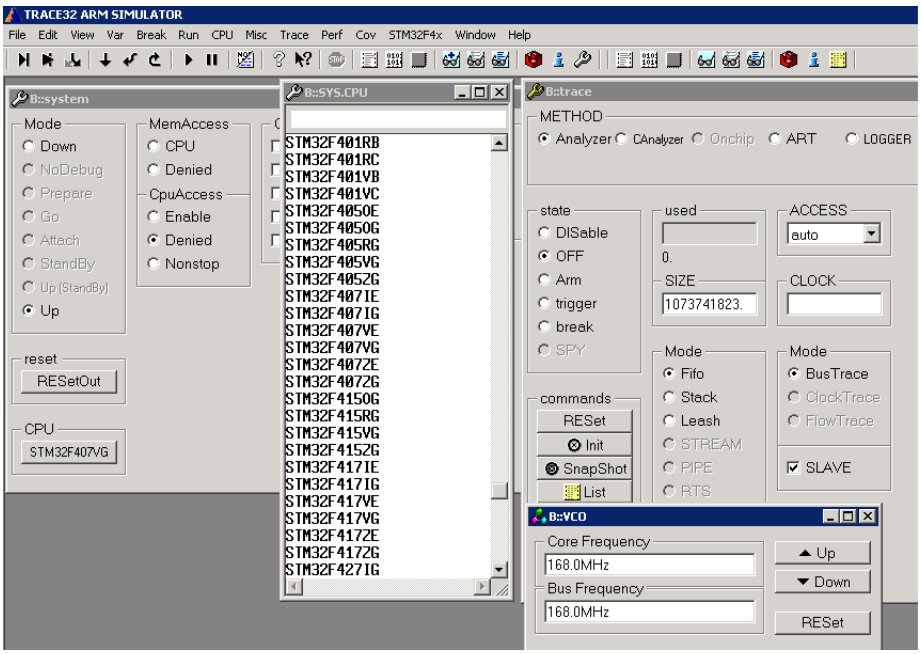

Figure 4.2: Trace32 PowerView configuration windows.

Trace32 PowerView IDE is a script-based tool. This means that the environment can be full customize using the scripts. At start up, Trace32 calls a

default script for the basic configurations of the graphic interface (i.e., windows size). Modifying this script, as shown in Lst.4.14, Trace32 automatically sets the proper parameters for simulating the Cortex-M3 instruction set, sets the memory map for the STM32 Discovery board and loads the custom library (`nvic.dll`, `timerSTM32.dll` and `interruptGenerator.dll`). In this way, the simulation environment is properly set at startup.

Chapter 5

Experimental evaluation

This chapter reports the experimental results aiming at evaluating the performance of the proposed EDF kernel. Other Erika kernel conformance classes (see section 2.1.1) were tested for comparison purposes.

Before showing the results, the test purposes and the environment settings are described.

5.1 Experimental environment

In order to evaluate the performance of the proposed kernel we performed a comparison with other three Erika kernel conformance classes:

- original EDF kernel conformance class;
- conformance class FP, that provides a minimal implementation of fixed-priority scheduling;
- OSEK kernel, in particular the BCC2 conformance class, which provides fixed-priority scheduling with stack sharing, more than one task per priority level and multiple task activations.

The original EDF kernel version was selected because it is the baseline for the proposed kernel; the conformance class FP is the most optimized kernel provided by Erika Enterprise; the BCC2 OSEK kernel, instead, is the standard kernel type adopted for engine control by automotive industries.

These comparative experiments want to prove that the proposed kernel drawbacks, in terms of memory footprint and run-time overhead, is very restricted with respect to the EDF and the fixed-priority based kernel.

5.1.1 Run-time overhead

This experiment has been conducted for measuring the resulting run-time overhead of the `ActivateTask` system call. Such a system call computes the task deadlines and manages the ready queue, resulting in the most time-consuming kernel mechanism.

The overhead has been measured on a STM34F4 platform running at 168Mhz with FPU enabled. The experiments compare the run-time overhead of proposed kernel with the overhead of the original EDF kernel used as a baseline for this work. The overhead has been measured as a function of the number of periodic tasks, keeping the number of AVR tasks to two.

Aiming at testing the system in a condition where it is not overloaded, the *UUniFast* algorithm [22] was used for creating a valid task sets. Given the number of task and the total utilization, this algorithm generates task utilization values, such that the generated utilization values are characterized by a uniform distribution.

A Java tool has been developed for generating the software source codes. Fig.5.1 shows the block diagram of such tool:

- OIL generator: it creates the OIL files for the selected task set;
- C generator: using the *UUniFast* algorithm, it creates the source code of the selected task set;
- PRACTICE script generator: it creates a script for Lauterbach Trace32 PowerView, such that the test can be automatically performed;
- Compiler: the source files are compiled and the system image file is stored in the Trace32 subdirectory;
- Simulator environment: it represents the Lauterbach Trace32 PowerView block.

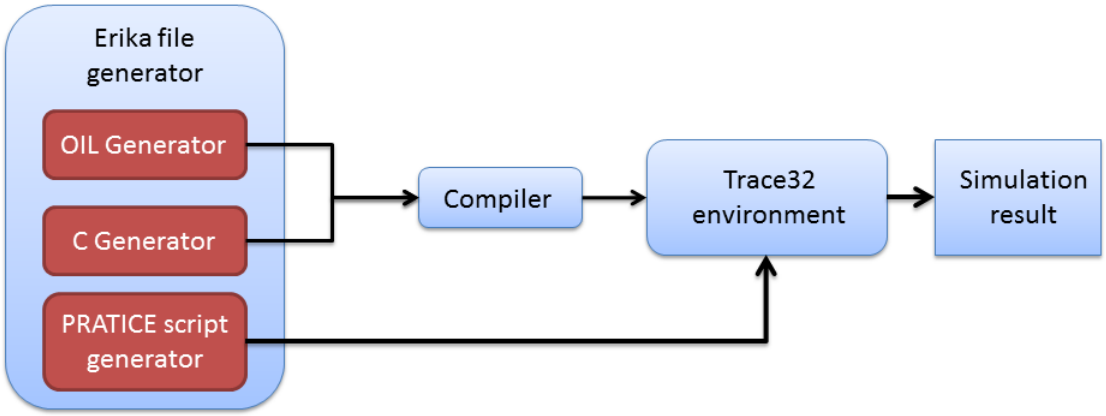

Figure 5.1: Automatic source generator block diagram.

Four task set classes were created by the Java tool. Each classes represents a task set with a specific number of tasks (three, five, seven and ten periodic tasks all with two AVR tasks). All tasks, independently of their classes, have the same processor utilization (that is equal to 75%), and the computation time of each task was computed by the *UUniFast* algorithm (Lst.5.1)

```
...
void TaskAvr1_AvrMode1(){
        float k=0;
        float i=0;
        for(i=0;i<53124;i++) //9820us
                k++;
        }
void TaskAvr1_AvrMode2(){
        float k=0;
        float i=0;
        for(i=0;i<24634;i++) //4556us
                k++;
        }
void TaskAvr1_AvrMode3(){
        float k=0;
        float i=0;
        for(i=0;i<18154;i++) //3359us
                k++;
        }
void TaskAvr1_AvrMode4(){
        float k=0;
        float i=0;
        for(i=0;i<10315;i++) //1909us
                k++;
        }
```

```
void TaskAvr1_AvrMode5(){
        float k=0;
        float i=0;
        for(i=0;i<8921;i++) //1650us
                k++;
        }
TASK(TaskAvr1){
        float w=OMEGA_REG->omega;
        if(w<0.00000028462033271790){ //omega < 1435.0605 RPM
        TaskAvr1_AvrMode1();
        }
        if(w>=0.00000028462033271790 && w<0.00000072053718299866){
        TaskAvr1_AvrMode2();
        }
        if(w>=0.00000072053718299866 && w<0.00000106258696365356){
        TaskAvr1_AvrMode3();
        }
        if(w>=0.00000106258696365356 && w<0.00000125689047088623){
        TaskAvr1_AvrMode4();
        }
        if(w>=0.00000125689047088623 && w<0.00000128916669692993){
        TaskAvr1_AvrMode5();
        }
}
void TaskAvr2_AvrMode1(){
        float k=0;
        float i=0;
        for(i=0;i<25968;i++) //4800us
                k++;
        }
void TaskAvr2_AvrMode2(){
        float k=0;
        float i=0;
        for(i=0;i<25586;i++) //4730us
                k++;
        }
void TaskAvr2_AvrMode3(){
        float k=0;
        float i=0;
        for(i=0;i<14961;i++) //2769us
                k++;
        }
TASK(TaskAvr2){
        float w=OMEGA_REG->omega;
        if(w<0.00000040474313011169){
        TaskAvr2_AvrMode1();
        }
        if(w>=0.00000040474313011169 && w<0.00000072471891555786){
        TaskAvr2_AvrMode2();
        }
```

```
        if(w>=0.00000072471891555786 && w<0.00000128916669692993){
        TaskAvr2_AvrMode3();
        }
}
TASK(taskPeriodic1){
        float k=0;
        float i=0;
        for(i=0;i<11971;i++) //2214 us
                k++;
}
TASK(taskPeriodic2){
        float k=0;
        float i=0;
        for(i=0;i<77176;i++) //14269 us
                k++;
}
TASK(taskPeriodic3){
        float k=0;
        float i=0;
        for(i=0;i<69647;i++) //12877 us
                k++;
}

...
```

Listing 5.1: Task set, with three periodic tasks and two AVR tasks, used for computing the run-time overhead.

The maximum angular acceleration is the same for both AVR tasks, meanwhile the angular deadline is set at 180 degrees for the first task and 360 degrees for the second one. The task sets were compiled using the GNU ARM compiler with the execution time optimization (-O3) flag enabled.

Each task set has been executed for six seconds of simulated time, due the limited sample buffer of the Trace32PowerView. Moreover a significant number of samples was collected (considering that the activation time for an AVR task, with angular period equals to 360 degrees, is usually bounded between 9 and 120 milliseconds).

The simulations are executed on a machine equipped with an Intel Core i7 4790k processor running at 4GHz with 32GBs of RAM and a 500GBs of SSD. With this hardware setting the total time for obtaining the result from one simulation is about three minutes (see Table.5.1).

Simulation time (secs)	Trace time (secs)	Processing time (secs)	RAM occupancy (GBs)
1	19	16	4.1
3	48	40	12.3
6	96	92	25.2

Table 5.1: Times and memory consumption for the proposed simulation framework.

5.1.2 Footprint

This experiment aims at comparing the memory footprint of the proposed kernel with the footprints of the OSEK BCC2, FP and original EDF kernels. The experiment evaluates the footprint as a function of the number of AVR tasks keeping the number of periodic tasks to two. This choice was made because the OSEK BCC2, the original EDF and the FP kernels do not distinguish the AVR tasks from the periodic ones, while the proposed kernel in this thesis is specially conceived for handling AVR tasks.

Hence, the measured binaries contain only the kernel structures and they not contain the function implementations or other specific application features, for instance the C source files contain only the `ActivateTask` system call (Lst.5.2). The results are obtained compiling the kernels for the STM32F4 platform using the GNU ARM compiler with the size optimization (-Os) flag enabled.

In this experiment we used the proposed kernel configured for using the Fast-SQRT algorithm for computing the deadlines of AVR tasks. Moreover to evaluate the overhead trend all the AVR tasks have different angular parameters (angular deadline and maximum angular acceleration), in this way a different data structure is created for each task.

```
#include "ee.h"
TASK(taskPeriodic1){}
TASK(taskPeriodic2){}
TASK(TaskAvr1){}
TASK(TaskAvr2){}
TASK(TaskAvr3){}
TASK(TaskAvr4){}
TASK(TaskAvr5){}
TASK(TaskAvr6){}
TASK(TaskAvr7){}
TASK(TaskAvr8){}
```

```
int main(void){
ActivateTask(taskPeriodic1);
ActivateTask(taskPeriodic2);
ActivateTask(TaskAvr1);
ActivateTask(TaskAvr2);
ActivateTask(TaskAvr3);
ActivateTask(TaskAvr4);
ActivateTask(TaskAvr5);
ActivateTask(TaskAvr6);
ActivateTask(TaskAvr7);
ActivateTask(TaskAvr8);
};
```

Listing 5.2: C source file of an application for the footprint evaluation.

The Lst.5.3 shows the Windows PowerShell batch script to evaluate the binary size.

```
echo "EDF,EDF+AVR,FP,OSEK"
$sizeEDF=(Get-Item '.\footprint\0_AVR\edf\Debug\c_mX.bin').length
$sizeEDFAVR=(Get-Item '.\footprint\0_AVR\edfavr\Debug\c_mX.bin').length
$sizeFP=(Get-Item '.\footprint\0_AVR\fp\Debug\c_mX.bin').length
$sizeOSEK=(Get-Item '.\footprint\0_AVR\osek\Debug\c_mX.bin').length
echo "$sizeEDF,$sizeEDFAVR,sizeFP,$sizeOSEK"

$sizeEDF=(Get-Item '.\footprint\1_AVR\edf\Debug\c_mX.bin').length
$sizeEDFAVR=(Get-Item '.\footprint\1_AVR\edfavr\Debug\c_mX.bin').length
$sizeFP=(Get-Item '.\footprint\1_AVR\fp\Debug\c_mX.bin').length
$sizeOSEK=(Get-Item '.\footprint\1_AVR\osek\Debug\c_mX.bin').length
echo "$sizeEDF,$sizeEDFAVR,$sizeFP,$sizeOSEK"

$sizeEDF=(Get-Item '.\footprint\2_AVR\edf\Debug\c_mX.bin').length
$sizeEDFAVR=(Get-Item '.\footprint\2_AVR\edfavr\Debug\c_mX.bin').length
$sizeFP=(Get-Item '.\footprint\2_AVR\fp\Debug\c_mX.bin').length
$sizeOSEK=(Get-Item '.\footprint\2_AVR\osek\Debug\c_mX.bin').length
echo "$sizeEDF,$sizeEDFAVR,$sizeFP,$sizeOSEK"

$sizeEDF=(Get-Item '.\footprint\3_AVR\edf\Debug\c_mX.bin').length
$sizeEDFAVR=(Get-Item '.\footprint\3_AVR\edfavr\Debug\c_mX.bin').length
$sizeFP=(Get-Item '.\footprint\3_AVR\fp\Debug\c_mX.bin').length
$sizeOSEK=(Get-Item '.\footprint\3_AVR\osek\Debug\c_mX.bin').length
echo "$sizeEDF,$sizeEDFAVR,$sizeFP,$sizeOSEK"
....
```

Listing 5.3: PowerShell script to evaluate memory footprint.

5.2 Experimental results

5.2.1 Run-time overhead

Fig.5.2 shows a bar plot representing the maximum and the average run-time overhead (expressed in microseconds) for the `ActivateTask` system call of the proposed kernel (referred to as EDF-AVR) under both the cases of application of the FastSQRT algorithm and the lookup table, configured to use their default SpeedType (RPM speed type for lookup table configuration and *rev/tick* for the FstSQRT algorithm). Moreover, the original kernel (referred to as EDF) run-time overhead has been computed for comparison purposes. The

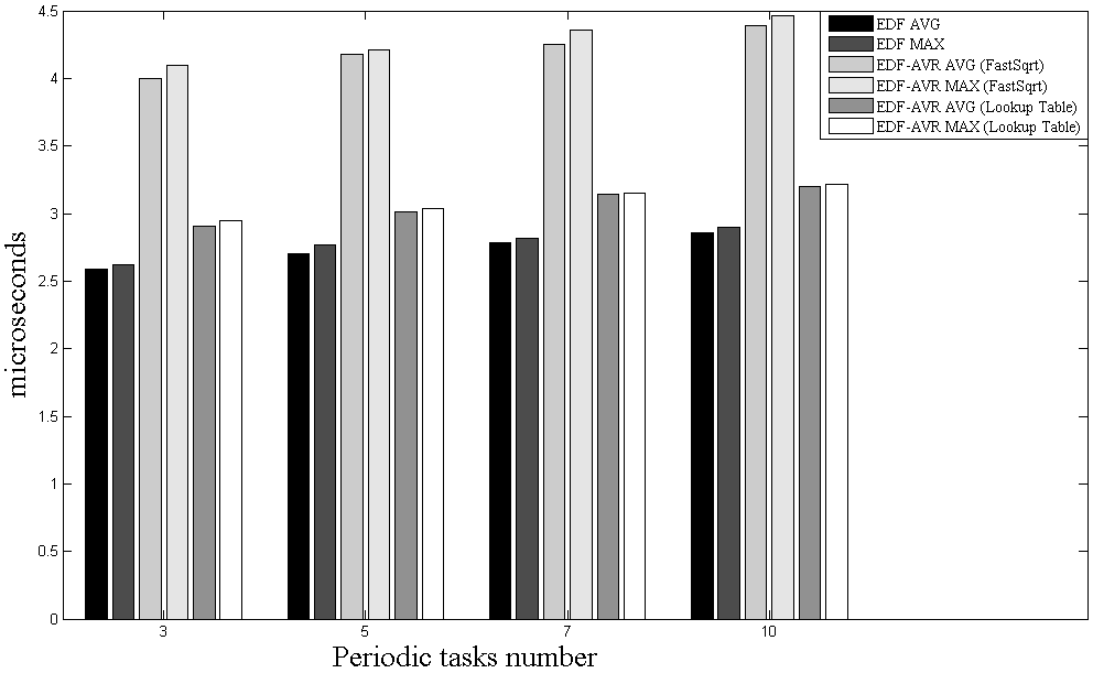

Figure 5.2: Maximum and average run-time overhead.

result shows that the proposed kernel have an additional overhead ranging from a few hundreds of nanoseconds to a maximum of 1.5 microseconds with respect to the original EDF kernel. The maximum additional overhead corresponds to the FastSQRT kernel configuration due to floating point operations. On the other hand the overhead slowly increase, hence the system has a good scalability.

Moreover, consider now the case in which an AVR task having a minimum deadline of 4 milliseconds (considering the Eq. 1.1 with $\omega = 6500RPM$ and

Number of tasks	3	5	7	10
EDF AVG	0.0065%	0.0068%	0.0069%	0.0072%
EDF MAX	0.0066%	0.0069%	0.0070%	0.0072%
EDF-AVR (FastSQRT) AVG	0.01%	0.0104%	0.0106%	0.0110%
EDF-AVR (FastSQRT) MAX	0.0102%	0.0105%	0.0109%	0.0112%
EDF-AVR (Lookup table) AVG	0.0073%	0.0075%	0.0079%	0.0080%
EDF-AVR (Lookup table) MAX	0.0074%	0.0076%	0.0079%	0.0080%

Table 5.2: Ratio between overhead values and the minimal deadline.

$\Delta = 180 degrees$). In this case the higher overhead value represents the 0.01% of the available task time. Tbl 5.2 shows the ratio between overhead values and the minimal deadline.

5.2.2 Footprint

Fig. 5.3 reports the footprint in bytes for OSEK BCC2, FP, EDF and EDF-AVR kernels as a function of the number of AVR tasks, while keeping the number of periodic tasks to two.

As can be observed from the graph, the OSEK BCC2 kernel shows the greater footprint: this is because it contains a consistent number of data checks and mechanisms required for being fully compliant with the OSEK standard.

The EDF-AVR kernel has the same footprint of original AVR when no AVR tasks are present (as clearly expected) and requires around 180 additional bytes for handling one AVR task with respect to EDF kernel (that requires around 40 additional bytes for handling one AVR tasks). This relevant footprint difference is due to the implementation of the new `ActivateTask` system call. In fact,

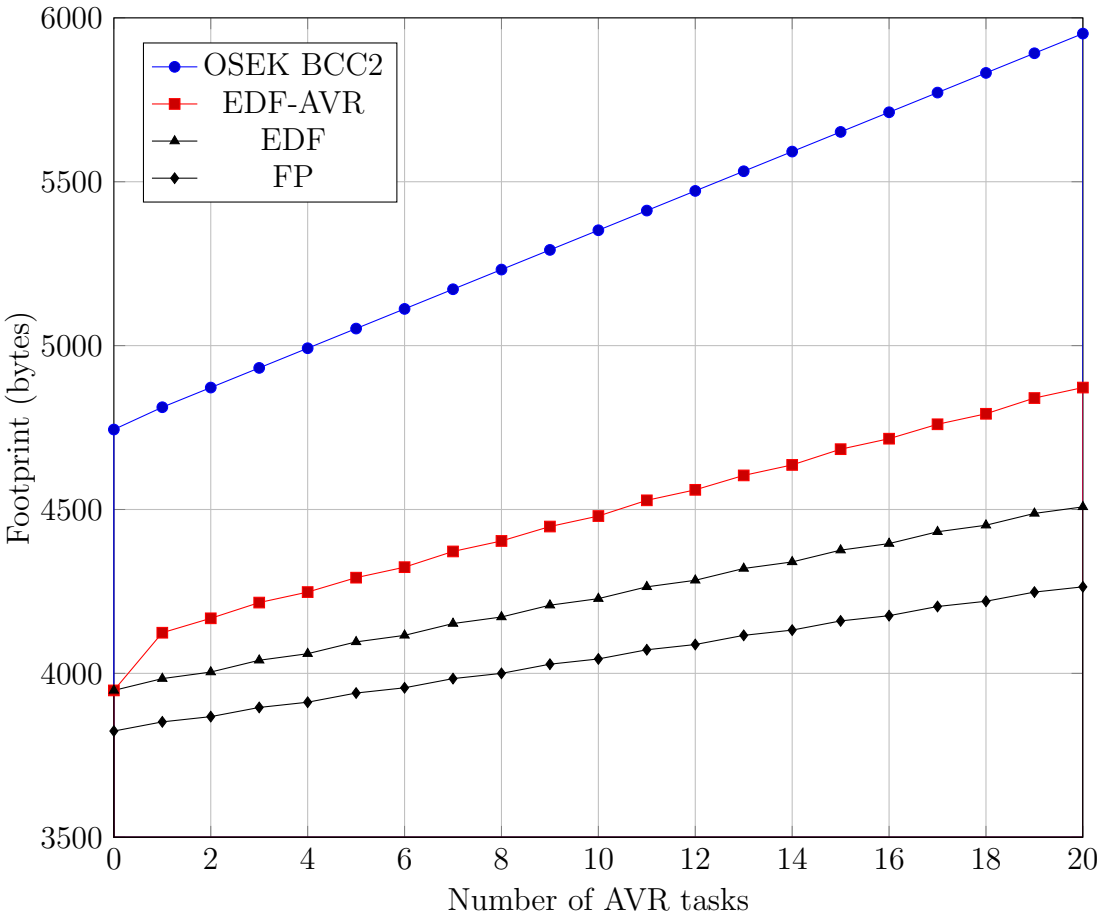

Figure 5.3: Footprint in bytes for different conformance classes of Erika.

in the case of two AVR tasks the additional footprint for EDF-AVR kernel is around 48 bytes with respect to the case with one AVR tasks.

FP kernel shows the lower footprint: this is because it is a minimal implementation of the fixed-priority scheduling.

Moreover, this experiment is performed using different angular parameters (angular period and angular acceleration) for each task, this introduces a kind of pessimism.

Fig. 5.4 shows the memory footprint of EDF-AVR kernel in case angular period assumes four distinct values (that represents a common engine control application). In this case, the footprint results to be reduced with respect to the EDF-AVR with different angular parameters for each task.

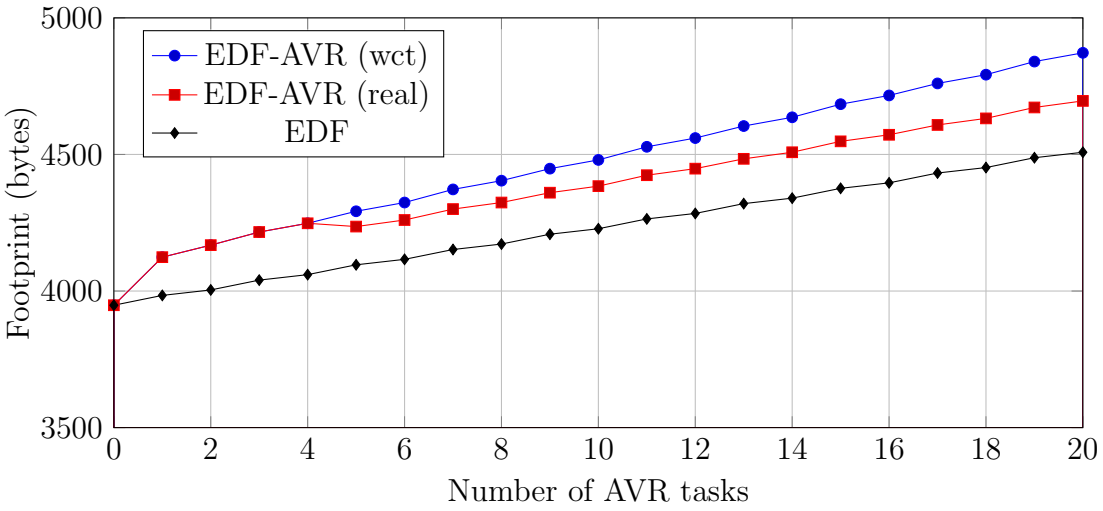

Figure 5.4: Footprint in bytes for EDf-AVR worst case and real case.

The additional footprint for handling an AVR task under this EDF-AVR configuration is around 40 bytes as well as the EDF kernel.

Chapter 6

Conclusion

In this master thesis we studied the problem of design and implementation of a real-time operating system for supporting engine control applications with dynamic-priority scheduling. This work has been based on the ERIKA Enterprise real-time operating system.

Being engine control applications typically managed by an AUTOSAR/OSEK standard operating system (RTOS), the new kernel has been designed to have an API with minimal differences with respect to the OSEK standard, thus limiting the effort for adopting the proposed solution in existing engine control applications. The same design constraints have been considered for the RTOS configuration, providing minimal extensions to the OSEK Implementation Language (OIL), which is the standard language for configuring an OSEK RTOS.

Typically engine control applications include computational activities consisting of periodic tasks, activated by timers, and engine-triggered tasks, activated at specific crankshaft position. The deadlines of such tasks depend on the engine speed, hence two different approaches have been considered for the kernel implementation depending on the representation of the engine speed available in the system.

An approach is based on a fast algorithm for computing the square root function (FastSQRT), while the other one relies on lookup tables. Both approaches have been discussed and compared in terms of precision, footprint, and run-time overhead.

The deadline computation error of both approaches results less than 0.04%.

Moreover, in the case of the FastSQRT approach the approximate deadline results always less than the theoretical one. This means that the system is always in a safe state (for instance if a job misses the approximate deadline there is a small time margin before the job misses the theoretical one).

A comparison made with the existing EDF kernel included in ERIKA Enterprise showed that the overhead introduced by the deadline computation is bounded from a few hundreds of nanoseconds to a maximum of 1.5 microseconds, depending on its configuration. In particular the lookup table approach using RPM as engine speed representation introduces the minimum overhead.

The proposed kernel requires about 250 bytes of additional footprint with respect to the existing EDF kernel and less than 500 bytes increment with respect to a minimal implementation of fixed-priority scheduling to handle 10 AVR tasks. Analysis of the results showed that the footprint also depends on the angular parameters. In fact the additional footprint for adding a task with the same angular parameters is comparable to the EDF additional footprint.

A simulation framework has also been developed for studying the execution of tasks under the proposed kernel. Such a framework has been realized extending the Lauterbach Trace32 suite with an engine crankshaft simulator, for generating angular events, and a free runner timer, required by the Erika EDF implementation, to support the execution of the implemented EDF kernel. Thanks to this framework it is possible to collect execution traces of real code without using any hardware devices (microcontrollers, debugger, etc.)

The efforts needed to move an existing engine control application from an OSEK RTOS to the proposed kernel are minimal. Few steps are needed for moving an application, that are:

- select the mechanism for reading the engine speed (which is generally already present in such applications);
- replace all the occurrences of the `ActivateTask` system call used for activating the AVR tasks with the new version;
- set the angular parameters (angular deadline and maximum engine acceleration) in the OIL configuration file.

Future works

A number of open issues must be explored to allow a full support to engine control application. These issues suggest a variety of research directions.

One such direction would be to investigate the support for multi-core devices, that are becoming an important point of reference for the automotive industries. Such devices involve many other problems related to the scheduling policy for handling multiple processors.

Moreover, other kind of dynamic scheduling can be taken in account also for better handling shared resources.

Bibliography

[1] D. Buttle, "Real-time in the prime-time." Keynote speech given at the 24th Euromicro Conference on real-Time Systems(ECRTS 2012), Pisa, Italy, July 12th 2012.

[2] L. Guzzella and C. H. Onder, "Introduction to Modeling and Control of Internal Combustion Engine System" 2th Edition.

[3] J. Kim, M. Lakshmanan and R. Rajkumar, "Rhythmic tasks: A new task modewl with continually varying periods for cyber-physical systems", in Proc. of the Third IEEE/ACM Int. Conference on Cyber-Physical Systems (ICCPS 2012), Beijing, China, April 17-19 2012, pp. 28-38.

[4] V. Pollex, T. Feld, F. Slomka, U. Margull, R. Mader and G. Wirrer, "Sufficient real-time analysis fo an engine control unit with constant angular velocities" in Proc. of the Design, Automation and Test Conference in Europe, Grenoble, France, March 18-22 2013.

[5] R. I. Davis, T. Feld, V. Pollex and F. Slomka, "Schedulability tests for tasks with variable rate-dependent behaviour under fixed priority scheduling", in Proc. 20th IEEE Real-Time and Embedded Technology and Applications Symposium, Berlin, Germany, April 15-17 2014.

[6] A. Biondi, A. Melani, M. Marinoni, M. D. Natale and G. Buttazzo, "Exact interference of adaptive variable-rate tasks under fixed-priority scheduling", in Proceedings of the 26th Euromicro Conference on realt-Time systems (ECRTS 2014), Madrid, Spain, July 8-11 2014.

[7] G. Buttazzo, "Rate Monotonic vs. EDF: Judgment Day", Real-Time System, Vol.28 pp. 1-22,2005.

[8] G. Buttazzo and P. Gai, "Efficient EDF implementation for small embedded system", in Proc. of the 2nd Int Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2006), Dresden, Germany, July 2006.

[9] G. Buttazzo, E. Bini and D. Buttle, "Rate-adaptive tasks: Model, analysis and design issues", in Proc. of the Int. Conference on Design, Automation and Test in Europe (DATE 2014), Dresden, Germany, March 24-28, 2014.

[10] A. Biondi and G. Buttazzo, "Engine Control: Task Modeling and Analysis", in Proceedings of the International Conference on Design, Automation and Test in Europe (DATE 2015), Grenoble, France, March 2015.

[11] Z. Guo and S. Baruah, "Uniprocessor EDF scheduling of AVR task systems", in Proc. of the ACM/IEEE 6th International Conference on Cyber-Physical Systems (ICCPS 2015), Seattle, USA, April 2015.

[12] A. Biondi, G. Buttazzo and S. Simoncelli, "Feasibility Analysis of Engine Control Tasks under EDF Scheduling", In Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS 15), Lund, Sweden, July 7-10, 2015.

[13] P. Gai, G. Lipari, L. Abeni, M. di Natale and E. Bini, "Architecture for a portable open source real-time kernel environment," in Proceedings of the Second Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial, November 2000

[14] OSEK, OSEK/VDX Operating System Specification 2.2.1. http://www.osek-vdx.org: OSEK Group,2003.

[15] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold", in Proc. of the 6th IEEE Int. Conference on Real-Time Computing Systems and Applications (RTCSA'99), Hong Kong, China, December 13-15, 1999.

[16] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo, "IRIS: A new reclaiming algorithm for server-based real-time systems", in Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada, May 25-28 2004.

[17] A. Biondi, G. Buttazzo, and M. Bertogna, "Supporting component-based development in partitioned multiprocessor real-time systems", in Proceedings of the 27th Euromicro Conference on Real-Time Systems(ECRTS 2015), Lund, Sweden, July 8-10, 2015.

[18] "RT-Druid A tool for architecture-level design of embedded systems", white paper, Evidence S.r.l. `http://www.evidence.eu.com`

[19] Trace32 instruction set simulator. [Online] Available: `http://www2.lauterbach.com/pdf/simulator_api.pdf`

[20] Freescale semiconductor Application Note AN3769. Using the engine position (CRANK and CAM) eTPU function. [Online]. Available: `http://cache.freescale.com/files/32bit/doc/app_note/AN3769.pdf`

[21] C. Lomont, "Fast inverse square root", [Online], Available: `http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf`

[22] E. Bini and G. Buttazzo, "Measuring the performance of schedulability tests", Real-Time Systems, vol. 30, no. 1-2, pp. 129–154, May 2005.