DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, *SWEDEN 2016*

Efficient Scheduling Library for FreeRTOS

ROBIN KASE

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

KTH Information and
Communication Technology

Efficient Scheduling Library for FreeRTOS

ROBIN KASE

Master's Thesis at KTH Information and Communication Technology
Supervisor: Kenji Kise (Tokyo Institute of Technology)
Examiner: Johnny Öberg (KTH)

TRITA-ICT-EX-2016:166

Abstract

At present, there is a gap between practical implementations of task scheduling on numerous popular Real-Time Operating Systems (RTOSs) and theoretical real-time scheduling. It is difficult to choose what theoretical real-time scheduling concepts to implement when designing a kernel, as theoretical concepts grow and improve over time. Furthermore, the kernel can be kept simpler when offering only simple fixed priority scheduling policy, as advanced scheduling features often require more complex implementation and larger overhead. By offering a real-time scheduling library implemented in user space, the user can choose whether to skip the overhead, or use more advanced theories. At the moment there exists already several scheduling frameworks for FreeRTOS. However, they are either difficult to use, not completely implemented in user space, or not providing various theoretical scheduling policies.

An open source scheduling library for FreeRTOS implemented in user space that is user friendly and runs with low overhead, Efficient Scheduling Library (ESFree) is proposed.

The proposed scheduling library provides polling server that runs aperiodic and sporadic jobs, dependable timing error detection and handling, Rate-Monotonic Scheduling (RMS), Deadline-Monotonic Scheduling (DMS) and Earliest Deadline First (EDF) scheduling policies to provide theoretical real-time scheduling features to speed up development of complex projects, and make FreeRTOS friendlier to students who have newly studied real-time scheduling.

Sammanfattning

För närvarande finns en klyfta mellan praktisk implementation av uppgiftsschemaläggning påflera populära realtidsoperativsystem (RTOS) och teoretisk realtidsschemaläggning. Det är svårt att välja vilka teoretiska realtidskoncept som ska implementeras när en kärna designas, eftersom teoretiska koncept ökar och förbättras över tid. Dessutom kan kärnan hållas enklare när endast enkel fixerad prioritetsschemaläggningspolicy erbjuds, dåavancerade schemaläggningsfunktioner ofta begär komplexare implementation och större overhead. Genom att erbjuda ett realtidsschemaläggningsbibliotek implementerad inom användarutrymmet, kan användaren välja mellan att skippa overheaden eller använda mera avancerade teorier. För tillfället finns det redan flera schemaläggningsramverk för FreeRTOS. Emellertid antingen är de svåra att använda, inte helt implementerade inom användarutrymmet, eller de tillhandahåller inte diverse teoretiska schemaläggningspolicyn.

Ett nytt öppet källkodsbibliotek för FreeRTOS implementerad inom användarutrymmet som är användarvänlig och exekverar med låg overhead, Efficient Scheduling Library (ESFree) föreslås.

Det föreslagna schemaläggningsbiblioteket tillhandahåller pollande server som kör aperiodiska och sporadiska jobb, pålitlig timingfeldetektering och hantering, Rate-Monoton Schemaläggning (RMS), Tidsgräns-Monoton Schemaläggning (DMS) och Tidigast Tidsgräns Först (EDF) schemaläggningspolicyn för att tillhandahålla teoretisk realtidsschemaläggningsfunktioner för att påskynda utveckling av komplexa projekt, och göra FreeRTOS vänligare för studenter som nyligen har studerad realtidsschemaläggning.

Acknowledgements

First I would like to thank Kenji Kise at Tokyo Institute of Technology for supervising me through this master's thesis.

I would also like to thank Thiem Van Chu and Susumu Mashimo, members at Kise Laboratory for advising me in scientific writing, tools and hardware.

Finally I would like to thank Scandinavia-Japan Sasakawa Foundation for supporting my exchange studies at Tokyo Institute of Technology.

Contents

Chapter 1

Introduction

Real-Time Operating Systems (RTOSs) are used in a large variety of systems such as Internet of Things (IoT), robotics, spacecrafts, aircrafts, automotives, network infrastructures and many more. Yet several commercial RTOSs do only offer simple fixed priority scheduling policy.

Offering concepts that are taught in real-time scheduling theory would be practical for development of complex real-time systems, teaching students real-time theory, and making FreeRTOS friendlier for students who have newly learned real-time theories.

Offering a library providing such concepts spawns the choise, whether to use theoretical features or stick to the original kernel Application Program Interface (API). Although it would be more efficient to implement the scheduling library inside the kernel, this implies that the new kernel must be ported to platforms even if they are already supported by FreeRTOS. Moreover, this requires a deep knowledge of the kernel, and could cause loss of safety certification and reliability. By implementing the library in the user space, the library becomes supported by any platform that is already supported by FreeRTOS without altering safety certification and reliability.

Existing scheduler frameworks for FreeRTOS are either not completely implemented in user space, or not providing theoretical real-time scheduling features. None of the existing scheduler frameworks are designed to be a library that is easy to configure, use and switch scheduling policy. Thus, a scheduling library for FreeRTOS implemented in user space is proposed. The term user space does not imply the execution in user mode of the CPU which cannot access protected memory space, in this paper. It implies that FreeRTOS kernel is not modified and in fact, the proposed scheduling library executes in kernel mode of the CPU as all other user applications in FreeRTOS.

1.1 FreeRTOS

FreeRTOS is an open source RTOS that has been ported to more than 30 embedded system architectures. FreeRTOS is written in C and has small footprint and low overhead. Features provided by the kernel are multithreading by tasks, queues, semaphores, software timers, mutexes and event groups [1]. However, several extension modules that offer more advanced operating system functionalities are provided such as:

- FreeRTOS+Nabto (providing IoT solution)
- FreeRTOS+FAT SL (providing FAT file sytem)
- FreeRTOS+UDP (providing socket based UDP/IP interface)
- FreeRTOS+CLI (providing command line interface)
- FreeRTOS+Trace (providing run time diagnostic tool)
- FreeRTOS+IO (providing POSIX like I/O interface)

1.1.1 FreeRTOS Overhead and Footprint

The context switch time between tasks on FreeRTOS is dependent on port, compiler and configuration. For example, a context switch time of 84 CPU cycles can be obtained on ARM Cortex-M3 port with Keil compiler and fully optimized configuration [1]. The FreeRTOS kernel itself has a footprint of 5–10kB on IAR STR71x ARM7 port with full optimization and minimum configuration [1].

1.1.2 Scheduling Policy of FreeRTOS

The scheduling policy provided by FreeRTOS is simple fixed priority scheduling policy, where the user assigns a priority to each task during task creation. The priority must be within the range of 0 and configMAX_PRIORITIES-1, where configMAX_PRIORITIES is defined within FreeRTOSConfig.h file. If port optimized task selection is enabled, configMAX_PRIORITIES cannot be higher than 32 [1]. The highest priority task which is in ready state will always get the CPU time. Tasks that have same priority will share the CPU time equally in round robin manner if time slicing is enabled [1].

1.2 Contribution

This thesis proposes Efficient Scheduling Library for FreeRTOS (ESFree). ESFree is implemented in the user space of FreeRTOS and available with open source license. ESFree consists of one c file and one header file, and provides:

- Periodic tasks with additional parameters that are not included in native FreeRTOS tasks as:
  - Phase
  - Period
  - Deadline
  - Worst-Case Execution Time (WCET)
- Timing error detection and handling for exceeded WCET
- Timing error detection and handling for missed deadline
- Rate-Monotonic Scheduling (RMS) policy
- Deadline-Monotonic Scheduling (DMS) policy
- Earliest Deadline First (EDF) policy
- Aperiodic jobs
- Sporadic jobs
- Polling server

In this thesis, native FreeRTOS designates the original FreeRTOS without any external modules as ESFree.

Chapter 2

Background

This chapter describes theoretical background, the hardware platforms and related work.

2.1 Real-Time System

The main difference between real-time systems and other computer systems is that real-time systems have deadlines on the output of their threads. Threads in real-time systems are often called tasks. According to Buttazzo [2], real-time tasks can be categorized into three different categories depending on the consequence of a deadline miss:

- Hard real-time task: Producing output after the deadline may cause catastrophic consequences on the system under control.
- Firm real-time task: Producing output after the deadline is useless for the system, but does not cause any damage.
- Soft real-time task: Producing output after the deadline has still some utility for the system, although causing a performance degradation.

A system that is able to handle hard real-time tasks is called hard real-time system. An example of a hard real-time task could be management of the airbag deployment during collision in a vehicle. If the airbag is not deployed before the deadline when the vehicle collides, that would be a serious safety defect on the system.

An example of a firm real-time task could be imaging processing within a digital cable set-top box. If the image processing of a frame is not finished before its deadline it will cause jitter, and displaying the frame that has missed its deadline will cause even more jitter so it is discarded. Missing deadline in this system could diminish quality of service if it occurs too frequently.

An example of soft real-time tasks could be tasks in video game applications that shares many resources and need to be finished in scheduled time for the game

to play correctly. If the tasks miss the deadlines, the game might lag a bit, but will still play correctly.

2.2 Real-Time Scheduling

This section describes some well known concepts in real-time scheduling theory as periodic tasks, aperiodic jobs, sporadic jobs, Rate-Monotonic Scheduling (RMS) policy, Deadline-Monotonic Scheduling (DMS) policy, Earliest Deadline First (EDF) policy, and polling server.

2.2.1 Periodic Task

Tasks in real-time systems, do often perform same job periodically. For instance, a task might collect and send data from a sensor every second. Such tasks are called are called periodic tasks [2].

Fig. 2.1 shows an example of how to make a task periodic on FreeRTOS with a period of 500Hz.

Periodic tasks are often denoted with four different constraints in litterature, phase, period, Worst-Case Execution Time (WCET) and deadline.

The phase of a periodic task is the first release time of the periodic task after system start [2], and WCET is the prediction of how long time the periodic task will execute its job in the worst-case [2].

2.2.2 Rate-Monotonic Scheduling (RMS)

Rate-Monotonic Scheduling (RMS) is an optimal fixed priority scheduling policy for single core processors that gives highest priority to the periodic task with shortest period. RMS can guarantee the schedulability of periodic tasks if the CPU utilization is less than 69% [2].

2.2.3 Deadline-Monotonic Scheduling (DMS)

Deadline-Monotonic Scheduling (DMS) is an optimal fixed priority scheduling policy for single core processors that gives highest priority to the periodic task with shortest deadline. DMS can guarantee the schedulability of periodic tasks if the CPU utilization is less than 69% as RMS. RMS is often preferred when the period is shorter than the deadline of the periodic tasks, while DMS is often preferred when the deadline is shorter than the period of the periodic tasks [2].

2.2.4 Earliest Deadline First (EDF)

Earliest Deadline First (EDF) is an optimal dynamic priority scheduling policy for single core processors that gives highest priority to the periodic task with earliest deadline. EDF can guarantee the schedulability of periodic tasks if the CPU utilization is less than 100% [2].

```
#include "scheduler.h"

static void vTestTask( void *pvParam )
{
char* cParam = pvParam;
TickType_t xLastWakeTime;
const TickType_t xPeriod = pdMS_TO_TICKS( 500 );

  /* Initialise the xLastWakeTime variable with the current time. */
  xLastWakeTime = xTaskGetTickCount();

  for( ;; )
  {
    /* Wait for the next cycle. */
    vTaskDelayUntil( &xLastWakeTime, xPeriod );

    /* Perform action here. */
    if( 'a' == *cParam )
    {
      printf( "Hello %c\r\n", *cParam );
    }
    else
    {
      printf( "Bye\r\n" );
    }
  }
}

TaskHandle_t xHandle = NULL;
char cParam = 'a';

int main( void )
{
  xTaskCreate(
    vTestTask,                  /* Function that implements the task. */
    "Test",                     /* Text name for the task. */
    configMINIMAL_STACK_SIZE,   /* Stack size in words, not bytes. */
    &cParam,                    /* Parameter passed into the task. */
    1,                          /* Priority at which the task is created. */
    &xHandle );                 /* Used to pass out the created task's handle. */

  vTaskStartScheduler();
}
```

Figure 2.1. Example code for periodic task on native FreeRTOS.

2.2.5 Aperiodic Jobs

Aperiodic jobs are jobs that can be created and arrive the system at any time. They do not execute periodically as periodic tasks, and do only execute once. Aperiodic jobs are often scheduled and given CPU time at slack time when there is no periodic

tasks running, as aperiodic jobs do not have deadlines [2].

2.2.6 Sporadic Jobs

Sporadic jobs can also be created and arrive the system at any time, do not execute periodically, and do only execute once just like aperiodic jobs, yet sporadic jobs have deadlines. Hence sporadic jobs have higher priority than aperiodic jobs, but lower priority than periodic tasks. An acceptance test to test whether the sporadic job can meet its deadline is often required by the system when serving sporadic jobs [2].

2.2.7 Periodic Server

A periodic server is a server that behaves as a periodic task and processes aperiodic jobs and sporadic jobs which are stored in a queue during its execution time. There exists several periodic server algorithms as polling server, deferrable server, sporadic server and many more [2].

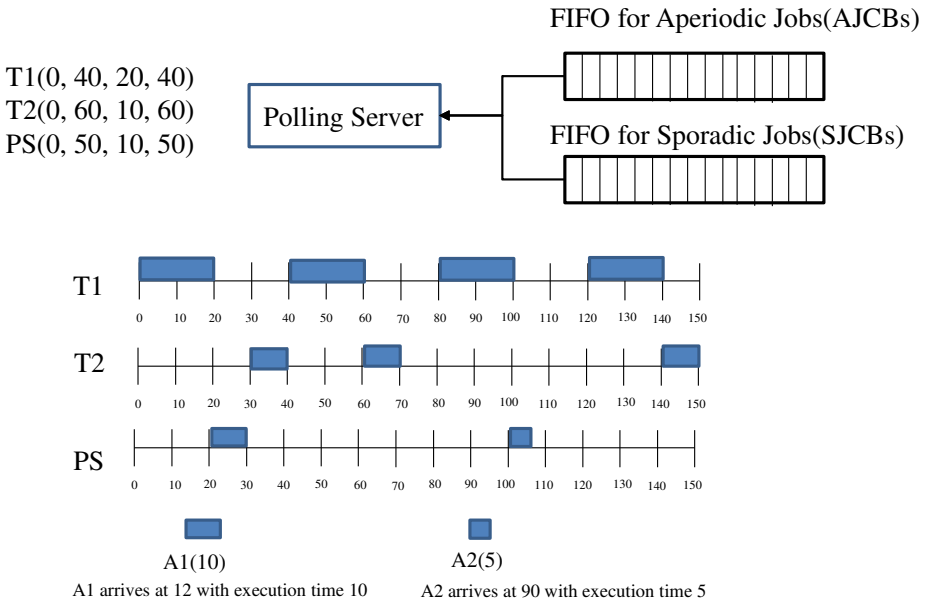


Figure 2.2. Polling server example

2.2.8 Polling Server

Polling server is the simplest periodic server algorithm that serves sporadic jobs or aperiodic jobs if there are any in the queues when the polling server executes. If the queues are empty, the polling server will not preserve its execution budget and

sleep until its next period [2]. Fig. 2.2 illustrates an example of a system containing two periodic tasks and one polling server within RMS policy.

2.3 Task States

A task in FreeRTOS can have following four different states [2]:

- Running: The task is actually executing.
- Ready: The task is able to execute but not currently executing, since a higher priority task is already in running state.
- Blocked: The task is currently waiting for a temporal or external event and cannot be selected to enter the running state. For instance the task might wait for a queue or semaphore. Tasks in the blocked state normally have a timeout period, after which the task will be unblocked even if the event the task was waiting for has not occurred.
- Suspended: The task cannot be selected to enter the running state as tasks in blocked state. However, tasks in suspended state do not have timeout. Tasks can only enter or exit the suspended state by explicitly calling the APIs vTaskSuspend() and xTaskResume().

A transition diagram for valid task state tranisitions in FreeRTOS is displayed in Fig. 2.3.

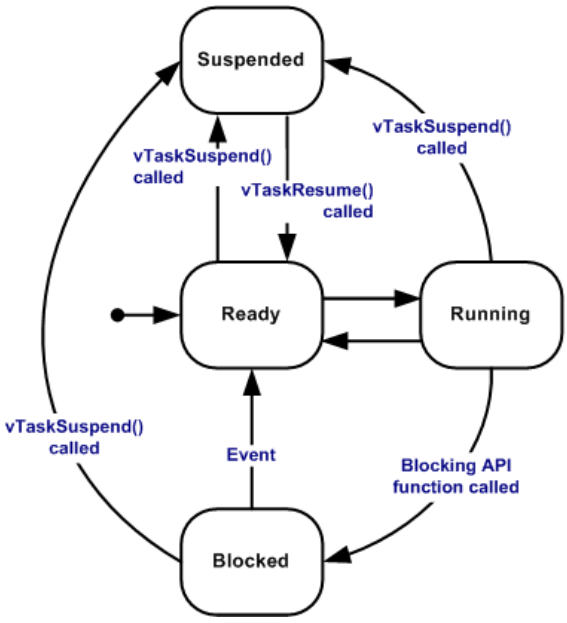


Figure 2.3. Task state transitions in FreeRTOS [1].

2.4 Task Control Block (TCB)

Task Control Block (TCB) is a data structure that stores information of the task that is necessary for the kernel to manage the task. Typical information stored in a TCBs on real-time systems accordig to Buttazzo [2], although only five of them are included in FreeRTOS TCBs (in version 8.2.3), are following:

- An identifier, which is usually a character string
- Memory address to the first instruction of the task
- Task type (periodic, aperiodic, or sporadic)
- Task criticality (hard, soft, or non-real-time)
- Priority
- Current task state (running, ready, block, or suspended)
- WCET
- Task period
- Relative deadline, specified by user
- Absolute deadline, computed by kernel
- Task utilization factor (only for periodic tasks)
- Pointer to the stack, where context of the task is stored
- Pointer to a directed acyclic graph, if there are precedence constraints
- Pointer to a list of shared resources, if a resource access protocol is provided by the kernel

The five parameters that are listed above and included in FreeRTOS TCBs (in version 8.2.3) are following:

- An identifier, which is a character string
- Memory address to the first instruction of the task
- Priority
- Current task state (running, ready, block, or suspended)
- Pointer to the stack, where context of the task is stored

2.5 Task Synchronization

RTOSs do often provide mechanisms as semaphores and mutexes to assist synchronization when tasks use shared resources.

2.5.1 Semaphore

Semaphore is a kernel mechanism for protecting shared resources from race conditions in concurrent programs. A semaphore can be thought as a counter that can only be accessed through two primitives usually called wait() and signal(). To protect the shared resource by a semaphore, the shared resource needs a unique semaphore S. The critical section where the shared resource is being accessed must begin with a wait(S) primitive and end with a signal(S) primitive [2].

In FreeRTOS, the signal(S) primitive is called xSemaphoreGive(S) and the wait(S) primitive is called xSemaphoreTake(S). xSemaphoreGive(S) decrements S, and xSemaphoreTake(S) blocks the calling task if the value of S is at the threshold. If the value of S is lower than the threshold, S is incremented and the task gains access to the shared resource. The threshold can be set to any value during creation if it is a counting semaphore. Binary semaphores has the threshold 1 [1].

2.5.2 Priority Inversion

Priority inversion is a phenomenon where a higher priority task is blocked by a lower priority task when the higher priority task waits for access to a protected shared resource (for instance by semaphore) already in use by a lower priority task. The problem is that a task with medium priority that does not access the shared resource at same time can preempt the lower priority task. The higher priority task will then have to wait for not only the critical section of lower priority task, but also for the execution time of the medium priority task, and thus the higher priority task might miss its deadline [2].

2.5.3 Priority Inheritance Protocol

Priority inheritance protocol is a mechanism that avoids unbound priority inversions by temporarily rising the priority of the lower priority task that causes priority inversion to the highest priority of the tasks it blocks. This prevents medium priority tasks from preempting the lower priority task, and thus minimizes the effect of priority inversion [2].

2.5.4 Mutex

Mutexes in FreeRTOS are binary semaphores that include priority inheritance mechanism [1].

2.6 Interrupt Service Routine (ISR)

An interrupt is an asynchronous signal to the processor emitted by hardware or software that needs immediate attention. The processor will then stop its current code execution, saving its state, and let an Interrupt Service Routine (ISR) execute. When the ISR has finished its execution, the processor will return its state before the interrupt was signaled and continue its previous code execution.

2.6.1 Tick ISR

The FreeRTOS kernel has a tick ISR which is invoked by a timer interrupt. The tick ISR increments the tick counter variable which is used by the kernel to measure time, and checks whether any task should be unblocked. If a task with higher priority than the currently running task has become ready, a context switch is issued if preemption is enabled. The frequency of the timer interrupts that invokes the tick ISR can be configured by the user [1].

2.7 Timing Error Detection and Handling

The idea of timing error detection and handling mechanism for WCET excess and deadline miss was first introduced in [3]. As the name tells, the mechanism detects when a task or job exceeds its WCET or misses its deadline. The method for handling these timing errors can be done in several ways, and may be tuned by the user for application specific needs.

Timing error detection and handling mechanism is effective for soft and firm real-time applications, as it can reduce waste computation time of soft and firm real-time tasks when they already have missed their deadlines. The mechanism is also beneficial for safety critical hard real-time applications, as numerous hard real-time systems often are implemented with "the dangerous assumption that due to correct design, analysis, and testing, a timing error will never occur" [3]. Yet errors in these systems do occur, primarily due to the difficulties in accurately measuring the WCET of the code [3].

2.8 Platform

As ESFree is implemented in user space and FreeRTOS maintains backward compatibility, ESFree should run on any platform that runs FreeRTOS version 8.2.1 or later. ESFree has been tested and verified on following platforms:

- Digilent ZedBoard (ARM Cortex-A9) with FreeRTOS version 8.2.3
- Digilent Nexys 4 DDR (Xilinx MicroBlaze) with FreeRTOS version 8.2.3
- STMicroelectronics STM32 Nucleo-F401RE (ARM Cortex-M4F) with FreeRTOS version 8.2.1

2.8.1 Digilent ZedBoard

Digilent ZedBoard is a development board for Xilinx Zynq-7000 which contains an ARM Cortex-A9 processor core and field programmable gate array (FPGA) in a single device [4]. A photo of Digilent ZedBoard running ESFree is displayed in Fig. 2.5.

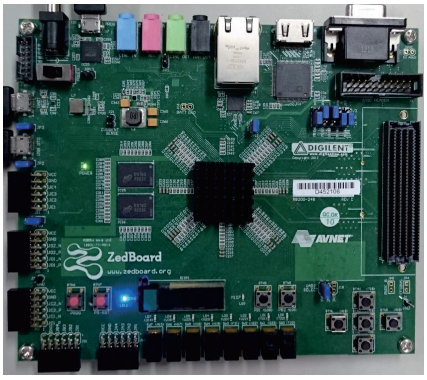


Figure 2.4. Digilent ZedBoard

2.8.2 Digilent Nexys 4 DDR

Digilent Nexys 4 DDR is a development board for Xilinx Artix-7 FPGA [5]. MicroBlaze soft microprocessor core is used as processor core for Xilinx FPGAs[6]. A photo of Digilent Nexys 4 DDR running ESFree is displayed in Fig. 2.5.

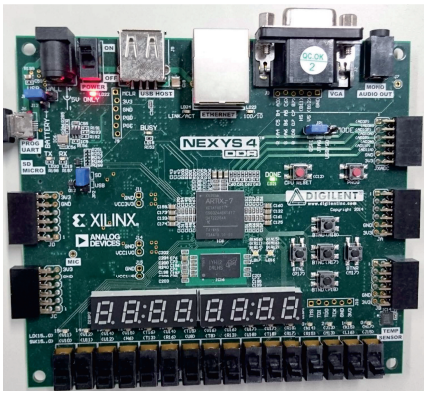


Figure 2.5. Digilent Nexys 4 DDR

2.8.3 STMicroelectronics STM32 Nucleo-F401RE

STMicroelectronics STM32 Nucleo-F401RE is a development board for ARM Cortex-M4F processor core [7]. A photo of STMicroelectronics STM32 Nucleo-F401RE

running ESFree is displayed in Fig. 2.6.

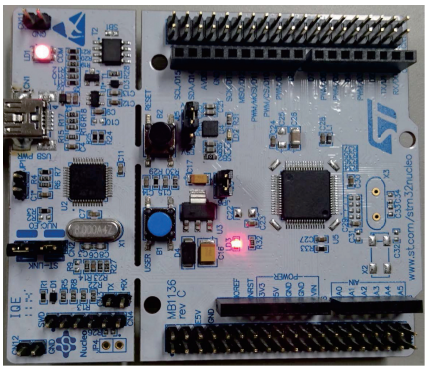


Figure 2.6. STMicroelectronics STM32 Nucleo-F401RE

2.9 Related Work

Table 2.1. Comparison between ESFree and HST

	ESFree	HST
Scheduling Policy	RMS, DMS, EDF (two implementations)	RMS, EDF
Timing error detection and handling	WCET, Deadline	Deadline
Scheduling method for aperiodic or sporadic job	Polling server	Slack stealing, Dual priority
Phase for periodic tasks	Yes	No
Aperiodic jobs	Yes	Yes
Sporadic jobs	Yes	No
Handling counter overflow	Yes	No
Can use priority inheritance mechanism	Yes	No

A similar scheduler implemented in the user space of FreeRTOS called Heterogenous Scheduler Task (HST) is proposed in [8]. HST has example implementations for RMS, EDF, Dual Priority scheduling and scheduling of aperiodic jobs with slack stealing on RMS. The main difference between HST and ESFree is that HST is designed for assisting development and tuning of scheduling policies from user space, while ESFree is designed to have simple API for convenient usage with efficient implementation and diverse features. A comparison of features between ESFree and HST is presented in Table 2.1.

HST needs to load a different implementation of scheduler logic to change the scheduling policy, while ESFree requires the user to modify configurations in the header file to test and experiment with various scheduling features. Furthermore, the user is required to configure trace macros to call HST API for using HST, which is not user friendly for beginners in FreeRTOS, especially as it is not documented

by HST. Trace macros are macros inside the kernel API that can be defined by the user, originally intended to be used for tracing and debugging [1]. Moreover, HST can not use priority inheritance mechanism provided in FreeRTOS mutexes, as HST does not set priorities on its tasks. ESFree uses trace macros in a similar way to HST only when the user desires efficiency for EDF, and offers an alternative EDF without trace macro usage which is user friendlier and compatible with FreeRTOS priority inheritance mechanism. At last, HST's example implementations does not take care of counter overflows of execution time and absolute deadline, which is a serious defect for real-time applications, although HST's example implementations are made for demonstration purpose.

Several previous work has proposed scheduling frameworks similar to HST but for other operating systems. An API for application defined scheduling algorithms in real-time POSIX, with implementation in MaRTE OS is proposed in [9]. An improved version of this API is presented in [10]. An external scheduler framework Exsched implemented for Linux and VxWorks is presented in [11]. An efficient implementation of application defined scheduling library for RTLinux is proposed in [12].

Numerous scheduler frameworks for real-time systems have been developed to provide hierarchical scheduling. Hierarchical Scheduling Framework (HSF) support for FreeRTOS that modifies the kernel without changing the API is proposed in [13]. Other scheduling frameworks as FSF [14] and SF3P [15] provides hierarchical scheduling on Marte OS, Shark and POSIX-compliant operating systems. Though hierarchical scheduling is rather an categorizing technique of tasks than theoretical scheduling.

Some ideas are borrowed from an efficient EDF implementation inside the kernel of ERIKA Enterprise presented in [16] for the EDF implementations in ESFree.

The timing error detection and handling implemented in ESFree is based on the idea introduced in [3].

Chapter 3

Efficient Scheduling Library for FreeRTOS (ESFree) : System Architecture

This chapter describes the architecture of ESFree.

3.1 Extended TCB

ESFree has its own defined extended TCB called SchedTCB, that includes addtional information for managing periodic tasks from the user space. ESFree's own Aperiodic Job Control Block (AJCB) and Sporadic Job Control Block (SJCB) are defined for managing aperiodic jobs and sporadic jobs.

Some of the additional information in SchedTCB are:

- Pointer to task function of the periodic task (the function defined by the user)
- Priority of the task
- Task handler
- Release time
- Last release time
- Relative deadline
- Absolute deadline
- Period
- WCET
- Execution time counter

AJCB contains following information of aperiodic jobs:

- Pointer to job function of the aperiodic job
- Name of the aperiodic job
- Parameters to the job function
- WCET
- Execution time counter

SJCB contains following information of sporadic jobs:

- Pointer to the job function of the sporadic job
- Name of the sporadic job
- Parameters to the job function
- Relative deadline
- Absolute deadline
- WCET
- Execution time counter

SchedTCBs are managed with an array if the scheduling policy is fixed priority type as RMS or DMS, since it results in lower footprint and faster full scan of all periodic tasks than other data structures. Sequential search through the array for a specific SchedTCB is only required when deleting a periodic task. Full scan of all periodic tasks is required when detecting timing errors.

When the scheduling policy is dynamic priority type as EDF, ESFree uses sorted linked list that is used by the FreeRTOS kernel for managing tasks for managing SchedTCBs. The reason is that it is an efficient way of having the periodic tasks sorted by deadline, and the data structure provided by FreeRTOS kernel can be used which results in smaller footprint.

AJCBs and SJCBs are always managed with First-In First-Out (FIFO) queues, as they should be served in FIFO manner.

3.2 Scheduler Task

The scheduler task in ESFree is only needed if EDF, timing error detection and handling for WCET excess, timing error detection and handling for deadline miss, or polling server is used. The scheduler task is implemented as a task with highest priority, and its task function includes EDF and timing error detection and handling management depending on what features are enabled.

ESFree uses task notifications to manage release and block of the scheduler task. Task notification is a feature that can replace semaphores, event groups and queues in some situations, yet more efficiently implemented. Details of the task notification feature are described in [1].

3.3 Fixed Priority Scheduling Policies

A considerable difference between ESFree and other user space schedulers as HST [8] is that ESFree has varied implementations for fixed priority scheduling policies and dynamic priority scheduling policies to gain efficiency.

When using fixed priority scheduling policies as RMS or DMS, ESFree calculates and assigns priorities to each periodic task before native FreeRTOS scheduler starts. Since scheduling of tasks in the RMS and DMS implementations of ESFree is done by the native FreeRTOS scheduler, the overhead required during context switches is nearly the same as native FreeRTOS. The additional overhead is caused by few instructions executed within the ESFree API that wraps around the user defined task function. On the contrary, HST does always need to switch in the HST scheduler task in between a context switch between tasks, no matter what kind of scheduling policy is used. The overhead of context switches in HST is therefore approximately twice as much as native FreeRTOS scheduler [8]. A comparison between context switches in HST and ESFree within RMS policy is illustrated in Fig. 3.1.

3.4 Dynamic Priority Scheduling Policies

There are two different kind of implementations for EDF in ESFree, efficient implementation and naive implementation.

3.4.1 Efficient EDF

The efficient implementation of EDF uses the trace macro feature of FreeRTOS as HST also does, to be notified when tasks become ready, blocked or suspended. There are only two priorities for periodic tasks in this implementation, running priority and not running priority. All tasks are initially assigned not running priority. The scheduler task is switched in and makes sure that the periodic task with earliest absolute deadline in ready or running state has running priority when:

- A task becomes blocked
- A task suspends itself
- A task becomes ready while no other tasks are running
- A task with earlier absolute deadline than current running task becomes ready

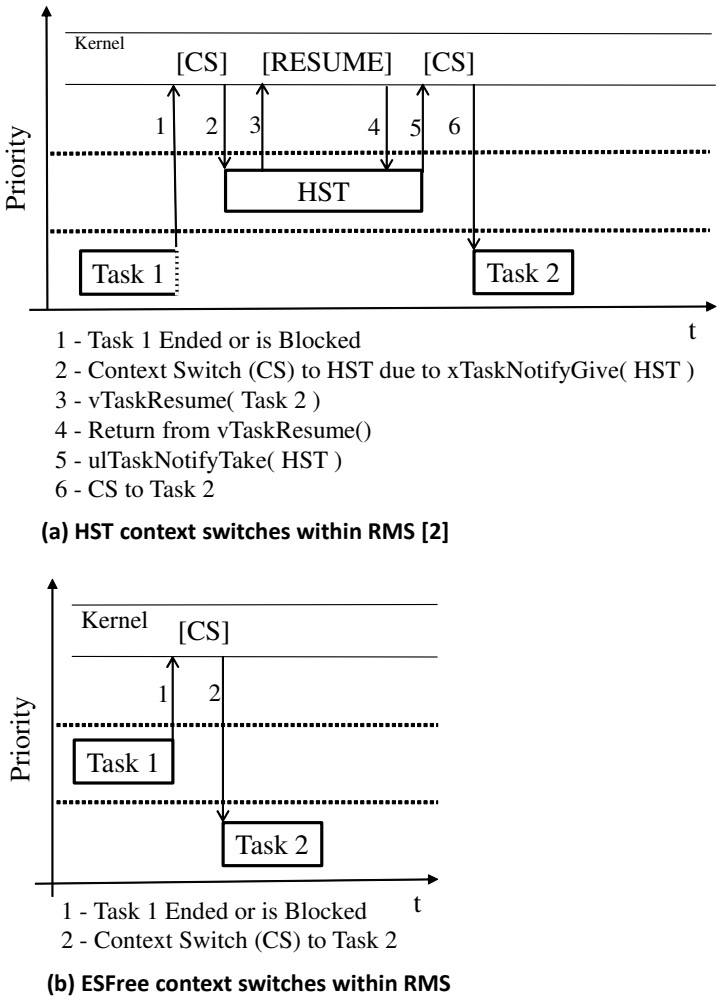


Figure 3.1. HST and ESFree context switch examples

The model of context switches is the same as HST has which is described in Fig. 3.1(a).

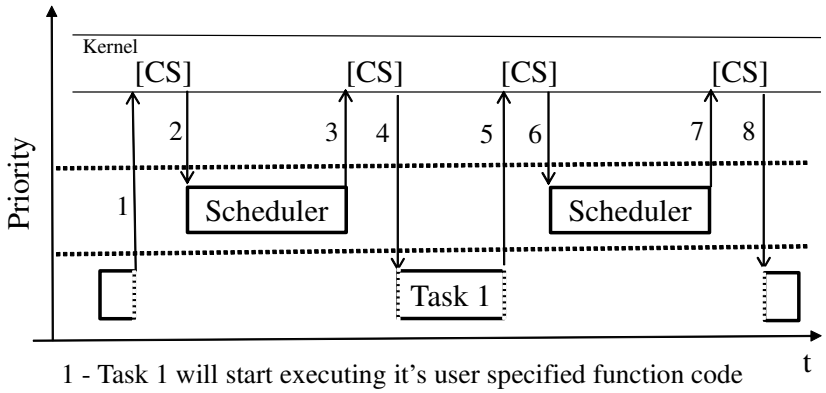
However, the user needs to configure the trace macros, which might require deeper knowledge of the kernel and application source code in case the library is combined with other applications that also uses trace macros as FreeRTOS+Trace.

Overflows of absolute deadline and execution time counter in ESFree are mostly handled by Implicit Circular Timer's Overflow Handler (ICTOH) method proposed by Carlini and Buttazzo, which handles counter overflows with a very small overhead compared to other methods [17]. Yet, in some cases where one relational operator on two variables can be used to detect overflow, that is prefered instead of ICTOH method.

3.4.2 Naive EDF

The naive implementation of EDF does always make context switch to the scheduler task of ESFree at the beginning and end of a periodic task. The scheduler task will then calculate and assign priority for every periodic task according to their absolute deadlines when it is switched in. This procedure is necessary for assuring that EDF policy is maintained when trace macros are not used, since the only way for the user to get notification when tasks become ready, blocked or suspended is by using trace macros.

Fig. 3.2 illustrates the overhead caused by naive EDF implementation. The overhead of this implementation is substantial compared to the efficient EDF implementation. The advantage of this implementation is that the user does not need to configure trace macros. Another advantage is that priority inheritance mechanism which is implemented in FreeRTOS mutexes is compatible with naive EDF, yet not with efficient EDF, since all periodic tasks have individual priorities in naive EDF, while there are only two priorities in efficient EDF.



1 - Task 1 will start executing it's user specified function code
2 - Context switch (CS) to scheduler task
3 - Scheduler task has updated all priorities
4 - CS to Task 1 (if it still has highest priority)
5 - The user specified function code of Task 1 has finished
6 - CS to scheduler task
7 - Scheduler task has updated all priorities
8 - CS to Task 1 (if it still has highest priority)

Figure 3.2. Context switch example on naive EDF

3.5 Aperiodic Job

Aperiodic jobs in ESFree are pure functions managed by the polling server. They are not implemented as tasks, and the WCET information in AJCB is only used for acceptance test for sporadic jobs.

3.6 Sporadic Job

Sporadic jobs are identically implemented as aperiodic jobs with one distinction, that sporadic jobs have deadlines. Hence sporadic jobs have higher priority than aperiodic jobs, and an acceptance test to check whether the sporadic job can meet its deadline is carried out when the API for creating sporadic jobs is called. A sporadic job is accepted and put into the FIFO queue of SJCBs if its deadline is less than its maximum response time. The maximum response time is calculated based on formula presented in [2]:

- If WCET of the sporadic job is less than or equal to the execution budget of the polling server, the maximum response time is (polling server period * 2 + worst-case release time)

- If WCET of the sporadic job is greater than the execution budget of the polling server, the maximum response time is { polling server period + (WCET / polling server execution budget + 1) * polling server period + worst-case release time}

The worst-case release time of the sporadic job is approximated with a parameter holding maximum response time of previously enqueued sporadic jobs or currently running aperiodic job.

3.7 Polling Server

The polling server is implemented as a periodic task with following sequence of instructions in the task function:

1. If the FIFO queue of SJCBs is not empty, go to 2), otherwise go to 3)

2. Execute the job function of the first sporadic job in the queue, then go back to 1)

3. If the FIFO queue of AJCBs is not empty, go to 4), otherwise return

4. Execute the job function of the first aperiodic job in the queue, then go back to 1)

Although polling server is the simplest periodic server algorithm which might give longer response time, evaluation studies show that polling server offers higher schedulability of tasks than deferrable server and sporadic server when used in hierarchical scheduling, due to its low overhead [18].

3.8 Timing Error Detection and Handling on ESFree

ESFree provides two types of timing error detection and handling, one for detecting and handling periodic tasks that exceeded their WCET, and the other for detecting and handling tasks that have missed their deadline.

An example of WCET excess handling is described in Fig. 3.3. If WCET excess of a periodic task is detected, it will be suspended by the scheduler task until the next period of the suspended task.

An example scenario of deadline miss handling is described in Fig. 3.4. If deadline miss of a periodic task is detected, it will be deleted and recreated with phase set to beginning of its next period by the scheduler task. If sporadic tasks are used, deadline miss of sporadic tasks are also detected, yet no additional handling except notification to user by message printing is provided.

Fig. 3.5 describes an example scenario when both timing error detection and handling features are combined. WCET excess of the periodic task T1 is first detected and suspended. When T1 also misses its deadline, it is deleted and recreated with release time set to beginning of its next period.

Detection of exceeded WCET is implemented inside the tick ISR hook function. The handling of WCET excess is implemented inside the scheduler task which is switched in by the tick ISR hook function when WCET excess is detected.

As detection of deadline miss requires to scan all periodic tasks and sporadic jobs which causes a larger overhead compared to WCET detection that only needs to check the running periodic task, detection and handling of deadline miss is implemented inside the scheduler task function. Thus, the tick ISR hook function switches in the scheduler task when the scheduler task period arrives, if timing error detection and handling for deadline miss is enabled. The scheduler task period can be configured by the user.

Hence detection and handling of WCET excess is done with the frequency of tick interrupt, while detection and handling of deadline miss is done with the frequency of scheduler task period. However, if EDF is enabled, context switch to the scheduler task is more frequent, and deadline miss detection and handling is done whenever the scheduler task is switched in.

Timing error detection and handling for WCET excess must be enabled when using polling server for proper execution, since polling server is implemented as periodic task, and the only way to manage the execution budget of polling server is by using the timing error detection and handling.

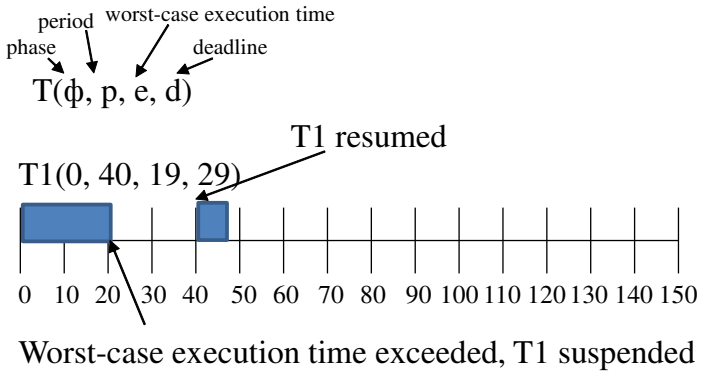


Figure 3.3. Timing error detection and handling for WCET excess.

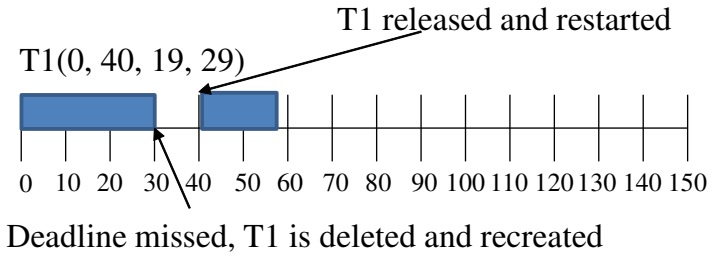


Figure 3.4. Timing error detection and handling for deadline miss.

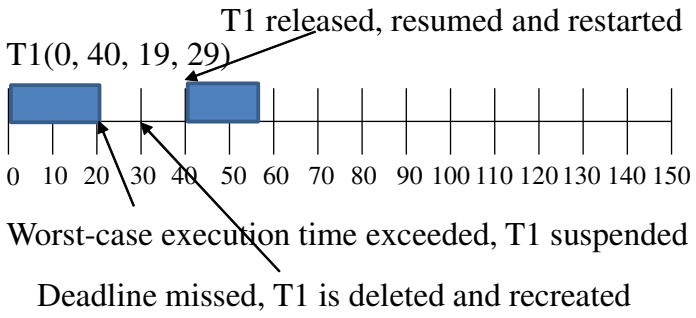


Figure 3.5. Combination of both timing error detection and handling features.

Chapter 4

ESFree API & Configurations

This chapter describes the API and configurations of ESFree that the user needs to be aware of to use ESFree.

4.1 ESFree API

ESFree has a simple API that includes 6 functions:

- vSchedulerInit(): Initializes linked lists or an array for SchedTCB management. This function must be called before any other API function from this library.
- vSchedulerPeriodicTaskCreate(): Creates SchedTCB for a periodic task.
- vSchedulerPeriodicTaskDelete(): Deletes a periodic task.
- vSchedulerStart(): Creates all periodic tasks and starts the system.
- vSchedulerAperiodicJobCreate(): Only available if usage of aperiodic jobs is enabled. Creates AJCB for an aperiodic job.
- xSchedulerSporadicJobCreate(): Only available if usage of sporadic jobs is enabled. Creates SJCB for a sporadic job.

4.1.1 Creating Periodic Tasks

An example code for creating a periodic task is given in Fig. 4.1. pdMS_TO_TICKS is a macro defined by FreeRTOS that converts ms to ticks. Priority set by the user when creating a periodic task is only used when the scheduling policy is set to manual. The parameter is ignored in any other case.

In most RTOSs, the user needs to add a sleep to adjust the phase, and an infinite loop with sleep to adjust the period in the task function code of a periodic task. However, as seen in Fig. 4.1 ESFree takes care of both phases and periods, so that the user only needs to input the pure functionality in the task function code.

```
#include "scheduler.h"

static void vTestTask( void *pvParam )
{
char* cParam = pvParam;

  if( 'a' == *cParam )
  {
    printf( "Hello %c\r\n", *cParam );
  }
  else
  {
    printf( "Bye\r\n" );
  }
}

TaskHandle_t xHandle = NULL;
char cParam = 'a';

int main( void )
{
  vSchedulerInit();

  vSchedulerPeriodicTaskCreate(
    vTestTask, /* The task function. */
    "Test", /* Name of the task. */
    configMINIMAL_STACK_SIZE, /* Stack size. */
    &cParam, /* Parameter. */
    1, /* Priority of the task. */
    &xHandle, /* Pointer to task handle. */
    pdMS_TO_TICKS( 50 ), /* Phase. */
    pdMS_TO_TICKS( 500 ), /* Period. */
    pdMS_TO_TICKS( 100 ), /* WCET. */
    pdMS_TO_TICKS( 500 )); /* Deadline. */

  vSchedulerStart();
}
```

Figure 4.1. Example code for creating a periodic tasks.

4.1.2 Creating Aperiodic Jobs and Sporadic Jobs

Aperiodic jobs and sporadic jobs can only be created if they are enabled. An example of vSchedulerAperiodicJobCreate() is given in Fig. 4.2, and an example of SchedulerSporadicJobCreate() is given in Fig. 4.3. vSchedulerAperiodicJobCreate() and xSchedulerSporadicJobCreate() can be called from main() function, or at any time from any task.

```
#include "scheduler.h"

static void vAperiodic1( void *pvParam )
{
  /* Task function code here. */
}

int main( void )
{
                    .
                    .
                    .
  vSchedulerAperiodicJobCreate(
    vAperiodic1, /* The job function. */
    "A1", /* Name of the job. */
    NULL, /* Parameters. */
    pdMS_TO_TICKS( 100 ) ); /* WCET */
                    .
                    .
                    .
}
```

Figure 4.2. Example code for creating an aperiodic job.

4.2 ESFree Configurations

4.2.1 scheduler.h

All the user needs to do for using ESFree is to include the scheduler.h file which must be kept with scheduler.c in same directory. The configurations of ESFree are done in the scheduler.h file. The user can choose scheduling policy by setting the define schedSCHEDULING_POLICY to:

- schedSCHEDULING_POLICY_MANUAL: for manual scheduling policy, i.e. the user sets the priorites
- schedSCHEDULING_POLICY_RMS: for RMS
- schedSCHEDULING_POLICY_DMS: for DMS
- schedSCHEDULING_POLICY_EDF: for EDF

If the scheduling policy is EDF, one of the defines schedEDF_NAIVE or schedEDF_EFFICIENT must be set to 1 and the other to 0. 1 means enable and 0 means disable.

All other configurations:

- schedMAX_NUMBER_OF_PERIODIC_TASKS: must be set to maximum number of periodic tasks that will be created including polling server

```
static void vSporadic1( void *pvParam )
{
 /* Task function code here. */
}

int main( void )
{
BaseType_t xResult;
                    .
                    .
                    .
  xResult = xSchedulerSporadicJobCreate(
    vSporadic1, /* The job function. */
    "S1", /* Name of the job. */
    NULL, /* Parameters. */
    pdMS_TO_TICKS( 100 ), /* WCET */
    pdMS_TO_TICKS( 300 )); /* Deadline */

  if( pdFALSE == xResult )
  {
    printf("Sporadic Job not accepted.\r\n");
  }
                    .
                    .
                    .
}
```

Figure 4.3. Example code for creating a sporadic job.

- schedUSE_APERIODIC_JOBS: set to 1 enables aperiodic jobs
- schedUSE_SPORADIC_JOBS: set to 1 enables sporadic jobs
- schedMAX_NUMBER_OF_APERIODIC_JOBS: must be set to maximum number of aperiodic jobs that can be created, if aperiodic jobs are enabled
- schedMAX_NUMBER_OF_SPORADIC_JOBS: must be set to maximum number of sporadic jobs that can be created, if sporadic jobs are enabled
- schedUSE_TIMING_ERROR_DETECTION_DEADLINE: set to 1 enables timing error detection and handling for deadline miss
- schedUSE_TIMING_ERROR_DETECTION_EXECUTION_TIME: set to 1 enables timing error detection and handling for WCET excess

PS is automatically enabled if aperiodic jobs or sporadic jobs is enabled, otherwise disabled. Following defines must be configured if polling server is enabled:

- schedPOLLING_SERVER_PERIOD: must be set to period of polling server

- schedPOLLING_SERVER_DEADLINE: must be set to deadline of polling server in software ticks
- schedPOLLING_SERVER_STACK_SIZE: must be set to stack size of polling server in words
- schedPOLLING_SERVER_MAX_EXECUTION_TIME: must be set to execution budget of polling server in software ticks
- schedPOLLING_SERVER_PRIORITY: must be set to the priority of polling server, if the scheduling policy is manual

The deadline of polling server is only used for calculating the priority of polling server if the scheduling policy is DMS or EDF. Timing error detection and handling of missed deadlines will not affect polling server.

The scheduler task is automatically enabled when EDF, timing error detection and handling for WCET excess, timing error detection and handling for deadline miss, or polling server is enabled, otherwise disabled. Following defines must be configured if scheduler task is enabled:

- schedSCHEDULER_TASK_PERIOD: must be set to period of the scheduler task in software ticks if timing error detection and handling for deadline miss is enabled
- schedSCHEDULER_TASK_STACK_SIZE: must be set to stack size of the scheduler task in words
- schedYIELD_FROM_ISR(xSwitchingRequired): must be port specifically configured to the macro that can cause a context switch at the end of an ISR, which is either portYIELD_FROM_ISR or portEND_SWITCHING_ISR

schedSCHEDULER_PRIORITY is by default set to (configMAX_PRIORITIES - 1) and should be untouched.

4.2.2 FreeRTOS Configurations

There are some configurations of FreeRTOS that must be considered when using ESFree. Maximum number of priorities should be set to number of periodic tasks including polling server + 1, if the scheduling policy is not efficient EDF. For efficient EDF 3 priorities are enough. Following defines must be set to 1 in FreeRTOSConfig.h to enable necessary kernel features that ESFree uses:

- configUSE_PREEMPTION
- configUSE_TICK_HOOK
- configUSE_TASK_NOTIFICATIONS (leaving this undefined works just as fine as FreeRTOS defines it to 1 automatically when undefined)

- configNUM_THREAD_LOCAL_STORAGE_POINTERS
- INCLUDE_vTaskPrioritySet
- INCLUDE_vTaskDelete
- INCLUDE_vTaskSuspend (only necessary if timing error detection and handling is enabled)
- INCLUDE_vTaskDelayUntil
- INCLUDE_xTaskGetCurrentTaskHandle
- INCLUDE_xTaskGetIdleTaskHandle

4.2.3 Configurations for Efficient EDF

Following defines need to be set to 1 in FreeRTOSConfig.h when using efficient EDF:

- configUSE_TRACE_FACILITY
- INCLUDE_xTimerGetTimerDaemonTaskHandle
- INCLUDE_xTaskGetSchedulerState

Also trace macros should be defined at the of FreeRTOSConfig.h as follows:

- #define traceBLOCKING_ON_QUEUE_RECEIVE(xQueue) vSchedulerBlockTrace();
- #define traceBLOCKING_ON_QUEUE_SEND(xQueue) vSchedulerBlockTrace();
- #define traceTASK_DELAY_UNTIL() vSchedulerBlockTrace();
- #define traceTASK_SUSPEND(xTask) vSchedulerSuspendTrace(xTask);
- #define traceMOVED_TASK_TO_READY_STATE(xTask) vSchedulerReadyTrace(xTask);

Chapter 5

Evaluation

The footprint of ESFree is 1.3–4.2kB depending on what features are enabled, and the size of a SchedTCB is 55–79B.

Table 5.1. Platform specification

	Digilent ZedBoard	Digilent Nexys 4 DDR	STMicroelectronics STM32 Nucleo-F401RE
CPU	ARM Cortex-A9	Xilinx MicroBlaze	ARM Cortex-M4F
Clock rate	800MHz	84MHz	84MHz
Memory	512MB DDR3-SDRAM 667MHz	256kB LMB BRAM	512kB Flash

Table 5.2. Context switch overhead (in CPU cycles) on ZedBoard (cache disabled).

	FreeRTOS	ESFree RMS	ESFree EDF Naive	ESFree EDF Efficient
10 tasks average	7,010	7,403	69,567	28,441
10 tasks standard deviation	298	561	2,534	921
30 tasks average	8,519	8,723	210,349	32,044
30 tasks standard deviation	1,579	1,202	18,874	3,010

Table 5.3. Context switch overhead (in CPU cycles) on ZedBoard (cache enabled).

	FreeRTOS	ESFree RMS	ESFree EDF Naive	ESFree EDF Efficient
10 tasks average	558	568	5,939	2,412
10 tasks standard deviation	27	30	161	53
30 tasks average	655	676	16,520	2,627
30 tasks standard deviation	78	88	621	157

The average context switch overhead of ESFree is measured in two test cases, one test case with a set of 10 periodic tasks, and the other with a set of 30 periodic tasks. The method for measuring context switch time is illustrated in Fig. 5.1. The

Table 5.4. Context switch overhead (in CPU cycles) on STM32 Nucleo-F401RE (cache disabled).

	FreeRTOS	ESFree RMS	ESFree EDF Naive	ESFree EDF Efficient
10 tasks average	464	496	6,799	1,809
10 tasks standard deviation	32	33	229	63
30 tasks average	578	612	24,923	2,040
30 tasks standard deviation	102	102	864	210

Table 5.5. Context switch overhead (in CPU cycles) on STM32 Nucleo-F401RE (cache enabled).

	FreeRTOS	ESFree RMS	ESFree EDF Naive	ESFree EDF Efficient
10 tasks average	433	457	5,749	1,630
10 tasks standard deviation	29	30	225	47
30 tasks average	519	546	20,266	1,803
30 tasks standard deviation	78	77	759	161

Table 5.6. Context switch overhead (in CPU cycles) on Nexys 4 DDR.

	FreeRTOS	ESFree RMS	ESFree EDF Naive	ESFree EDF Efficient
10 tasks average	520	562	3,433	1,675
10 tasks standard deviation	23	22	177	49
30 tasks average	607	645	10,658	1,842
30 tasks standard deviation	103	75	629	152

context switch time (in CPU cycles) between periodic tasks is obtained 100 times, by letting every periodic task resetting and starting the cycle counter at the end of its function, and reading the cycle counter at the beginning of its function. However, the last periodic task in each period is excluded from the data set, since otherwise idle time between two periods will be reflected in the measured value when the last periodic task in the period resets and starts the cycle counter, and the first periodic task in the next period reads the cycle counter.

Specifications of the different platforms ESFree is evaluated on are presented in Table 5.1. Table 5.2 shows the comparison of worst-case context switch time measurements on Digilent ZedBoard platform using ESFree and native FreeRTOS, with instruction cache, data cache and L2 cache disabled. Here, the worst-case context switch time for FreeRTOS and ESFree is when a higher priority task has finished its function and the CPU time goes to a lower priority task. Table 5.3 shows same comparison with instruction cache and data cache enabled. Table 5.4 and Table 5.5 shows equivalent results from measurements on STMicroelectronics STM32 Nucleo-F401RE platform. Table 5.6 shows same measurements for Digilent

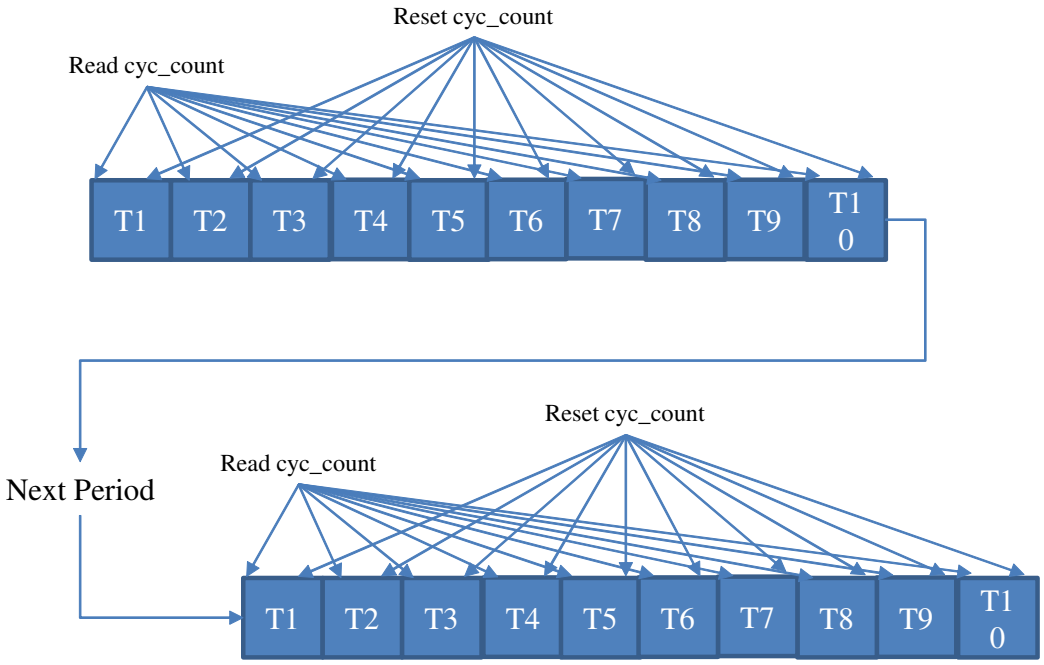


Figure 5.1. Evaluation method.

Nexys 4 DDR with whole application stored inside the Local Memory Bus (LMB) Block RAM, thus enabling and disabling cache gives same result.

I do not measure DMS policy since it has same overhead as RMS. FreeRTOS is configured with trace macros, asserts and stack overflow checking disabled, except efficient EDF where trace macros must be enabled. GCC compiler optimization was O2 for all measurements.

As the evaluation results show, the context switch overhead of fixed priority scheduling policies as RMS and DMS in ESFree exceed overhead of native FreeRTOS with only 2–8%, which is much lower than HST's overhead of 100% of native FreeRTOS [8]. Naive EDF requires large context switch overhead that grows linearly with amount of tasks, and the overhead is approximately 10–15 times more than native FreeRTOS overhead when running 10 periodic tasks. Yet, efficient EDF manages to reduce the context switch overhead from 10–15 times to 4–5 times of native FreeRTOS. As EDF causes less context switches than RMS and promises schedulability on a CPU utilization factor up to 100% [19], the efficient EDF implementation is still effective for practical embedded applications despite its overhead. Evaluation results on EDF in HST are not documented.

Fig. 5.3 demonstrates the correctness of ESFree behaviour to the user specified requirements. ESFree is configured using naive EDF with timing error detection and handling for WCET excess, timing error detection and handling for deadline miss, sporadic jobs and aperiodic jobs enabled. The system contains 4 periodic

tasks and 1 polling server. The polling server has 4 aperiodic jobs queueing in the AJCB FIFO and 2 sporadic jobs queueing in the SJCB FIFO. The periodic task T4 starts executing when the system starts as it has the earliest absolute deadline, and context switches to T2 which has the earliest absolute deadline when T4 has finished its job at 67ms after system start. T2 preempts T1 at 150ms after system start as the red arrow indicates. The polling server starts serving the sporadic job S1 at 194ms after system start. WCET excess of T3 is detected and handled at 433ms after system start, and the deadline miss of T3 is detected and handled at 501ms after system start. The exhausted budget of the polling server is detected and handled as a WCET excess at 563ms after system start. A new sporadic job S3 arrives at 597ms after system start. At 1014ms after system start S1 is finishes and the polling server starts to serve S2 until it exhausts its budget again at 1084ms after system start. A screen shot showing outputs from this test printed through UART (Universal Asynchronous Receiver/Transmitter) is in Fig. 5.2.

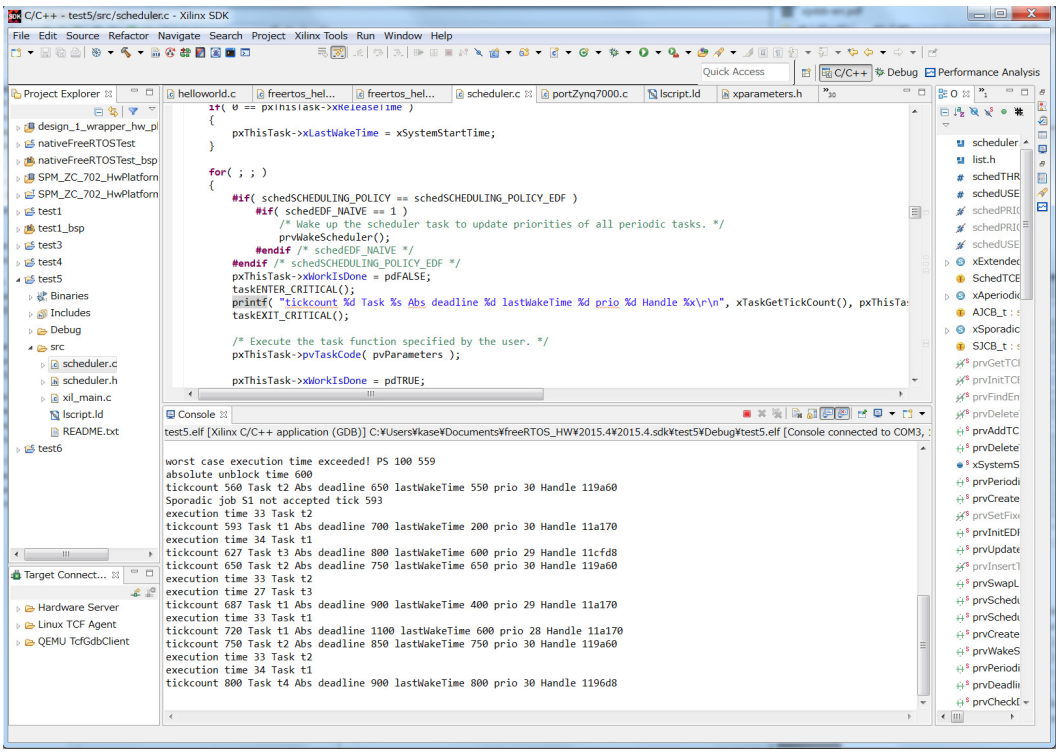


Figure 5.2. Outputs from ESFree verification on screen.

However, a corresponding test for efficient EDF presented in Fig. 5.4, shows a slightly different behaviour. The naive EDF implementation in Fig. 5.3 context switches from T1 to T3 at 627ms after system start, as T1 has finished its job and updated its deadline to 900ms after system start, and then T3 has the earliest deadline which is 800ms after system start. Yet the efficient EDF implementation

in Fig. 5.4 at the same timing continues executing the next job of T1 which is not the correct behaviour of an EDF policy. The reason is that the FreeRTOS API function vTaskDelayUntil() which is internally called by T1 when its job is finished, does not block if the next release time already is passed by the tick counter. As the scheduling activities in efficient EDF is only invoked when trace macros are called, and the trace macro for vTaskDelayUntil() is only called if a task blocks due to vTaskDelayUntil(), the scheduling activities in efficient EDF cannot be invoked if vTaskDelayUntil() does not block. Yet this problem does only occur if the deadline is longer than the period of a periodic task, which is the case of T1. To conclude, although efficient EDF is faster than naive EDF, efficient EDF does not always behave correctly for periodic tasks that have longer deadline than its period, which could be another reason to prefer naive EDF.

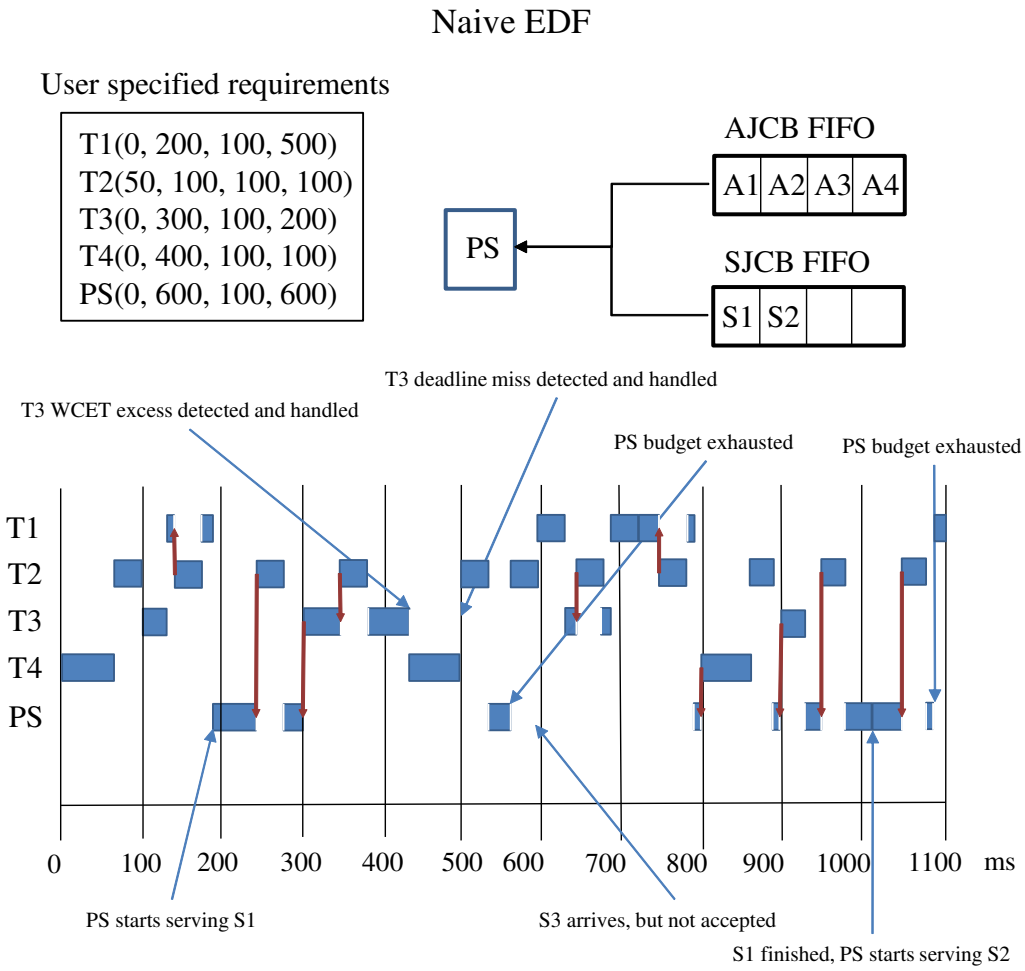


Figure 5.3. ESFree demonstration using all features with naive EDF.

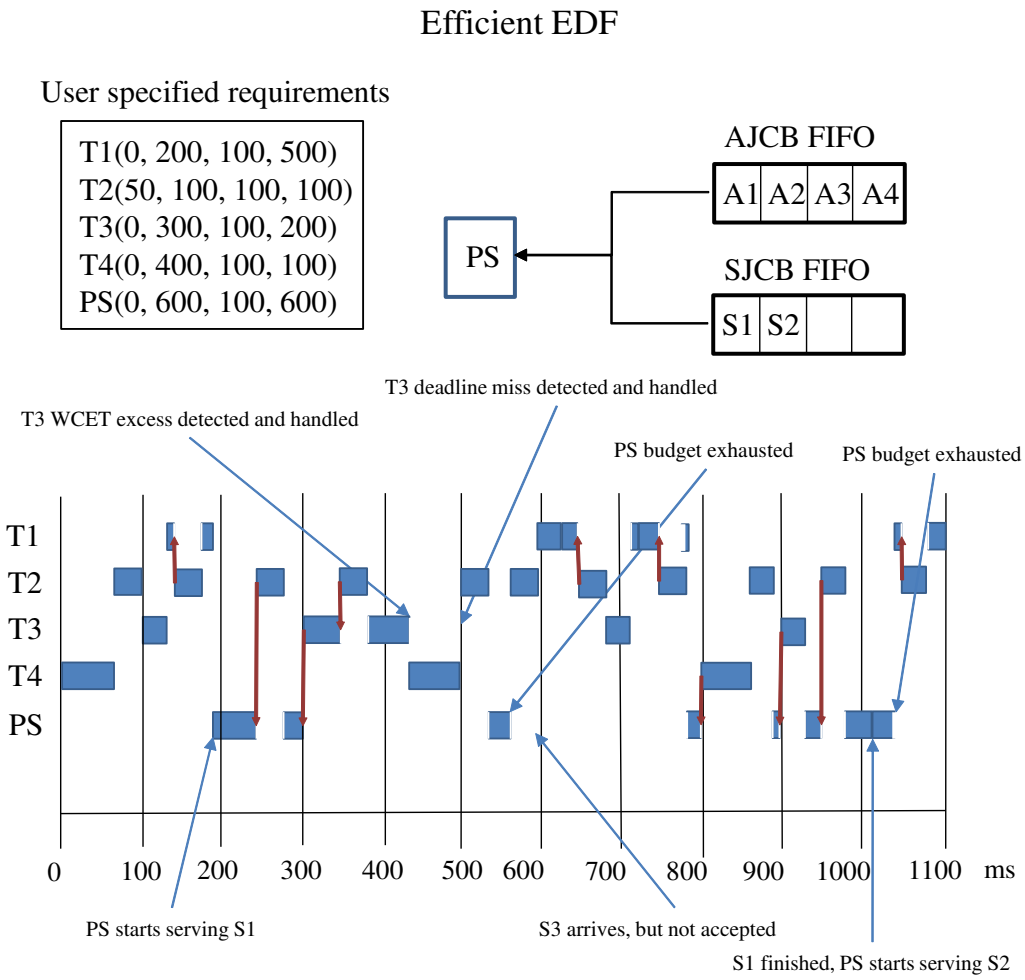


Figure 5.4. ESFree demonstration using all features with efficient EDF.

Chapter 6

Conclusion & Future Work

There is a gap between the scheduling theories taught in school and what practical RTOSs as FreeRTOS provide. Thus these RTOSs are less attractive for teaching purpose. Offering a scheduling library implemented in user space which lets the user to choose whether to skip the overhead of complex implementations, or use more advanced real-time scheduling theories, can reduce this drawback and speed up development of complex projects. Although there already exists several scheduling frameworks for FreeRTOS, they are either difficult to use, not completely implemented in user space, or not providing various theoretical scheduling features.

Therefore a user friendly efficient user space scheduling library with small footprint and low overhead for FreeRTOS, ESFree was proposed.

ESFree provides simple interface for using aperiodic jobs, sporadic jobs, polling server, timing error detection and handling for missed deadlines, timing error detection and handling for exceeded WCET, RMS, DMS and two different implementations of EDF. Furthermore, phases and periods of periodic tasks are set by the user during task creation and handled by ESFree, so that implementation of phases and periods in task function code is abstracted away from the user. All these factors together make ESFree to a useful tool for teaching purpose and introducing students to FreeRTOS. There was also an observation that trace macro dependent scheduling policy implementations as efficient EDF on FreeRTOS do not always behave correctly, if the system contains a periodic task with longer deadline than its period.

Future work will be implementing additional scheduling policies and different periodic servers as sporadic server and total bandwidth server for ESFree, and extending ESFree to support multicore when FreeRTOS starts to support multicore.

Bibliography

[1] Real Time Engineers Ltd. Freertos, Accesssed: 2016-06-03. `http://www.freertos.org/`.

[2] G. Buttazzo. *Hard real-time computing systems : predictable scheduling algorithms and applications*. Springer New York, 2011.

[3] D.B. Stewart and P.K. Khosla. Mechanisms for detecting and handling timing errors. *Communications of the ACM*, 40(1):87–93, 1997.

[4] Avnet Inc. *ZedBoard (Zynq Evaluation and Development) Hardware User's Guide*, 2012.

[5] Digilent Inc. *Nexys4 DDR FPGA Board Reference Manual*, 2016.

[6] Xilinx Inc. *MicroBlaze Processor Reference Guide*, 2016.

[7] STMicroelectronics. Stm32f401, Accessed: 2016-07-29. `http://www.st.com/content/st_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32f4-series/stm32f401.html?querycriteria=productId=LN1810`.

[8] Francisco E Paez, Jose M Urriza, Ricardo Cayssials, and Javier D Orozco. Freertos user mode scheduler for mixed critical systems. In *Sixth Argentine Conference on Embedded Systems (CASE)*, pages 37–42. IEEE, 2015.

[9] M.A. Rivas and M.G. Harbour. Posix-compatible application-defined scheduling in marte os. In *Proceedings of 14th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 67–75, 2002.

[10] M.A. Rivas and M.G. Harbour. A new generalized approach to application-defined scheduling. In *Proceedings of 16th Euromicro Conference on Real-Time Systems (ECRTS)*, 2004.

[11] M. Asberg, T. Notle, S. Kato, and R. Rajkumar. Exsched: An external cpu scheduler framework for real-time systems. In *Proceedings of Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 240–249. IEEE, 2012.

[12] A. Diaz, I. Ripoll, A. Crespo, and Balbastre P. A new application-defined scheduling implementation in rtlinux. In *Proceedings of the Sixth Real-Time Linux Workshop*, 2004.

[13] R. Inam, J. Maki-Turja, M. Sjodin, S. M. H. Ashjaei, and S. Afshar. Support for hierarchical scheduling in freertos. In *Proceedings of the 16th Emerging Technologies & Factory Automation (ETFA)*, pages 1–10. IEEE, 2011.

[14] M. Aldea, G. Bernat, I. Broster, A. Burns, R. Dobrin, M. Drake, G. Fohler, P. Gai, M. Gonzalez Harbour, G. Guidi, J.J. Gutierrez, T. Lennvall, G. Lipari, J.M. Martinez, J.L. Medina, J.C. Palencia, and M. Trimarchi. Fsf: A real-time scheduling architecture framework. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 113–124. IEEE, 2006.

[15] A. Gomez, L. Schor, P. Kumar, and L. Thiele. Sf3p: A framework to explore and prototype hierarchical compositions of real-time schedulers. In *Proceedings of the 25th IEEE International Symposium on Rapid System Prototyping (RSP)*, pages 2–8. IEEE, 2014.

[16] G. Buttazzo and Gai P. Efficient edf implementation for small embedded systems. In *Proceedings of the 2nd Int. Workshop on Operating Systems Platforms for Embedded Real-Time applications OSPERT*, 2006.

[17] A. Carlini and G. Buttazzo. An efficient time representation for real-time embedded systems. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 705–712, 2003.

[18] Davis R.I. and Burns A. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS)*, pages 389–398. IEEE, 2005.

[19] G. Buttazzo. Rate monotonic vs. edf: Judgment day. In *Real-Time Systems*, pages 5–26, 2005.

Appendix A

ESFree on GitHub

The source code of ESFree is published on following link:
https://github.com/ESFree/ESFree

The repository contains five files:

- scheduler.c
- scheduler.h
- xil_main.c
- license.txt
- README.md

xil_main.c is an example demonstration of ESFree on Xilinx platforms.

Appendix B

FreeRTOSConfig.h

This chapter contains examples of FreeRTOS configurations on Digilent ZedBoard that is compatible with ESFree.

B.1 FreeRTOSConfig.h example for Digilent ZedBoard not using trace macros

The source code for an example configuration of FreeRTOSConfig.h on Digilent ZedBoard that is compatible with ESFree without using trace macros, hence not using efficient EDF is listed below:

```
#include "xparameters.h"

#define configUSE_PREEMPTION 1
#define configUSE_MUTEXES 0
#define configUSE_RECURSIVE_MUTEXES 0
#define configUSE_COUNTING_SEMAPHORES 1
#define configUSE_TIMERS 1
#define configUSE_IDLE_HOOK 0
#define configUSE_TICK_HOOK 1
#define configUSE_MALLOC_FAILED_HOOK 1
#define configUSE_TRACE_FACILITY 0
#define configUSE_16_BIT_TICKS 0
#define configUSE_APPLICATION_TASK_TAG 0
#define configUSE_CO_ROUTINES 0
#define configTICK_RATE_HZ (1000)
#define configMAX_PRIORITIES (32)
#define configMAX_CO_ROUTINE_PRIORITIES 2
#define configMINIMAL_STACK_SIZE ( ( unsigned short ) 200)
#define configTOTAL_HEAP_SIZE ( ( size_t ) ( 65536 ) )
#define configMAX_TASK_NAME_LEN 10
#define configIDLE_SHOULD_YIELD 1
#define configTIMER_TASK_PRIORITY (configMAX_PRIORITIES - 1)
#define configTIMER_QUEUE_LENGTH 10
#define configTIMER_TASK_STACK_DEPTH ((configMINIMAL_STACK_SIZE) * 2)
#define configUSE_QUEUE_SETS 1
```

```
#define configCHECK_FOR_STACK_OVERFLOW 0
#define configQUEUE_REGISTRY_SIZE 10
#define configUSE_STATS_FORMATTING_FUNCTIONS 0
#define configNUM_THREAD_LOCAL_STORAGE_POINTERS 1
#define configUSE_TICKLESS_IDLE 0
#define configTASK_RETURN_ADDRESS     NULL
#define INCLUDE_vTaskPrioritySet                1
#define INCLUDE_uxTaskPriorityGet               1
#define INCLUDE_vTaskDelete                     1
#define INCLUDE_vTaskCleanUpResources           1
#define INCLUDE_vTaskSuspend                    1
#define INCLUDE_vTaskDelayUntil                 1
#define INCLUDE_vTaskDelay                      1
#define INCLUDE_eTaskGetState                   1
#define INCLUDE_xTimerPendFunctionCall          1
#define INCLUDE_pcTaskGetTaskName               1
#define INCLUDE_xTaskGetCurrentTaskHandle       1
#define INCLUDE_xTaskGetIdleTaskHandle          1
#define configMAX_API_CALL_INTERRUPT_PRIORITY (18)
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 1

#define configINTERRUPT_CONTROLLER_BASE_ADDRESS ( XPAR_PS7_SCUGIC_0_DIST_BASEADDR )
#define configINTERRUPT_CONTROLLER_CPU_INTERFACE_OFFSET ( -0xf00 )
#define configUNIQUE_INTERRUPT_PRIORITIES    32
void vApplicationAssert( const char *pcFile, uint32_t ulLine );
void FreeRTOS_SetupTickInterrupt( void );
#define configSETUP_TICK_INTERRUPT() FreeRTOS_SetupTickInterrupt()
void FreeRTOS_ClearTickInterrupt( void );
#define configCLEAR_TICK_INTERRUPT()  FreeRTOS_ClearTickInterrupt()
```

B.2 FreeRTOSConfig.h example for Digilent ZedBoard using trace macros

The source code for an example configuration of FreeRTOSConfig.h on Digilent ZedBoard that is compatible with ESFree using efficient EDF is listed below:

```
#ifndef _FREERTOSCONFIG_H
#define _FREERTOSCONFIG_H

#include "xparameters.h"

#define configUSE_PREEMPTION 1
#define configUSE_MUTEXES 0
#define configUSE_RECURSIVE_MUTEXES 0
#define configUSE_COUNTING_SEMAPHORES 1
#define configUSE_TIMERS 1
#define configUSE_IDLE_HOOK 0
#define configUSE_TICK_HOOK 1
#define configUSE_MALLOC_FAILED_HOOK 1
#define configUSE_TRACE_FACILITY 1
#define configUSE_16_BIT_TICKS 0
#define configUSE_APPLICATION_TASK_TAG 0
```

B.2. FREERTOSCONFIG.H EXAMPLE FOR DIGILENT ZEDBOARD USING TRACE MACROS

```
#define configUSE_CO_ROUTINES 0
#define configTICK_RATE_HZ (1000)
#define configMAX_PRIORITIES (32)
#define configMAX_CO_ROUTINE_PRIORITIES 2
#define configMINIMAL_STACK_SIZE ( ( unsigned short ) 200)
#define configTOTAL_HEAP_SIZE ( ( size_t ) ( 65536 ) )
#define configMAX_TASK_NAME_LEN 10
#define configIDLE_SHOULD_YIELD 1
#define configTIMER_TASK_PRIORITY (configMAX_PRIORITIES - 1)
#define configTIMER_QUEUE_LENGTH 10
#define configTIMER_TASK_STACK_DEPTH ((configMINIMAL_STACK_SIZE) * 2)
#define configUSE_QUEUE_SETS 1
#define configCHECK_FOR_STACK_OVERFLOW 0
#define configQUEUE_REGISTRY_SIZE 10
#define configUSE_STATS_FORMATTING_FUNCTIONS 0
#define configNUM_THREAD_LOCAL_STORAGE_POINTERS 1
#define configUSE_TICKLESS_IDLE 0
#define configTASK_RETURN_ADDRESS     NULL
#define INCLUDE_vTaskPrioritySet              1
#define INCLUDE_uxTaskPriorityGet             1
#define INCLUDE_vTaskDelete                   1
#define INCLUDE_vTaskCleanUpResources         1
#define INCLUDE_vTaskSuspend                  1
#define INCLUDE_vTaskDelayUntil               1
#define INCLUDE_vTaskDelay                    1
#define INCLUDE_eTaskGetState                 1
#define INCLUDE_xTimerPendFunctionCall        1
#define INCLUDE_pcTaskGetTaskName             1
#define INCLUDE_xTaskGetCurrentTaskHandle     1
#define INCLUDE_xTaskGetIdleTaskHandle        1
#define configMAX_API_CALL_INTERRUPT_PRIORITY (18)
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 1

#define configINTERRUPT_CONTROLLER_BASE_ADDRESS ( XPAR_PS7_SCUGIC_0_DIST_BASEADDR )
#define configINTERRUPT_CONTROLLER_CPU_INTERFACE_OFFSET ( -0xf00 )
#define configUNIQUE_INTERRUPT_PRIORITIES     32
void vApplicationAssert( const char *pcFile, uint32_t ulLine );
void FreeRTOS_SetupTickInterrupt( void );
#define configSETUP_TICK_INTERRUPT() FreeRTOS_SetupTickInterrupt()
void FreeRTOS_ClearTickInterrupt( void );
#define configCLEAR_TICK_INTERRUPT()  FreeRTOS_ClearTickInterrupt()

#define INCLUDE_xTimerGetTimerDaemonTaskHandle     1
#define INCLUDE_xTaskGetSchedulerState     1
#define traceBLOCKING_ON_QUEUE_RECEIVE( xQueue ) vSchedulerBlockTrace();
#define traceBLOCKING_ON_QUEUE_SEND( xQueue ) vSchedulerBlockTrace();
#define traceTASK_DELAY_UNTIL() vSchedulerBlockTrace();
#define traceTASK_SUSPEND( xTask ) vSchedulerSuspendTrace( xTask );
#define traceMOVED_TASK_TO_READY_STATE( xTask ) vSchedulerReadyTrace( xTask );
```

TRITA TRITA-ICT-EX-2016:166