UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

Monitoring Architecture for Real Time Systems

André Lousa Marques

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática
(2º ciclo de estudos)

Orientador: Prof. Doutor Paul Andrew Crocker

Covilhã, Outubro de 2016

Acknowledgments

I would like to express my deep gratitude to Dr. Paul Crocker, my supervisor, for his incentive and support whenever requested throughout these last years.

To my parents and closest family, I thank you for your support and encouragement throughout my studies.

Finally, I would also want to thank Hélder Silva, Manuel Coutinho and José Valdez from Edisoft, for the great work environment.

Resumo

Nem sempre é fácil perceber como é que um sistema operativo - e *software* em geral - chegaram a determinado resultado apenas olhando para este. A abordagem normal é usar registos, ou pequenas impressões em locais estratégicos do código, no entanto esta abordagem não é escalável de forma consistente e sustentada.

O propósito desta tese é o de propor e desenvolver uma ferramenta - uma ferramenta de monitorização - capaz de capturar e registar a execução de uma dada aplicação com o mínimo de impacto no contexto de sistemas embebidos de tempo-real, nomeadamente usando uma versão do sistema operativo de tempo-real Real-Time Executive for Multiprocessor Systems (RTEMS) qualificada para o espaço, e colocando essa informação à disposição para processamento e análise futura. Ambientes com múltiplos núcleos de processamento são também considerados.

O atual estado da arte em monitorização e registo de execução de *software* é apresentado, destacando tanto exemplos da literatura como ferramentas e *frameworks* existentes. Usando uma implementação da arquitetura proposta, a ferramenta foi testada em configurações com um ou mais núcleos de processamento em arquiteturas *sparc* e *arm*, tendo sido capaz de registar e gravar dados da execução de uma aplicação de exemplo, como vários níveis de detalhe.

Palavras-chave

Tracing, Monitoring, Sistema Operativo Tempo-Real, RTEMS, Sistema Embebido, Multicore.

Resumo alargado

Enquadramento

O propósito desta tese é o de propor e desenvolver uma ferramenta - uma ferramenta de monitorização - capaz de capturar e registar a execução de uma dada aplicação com o mínimo de impacto no contexto de sistemas embebidos de tempo-real, nomeadamente usando uma versão do sistema operativo de tempo-real RTEMS qualificada para o espaço, e colocando essa informação à disposição para processamento e análise futura. Monitorização em ambientes com múltiplos núcleos de processamento são também considerados.

O RTEMS é um sistema operativo de tempo-real livre de código aberto, usado maioritariamente em sistemas embebidos. É usado para várias áreas e aplicações onde o tempo e os prazos de execução são importantes, tais como as indústrias espaciais, defesa, medicina, aviação, aplicações cientificas, entre outras. É o principal sistema operativo livre usado pela European Space Agency (ESA) e pela National Aeronautics and Space Administration (NASA) para as suas missões espaciais.

Esta tese foi desenvolvida na Edisoft, S.A., que desenvolve e mantém uma versão do RTEMS - o *RTEMS Improvement* - que é usada pela ESA nos seus satélites.

Estado da Arte

O estado da arte sobre monitorização e histórico/registo de execução é apresentado no capitulo 2. Várias arquiteturas e abordagens são propostas. De forma resumida: Iyenghar *et al* [IWWP13] propõem usar rotinas de monitorização com impacto mensurável, permitindo ter em conta o impacto da monitorização nas análises de escalonabilidade; Plattner *et al* [PN81] destaca as limitações crescentes dos monitores em *hardware*; ferramentas como o *ThreadX* [Loga] envia os dados de execução na *idle thread* do sistema, por forma a minimizar o impacto neste.

Uma análise às ferramentas e soluções de monitorização é apresentada na secção 2.7. Apesar da maioria usar monitores em *software*, ainda existem monitores em *hardware* a serem desenvolvidos (e.g.: pela *Green Hills* [Hila] e *Segger* [SEGc]). Um detalhe importante é a visualização destes dados: 20 dos maiores sistemas operativos usados atualmente na indústria usa a ferramenta *Tracealizer* [Per] para visualização e análise dos dados recolhidos de uma execução. A ferramenta *tracecompass* [Pol] é a alternativa livre *open-source*, usada por sistemas como o *Linux* e o RTEMS. Nesta medida, as ferramentas de monitorização atuais tendem a não incluir visualizadores e processados para os dados que capturam. Em vez disso, quando registam a execução de uma aplicação gravam os dados num formato estandardizado (detalhado na secção 2.4) (ou convertem posteriormente para esse formato), que estas ferramentas depois tratam de tornar a análise desses dados mais intuitiva e simples.

A literatura revela uma miscelânea de nomenclaturas usadas para descrever arquiteturas de monitorização, e vários pontos de vista sobre o propósito de uma ferramenta de monitorização. Esta foca-se mais na análise da execução de um sistema, e nem tanto em como essa execução é obtida. No entanto, garantir que esses dados são obtidos com o menor impacto possível na execução da aplicação é da maior importância para uma plataforma com recursos limitados como os usados na área de sistemas embebidos e de tempo-real.

Desenvolvimento

O capitulo 3 apresenta os requisitos da ferramenta desenvolvida, considerando um ambiente com apenas um núcleo de processamento (no contexto do projeto *RTEMS Qualification Extensions* para a ESA). O desenho e arquitetura da ferramenta são apresentados no capítulo 4.
Neste contexto a ferramenta foi utilizada com a plataforma GR712 da Gaisler, que possui dois núcleos *sparc leon 3* em que apenas um é utilizado (o sistema operativo não está preparado para sistemas com mais que um núcleo), a correr o *RTEMS Improvement* da Edisoft S.A..
A captura dos eventos é feita recorrendo a uma funcionalidade especifica do *linker* da GNU is Not Unix (GNU) para encapsulamento de funções denominada de *wrap*, onde uma função pode ser substituída por outra com a mesma assinatura desde que precedida por `__wrapper_`. Esta funcionalidade permite que chamadas ao sistema, por exemplo, sejam substituídas por funções intermédias (precedidas por `__wrapper_`) que além de chamarem as funções originais (a função original é acedida normalmente, precedida por `__real_`) registam informações sobre o contexto e conteúdo dessa mesma chamada.

Nesta versão da ferramenta todos os eventos são guardados num *buffer* global de eventos, que são posteriormente enviados para uma plataforma externa à qual o sistema está ligado por *SpaceWire*. A periodicidade com que os eventos são enviados pode ser:

- em direto - o utilizador pode configurar a ferramenta para enviar um determinado número de eventos (se os houver) a cada n microsegundos.
- quando a execução termina - imediatamente antes de terminar o sistema, todos os eventos guardados são enviados para a plataforma externa.

Em ambos os casos é possível à aplicação a ser monitorizada forçar o envio a qualquer momento de quantos eventos queira. Na plataforma externa os eventos são recebidos e guardados ou numa base de dados My Structured Query Language (MySQL) ou em ficheiros Comma-Separated Values (CSV). Estes dados podem depois ser visualizados e processados posteriormente.

O capitulo 6 faz uma breve apresentação sobre sistemas com múltiplos núcleos, a sua utilização na indústria espacial e como o RTEMS suporta este tipo de sistemas. É também apresentada uma versão estendida da ferramenta adaptada a este tipo de sistemas, cujas diferenças são:

- Um *buffer* por processador/núcleo de processamento - usar um único *buffer* de eventos para vários núcleos de processamento iria criar um constrangimento, já só um de cada vez poderia colocar os seus eventos no *buffer*. Como cada processador passa a ter o seu próprio *buffer*, este constrangimento desaparece, sem com isso aumentar os requisitos de memória (uma vez que a memória que seria usada no *buffer* global é dividida pelos vários processadores);
- Formato dos eventos - num sistema com vários núcleos de processamento, cada evento além de ter registado quando aconteceu passa também a precisar de registar onde aconteceu. Nesse sentido foi necessário criar um campo adicional para indicar o índice do processador onde cada evento ocorreu.

Esta versão entendida implicou alterar a versão do RTEMS utilizado para uma versão com suporte a múltiplos núcleo de processamento e também o canal de comunicação com a plataforma externa para usar uma porta série normal Recommend Standard 232 (RS-232), e correr num ambiente emulado em Quick Emulator (QEMU).

Conclusões e Trabalho Futuro

As conclusões e trabalho futuro são apresentados no capitulo 7. No final da tese foi produzido uma ferramenta de monitorização capaz de monitorizar a execução de uma aplicação num ambiente com um único núcleo de processamento, assim como uma extensão desta ferramenta capaz de fazer o mesmo mas com uma aplicação a executar em múltiplos núcleos ao mesmo tempo, recorrendo a uma versão do RTEMS com suporte a este tipo de sistema e a uma plataforma simulada. Testes iniciais com esta extensão da ferramenta mostram que também é viável para sistemas com múltiplos núcleos.

Algum trabalho futuro é apresentado, do qual se destaca a necessidade de tornar os dados recolhidos possíveis de ser visualizados numa ferramenta gráfica.

Abstract

It can be hard to understand how an operating system - and software in general - reached a certain output just by looking at said output. A simple approach is to use loggers, or simple print statements on some specific critical areas, however that is an approach that does not scale very well in a consistent and manageable way.

The purpose of this thesis is to propose and develop a tool - a Monitoring Tool - capable of capturing and recording the execution of a given application with minimal intrusion in the context of real-time embedded systems, namely using a space-qualified version of the RTEMS real-time operating system, and making that information available for further processing and analysis. Multicore environments are also considered.

The current state of the art in monitoring and execution tracing is presented, featuring both a literature review and a discussion of existing tools and frameworks. Using an implementation of the proposed architecture, the tool was tested in both unicore and multicore configurations in both sparc and arm architectures, and was able to record execution data of a sample application, with varying degrees of verbosity.

Keywords

Tracing, Monitoring, Real-Time Operating System, RTEMS, Embedded System, Multicore.

Contents

List of Figures

List of Tables

Listings

List of Acronyms

RTEMS Real-Time Executive for Multiprocessor Systems

RTOS Real Time Operating System

OS Operating System

ESA European Space Agency

SMP Symmetric MultiProcessing

CPU Central Processing Unit

SOC System on a Chip

ICE In-Circuit Emulator

RCP Rich Client Platform

ISR Interrupt Service Routine

IRQ Interrupt Request

UART Universal Asynchronous Receiver/Transmitter

JTAG Joint Test Action Group

ROM Read-Only Memory

RAM Random-Access Memory

RS-232 Recommend Standard 232

KB KiloByte

QT QT Framework

UML Unified Modeling Language

GUI Graphical User Interface

LTL Linear-time Temporal Logic

BSP Board Support Package

IPI Inter-Processor Interrupt

MrsP Multiprocessor Resource Sharing Protocol

SpW SpaceWire

AMBA Advanced Microcontroller Bus Architecture

PCI Peripheral Component Interconnect

I/O Input/Output

CTF Common Trace Format

TSDL Trace Stream Description Language

GB GigaByte

IDE Integrated Development Environment

TCP/IP Transmission Control Protocol/Internet Protocol

API Application Programming Interface

MB MegaByte

CSV Comma-Separated Values

EGSE Electrical Ground Support Equipment

RTE RunTime Environment

MySQL My Structured Query Language

SQL Structured Query Language

DBMS DataBase Management System

Hz Hertz

TCB Task Control Block

JRE Java Runtime Environment

URL Uniform Resource Locator

GRMON Gaisler Research Monitor

G-JLFP Global Job-Level Fixed Priority

G-FTP Global Fixed Task-Priority Pre-emptive

FIFO First In, First Out

ASCII American Standard Code for Information Interchange

MIL-STD-1553 Military Standard 1553

OAR On-Line Applications Research

NASA National Aeronautics and Space Administration

QEMU Quick Emulator

GNU GNU is Not Unix

CD Compact Disc

DSU Debug Support Unit

FPGA Field-Programmable Gate Array

Chapter 1

Introduction

Improving software methodologies and processes will never lead to bug free software, so there remains a need for better tools that can help developers find faults before they become failures.

In the real, macroscopic physical world, the act of observing a given event is a passive occurrence. We can not affect a physical outcome just by watching it happen. However, to observe a software system or application, just like observing quantum particles, implies an interaction with the system which can (and will) affect the resulting outcome.

1.1 Context

RTEMS is an open source full featured Real Time Operating System (RTOS) that supports a variety of open API and interface standards, targeted for embedded systems. The RTEMS project is managed by the On-Line Applications Research (OAR) corporation with the help of the RTEMS user community and serves as the base operating system for many applications with real-time needs, such as space, defense, medical, industry, aviation, and more. It is the main open source RTOS used by NASA and ESA for their space missions, and Portugal has the only RTEMS centre outside the United States of America, which is managed by Edisoft S.A.. The main RTEMS version used by ESA is Edisoft's RTEMS Improvement, a space-qualified version forked from RTEMS 4.8.

This thesis was developed at Edisoft, initially in the context of the *RTEMS Qualification Extensions* project proposed by ESA to Edisoft with the purpose to add support for Military Standard 1553 (MIL-STD-1553) [Agea] and SpaceWire [Ageb] buses and also the development of a Monitoring Tool for Edisoft's RTEMS Improvement. The focus of this thesis is the development of this Monitoring Tool, and then to extend it to work in a multicore configuration (not part of the scope of the initial project).

1.2 Motivation

I have been working with RTEMS since late 2013, and have been a participant of Google Summer of Code with the RTEMS project for two times as a student (2014 and 2015) and one time as a mentor (2016). This project allowed me to improve my knowledge in RTEMS by working in a professional setting with the company that developed and supports the RTEMS version used by all European space missions, such as ESA Galileo, and gather skills on a new field that I had never worked on: execution tracing.

1.3 Document Outline

In order to describe the work that was developed during the project, this document is structured as follows:

- Chapter 1 – **Introduction** – introduces the project and its motivations;
- Chapter 2 – **State of the Art** – presents an overview and discussion of the most relevant literature and existing monitoring tools and tracing architectures, frameworks and tools;
- Chapter 3 – **Monitoring Tool Requirements** – presents the software requirements for the developed monitoring tool;
- Chapter 4 – **Monitoring Tool Design** – presents the software design/architecture of the developed monitoring tool;
- Chapter 5 – **Monitoring Tool Testsuite** – shows how the monitoring tool was tested;
- Chapter 6 – **Multicore** – gives a brief overview of the SMP support in RTEMS and how the developed monitoring tool was extended to work in a multicore setting;
- Chapter 7 – **Conclusions and Future Work** – presents the project conclusions and some ideas for future work on this project.

Chapter 2

State of the Art

It can be difficult for a system's developer or user to see and understand what an operating system - and software in general - is doing just by looking at its outputs. Software can be analyzed either by static or dynamic methods, where the former focus on the static aspects of the software (i.e.: the source code) and the latter on its execution. Static methods include formal proofs of program correctness (or the analysis of the programming language itself) while dynamic methods include debuggers and tracing/monitoring tools.

The main difference between a debugger and a monitoring tool is that debuggers target programming languages, while monitoring tools target operating systems. For instance, while a mutex is just another program variable to a debugger, a monitoring tool must be able to recognize and provide relevant information about it. To have this sort of system specific information with a general purpose debugger (general purpose since it targets any program written in a certain language, being it a simple application or a complete Operating System (OS)) the user would have to have knowledge of where to get that information, and be able to access it.

This section details the current state of the art in the field of operating system execution monitoring. It starts with a definition for System Monitoring, going through the methods to capture and reason on what is going on in a live and running operating system, and what is being done to ensure that capturing this information does not disturb (or minimally disturb) the system and its expected outcomes.

2.1 System Monitoring

The term "Monitoring" may have slightly different meanings, depending on context. Monitoring, also called tracing, may refer either to the act of capturing the execution data from a given system, or the analysis of that data (either at run-time (online) or after the system shutdown (offline) or both). Trace data refers to a complete set of events resulting of one or more system executions, while an individual event refers to a particular change in the execution behaviour (e.g.: a specific context switch).

Although the term "monitoring" is used, monitoring a system does not always mean plain observation as monitors may also perform actions triggered by what is being observed. This type of monitoring occurs while the monitored system is running, and is called runtime-monitoring or runtime checking (refer to section 2.3.1).

A Monitoring Tool consists of retrieving event data from a system's execution for storage and/or analysis. Figure 2.1 portraits a high level view of most monitoring system architectures:

- The monitored system has a monitor component (software and/or hardware based) to capture the system's execution events and send those events to be processed;

- A processing component (usually running on a different, separate platform to reduce the impact on the monitored system execution), which receives and stores the events and may allow some further processing to happen on those events.

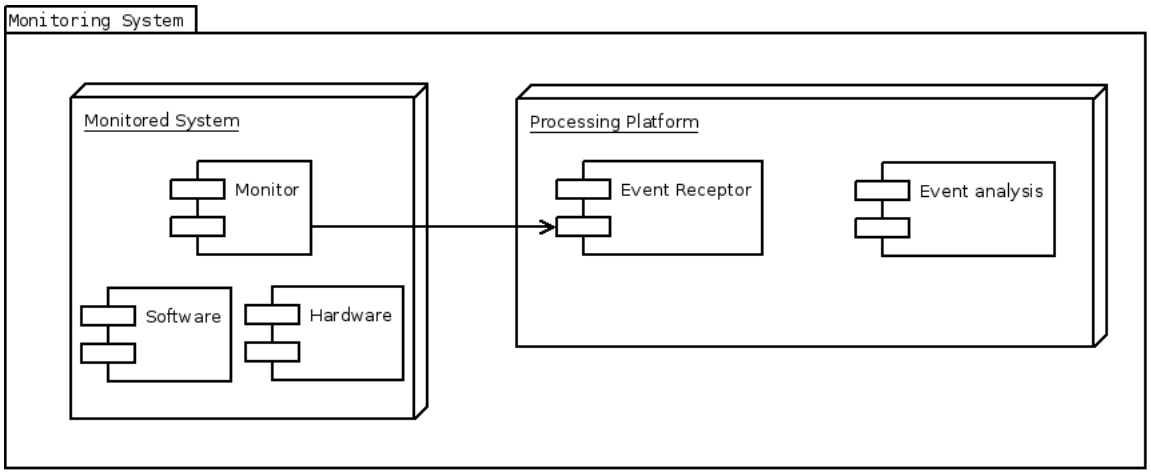

Figure 2.1: Monitoring System.

Event processing ranges from simply visualizing the events on a Graphical User Interface (GUI) screen to verifying if the system requirements were not violated. The capture component executes concurrently with the monitored system, while the processing part may happen long after the monitored system terminated its execution (as long as the events were recorded, which must happen while the system executes).

There are a number of techniques available to monitor a system (which will be detailed later in this chapter), but the main drawback of doing so is the impact on the monitored system's normal execution. This is even more critical when the target system consists of a deeply embedded system with limited computational and memory resources while at the same time having to perform under strict real-time execution constraints.

A Monitoring tool is thus useful to detect race conditions, deadlocks, and similar (real-time) problems that are hard to detect with other debugging tools [HR02] [Sho04]. It can also identify when an interrupt or context switch happened and pinpoint the timing of those events in the context of the overall system's operation, making it easier to detect unexpected behaviour.

The trace data collected can also be used for:

- Task scheduling analysis;
- Resource dimensioning;
- Performance analysis;
- Algorithm optimization.

The following subsections will describe the requirements and constraints that any monitoring tool must take into account.

Instrumentation Points

Monitoring implies adding logic to the system so its execution can be recorded for further (online or offline) processing. To do this monitoring points are chosen in the system, such that when the system's execution reaches those points the monitoring code is executed and that event is recorded. Monitoring points bind and direct the monitoring system towards specific points of interest, allowing for a better understanding of the system's execution flow. A monitoring point might be the system's context switch function, so every context switch in the system is logged and appropriately labelled, whilst identifying which tasks are being scheduled in and out of which queue in the CPU. The process of adding this recording logic to a system's source or object code is called instrumentation, and the places where it is invoked are referred to as monitoring or instrumentation points. Placing and choosing the location of these monitoring points can be done manually by the system's developer or user or dynamically through an automatic tool. Automatic tools may use previous execution traces to decide where to instrument the system, or evaluate what to instrument while the system is running [DGR04].

Placement

The monitoring logic which is invoked when an instrumentation point is hit may execute on the same system that is being monitored, or on a separate machine. The location where this logic executes is called placement. It can be classified as online, if the monitoring logic executes on the same machine, or offline if it executes on a separate machine.

Platform

The monitor part of the monitoring tool can be based on software, hardware, or a mix of the two. A software monitor requires the instrumentation of the system's code, while a hardware based monitor will (possibly) connect to the system and listen to the system's bus for clues on what is happening on the system. Monitoring platforms are further explained in section 2.2.

2.2 Capturing System Execution Trace Data

Monitoring a system or application execution is prone to disturb the said execution, as learning about its inner operations requires the installation of probes (ranging from actual physical probes to software probes in the form of instrumented code) on the monitored system. Any probe used for monitoring must meet two basic requirements:

- domain - the probe shall be able to capture information on the type of events that are expected to be monitored;
- performance - adding the probe to the system shall not disturb (over an acceptable overhead) the intended behaviour of the monitored system.

To capture the execution of a system it has to be observed. Monitoring tools can be classified [IWWP13] into the following categories, representing different approaches to monitoring:

- Hardware based (may also be referred to as on-chip monitoring);
- Software based;
- Hybrid - a mixed approach.

These approaches are detailed below.

2.2.1 Hardware Based

Hardware based monitoring makes use of additional hardware attached to the system to capture and retrieve execution data. It was clear since the beginning of system monitoring (in the fifties and sixties) that in order not to disturb the timing of the monitored process, separate hardware would be required [PN81], as hardware monitors reduce (or remove) the impact of monitoring on a system's performance [DGR04].
Old fashioned hardware based monitoring [IWWP13] [WH07] uses physical probes connected to the system's hardware to get a glimpse of its execution. Early hardware based monitors made use of oscilloscope probes, logic analyzers and In-Circuit Emulator (ICE)s. In-Circuit Emulator are hardware devices that take the place of the CPU, effectively replacing it. It has the advantage that since it will execute the instructions the real chip would execute, it can monitor all states of the process. These have lost popularity due to the increasing complexity of processors.

Nowadays instead of probing the chip pins hardware monitors probe the system bus. A System on a Chip (SOC) (a common architecture for embedded systems) includes in the same chip all the core components that compose a computer, such as the processor(s), memory and graphics. All these components are connected together through a bus (e.g.: Advanced Microcontroller Bus Architecture (AMBA), Peripheral Component Interconnect (PCI)), so to trace the execution of a system running on top of a SOC an on-chip monitor listens and records the traffic flowing on the bus.
While SOC hardware is increasingly difficult (if not impossible) to monitor with the classical hardware monitors, they are usually embedded with a hardware trace unit capable of listening to the bus traffic and storing it on a dedicated hardware trace buffer with minimal impact on the performance. Access to the trace data can be done via fast debugging interfaces such as Joint Test Action Group (JTAG) from a separate machine.
The problem with bus tracing is that it is only capable of capturing instructions going from memory to the CPU, or between the CPU and some peripheral device. Instructions stored on the CPU cache, for instance, are invisible to the on-chip monitor.

2.2.2 Software Based

Software based monitoring relies in the instrumentation of source code in order for the program to output information about its internal state [WH07]. Instrumentation points can be placed as comments or annotations in the target's software that can then be replaced automatically by monitoring code (this may also be called inline monitoring [HR02]).
Through instrumentation it is possible to monitor every aspect of a software system, but at a price: having a software monitor executing concurrently in the same machine will delay its normal execution. This means that the system being monitored is not the same system being deployed and used in the real-world, and that can easily render the monitoring tool useless.

As stated in section 2.1, one of the purposes of a monitoring tool is to detect complex system behaviour such as race conditions. These can easily be masked with the introduction of delays on the system, meaning that an instrumented system may not reveal the problems it is supposed to find. It has been suggested in the literature [WH07] that the software probes (i.e.: the instrumented code) is left in the system to avoid this, but this approach is frowned upon in an embedded environment where resources are scarce.

An important concept for software monitors is time and space partitioning [NPP15], as it fosters independence between tasks. This allows monitors and system tasks to coexist in the same system while keeping the system tasks timing properties (e.g.: their worst-case execution times) as well as their private data to remain private. Each partition is isolated regarding to space (i.e.: memory accesses from other partitions) and timing (i.e.: tasks executing on a partition do not have an impact on the timing characteristics of tasks running on another partition) operations, so it is clear that software monitors should be isolated from the system via space and time partitioning.

2.2.3 Hybrid

Hybrid based monitoring refers to the simultaneous use of hardware and software monitors, complementing their strengths and weaknesses [WH07]. Hardware monitors have low to zero impact on the monitored system, however they are restricted to external I/O interfaces or instructions passing through the system's bus. Software monitors can monitor every aspect of the system, but with an additional overhead. Using the two approaches at the same time permits to off-load the software monitor from monitoring the events that the hardware monitor is able to monitor without affecting the system, while the software monitor allows the capture of events that are unreachable to the hardware monitor.

2.3 Analysis and Processing Trace Data

As previously noted, a system's execution can be analyzed, viewed and/or processed while the system is still executing (runtime monitoring), or after the system has terminated its execution (offline monitoring). These two approaches are discussed below. At this stage the captured events may be dispatched to a separate verification process on the target (monitored) system, or sent via a communication interface to be processed on a separate machine.
Note that the approaches shown in this section only refer to the visualization/processing of trace data, as offline monitoring for instance (as explained below in section 2.3.2) may still involve a runtime component for event transmission to a remote machine.

2.3.1 Runtime Monitoring

For a while now formal methods have been used in the analysis and validation of real-time systems, more exactly to models of those systems [HR02]. They provide mathematical proof of a system's correctness, but those verifications do not apply to the system's actual implementation, as they are often much more detailed than their models and usually do not strictly follow it. Software systems are also increasing their complexity making it unfeasible to get a complete formal verification of a running system (the exception are microkernels such as seL4, although they make some assumptions such as the compiler, which is assumed to be correct [KEH+09]).

Runtime monitoring analyses a given system during its execution, allowing the monitoring system or another entity (e.g.: a developer or user) to have a better view of what the system is doing at a given time and possibly intervene and correct unintended behaviour. System requirements compliance can be checked on the fly and measures can be taken to restore the system back to a safe state, reducing deployment times as the number of tests the system otherwise would have to go through can be shortened.

Even with a full campaign of tests planned and run on a system, there is always the possibility of something going wrong. Budget restrictions can impose limits on the number and scope of the tests to the more likely causes of error, or to test only the critical part of the system. In that sense, a runtime monitoring tool working as a software-fault safety net can make sure that the intended system properties are preserved even in a catastrophic event. If the system operates in a hard to reach location where it is difficult or impossible for someone to go and recover the system (e.g.: deep sea or in the outer space) it can avoid system loss.

The basis of a runtime monitoring tool is the software requirements and properties of the monitored system [DGR04] [WH07], as they provide insight on the expected outcomes and behaviours of the system. The monitor uses these properties to try and discover faults in the system before they become failures (by running concurrently with the system, analyzing every computational step). To do this a specification language (e.g.: Linear-time Temporal Logic) is used to formally describe the system requirements and properties in a format that can be processed by the monitor, namely by defining the system's properties as sequences of computational states.
The main purpose of a runtime monitor is to detect faults in a system's execution trace. When a fault is detected the monitor can provide further information on its cause either to the system or user (fault diagnosis), or even assist the recovery of the system to a safe state by:

- directing the system to a correct state (forward recovery);
- reverting to a previously correct state (backward recovery).

Actions might include logging the event and warning the user, calling recovery methods or simply stopping or rebooting the system. To capture the events most runtime verification tools require software instrumentation of the target system [WH07].

2.3.2 Offline Monitoring

One way to reduce the monitoring impact on performance is to reduce the number of actions. Offline monitors capture the event data while the system is executing, but only process or transmit that data when the system terminates its execution [WH07]. This approach is only suitable for debugging and post-mortem analysis, as comparatively with runtime monitoring the monitor never interacts with the system directly, but rather it is the system which writes the events on the monitor event buffer (which is also the case with runtime monitoring).

Depending on the system, the events may be analyzed/viewed in the same machine or sent to another machine. Depending on the number of events being traced and memory available, this method may not be feasible for all applications, as all events will have to be stored until the system terminates. If the buffer fills up events will be lost either by overwriting old events or by dropping new ones. One way to deal with this is to transmit events periodically with a monitor (i.e.: a thread or process, usually with the lowest priority on the system) running alongside the monitored system as an approach halfway between runtime and offline monitoring [NPP15].

2.4 Trace Data Format

As with any data transferred between two systems, the format or structure used has to be known to both sides. The monitored system has to produce trace data in a way such that a human or

machine can parse that data and understand the message.

Iyenghar *et al* [IWWP13] developed and used a simple protocol-like trace format (figure 2.2) where each packet has a header and a payload. The rationale behind this design is that it promotes compactness (one event, one packet) and extensibility (more data can be easily added to an event) while reducing the number of operations on the monitored system.

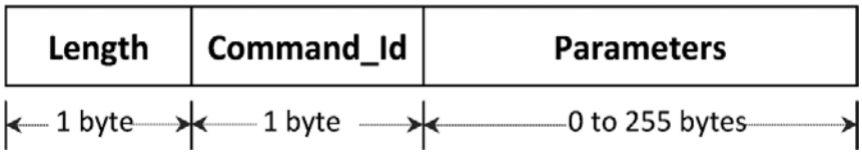

Figure 2.2: Trace format defined by Iyengar *et al* [IWWP13].

Apart from the trace formats defined and used by single projects, other formats are shared by different systems and applications. By standardizing the trace formats it is possible to develop system-independent trace processing tools, making it easier to develop monitoring tools as the system only has to provide the events/trace data.

The current standard on trace formats is the CTF, which organizes the trace data as multiple streams of binary events [Eff]. In this context a stream holds an ordered sequence of events, out of a specified group. It is up to the system being monitored to decide how many streams it will use and which events go on which stream, but at least one has to be provided: the metadata stream. The metadata stream is a file named "metadata" which contains a description of the trace data layout. This file is written in Trace Stream Description Language (TSDL), a C-like language to describe (non exhaustive):

- streams - such as the header and associated events;
- events - stream header, event name, id within the stream, data fields;
- system clock configuration - frequency.

2.5 Measuring the Impact on Performance

To measure performance the following metrics can be used [Hil05]:

- Performance profiling - how much time the system is spending on each function;
- Task profiling - how much time the system is spending on each task;
- A-B timing - how much time the system takes between two points in the system code;
- Response to external events - the time required for the system to respond to an external event (e.g.: interrupt);
- Task deadline performance - measures the time each task takes to reach its deadline (considering a multi-tasking application).

As with any other measurement, it is important to ensure that the measuring activity does not have affect the measured value (or at least the impact is known and can be accounted for). For software monitors these metrics can be retrieved by instrumenting specific portions of the monitor

to measure how much time the monitor takes in each step (e.g.: identify an event to be of a certain type or store an event). While the process of getting these measurements would increase the total overhead of the monitoring tool, they are likely to only have to be taken once, so this instrumented code can be left out or even removed after the measurements are taken. It is also possible that these can not be retrieved all in one go, as the code blocks being measured must not contain other code blocks being measured as well, since those measurements would include an overhead that will not exist on the final monitoring system.

As an example: if a certain function call takes around 100μs, and the monitor takes 2μ to identify that operation as a certain type of event and storing it on an internal buffer, then we have the monitoring tool overhead per each event of that type.
Other approaches may involve external hardware: Iyenghar *et al* [IWWP13] used a logic analyzer to measure the performance of their monitoring tool.

2.6 Literature Review

This section compiles the ideas and points of view of several authors on monitoring techniques. These are discussed in the section 2.8.

2.6.1 Architectures Used or Proposed

Iyenghar *et al* [IWWP13] proposed a generic software-based monitor that is independent of the underlying RTOS. Their architecture is simple:

1. The RTOS communicates with the monitor via its own framework;
2. The trace data is then sent to the host platform via a debug interface, such as JTAG or RS-232.

To reduce the communication overhead they have devised a frame, protocol-like format (see figure 2.2) for the trace data with the following characteristics:

- Compactness - minimize the data transfer size;
- Minimum number of operations on the target - minimize the impact on the target's execution;
- Extensibility - keeping the format simple enough so that more events can be added for transmission in the future.

The underlying RTOS invokes the monitor functions whenever an event is consumed. The event is then stored in a memory buffer and later sent to the host platform during the idle cycles of the CPU by the monitor. The buffer is configurable at compilation time, namely:

- buffer size;
- behaviour on buffer overflow. The buffer can discard events if full, and when it does the host is notified that events were lost.

They measured the time taken to store a 23 byte event on the buffer and sending it to the host via a RS-232 interface at 50μs. The memory requirements of their prototype are at 1061 bytes in Read-Only Memory (ROM) and 135 bytes in Random-Access Memory (RAM) (total memory size of $\approx$ 1 KiloByte (KB)).

On the host side they have created a QT Framework (QT) GUI application where the incoming trace data is decoded (from the protocol-like format) and shown as Unified Modeling Language (UML) sequence and timing diagrams.

Watterson *et al* [WH07] state that a modular approach is more suitable for a runtime monitoring tool for use on an embedded environment, where events are captured on the monitored system and sent to a verification process running on a separate machine, reducing execution overhead.

Nelissen *et al* [NPP15] recognize that space and time partitioning is important to ensure that a runtime monitor does not affect the monitored system, either in its behaviour or by propagating faults. They propose a monitoring architecture where each event type (rather than each monitor) has their own buffer to store the events. Each buffer works on a single producer, multiple consumer fashion (each reader has a pointer to the oldest event that they have not read yet), and since the events read are never removed (the buffers are circular, so new events overwrite older ones) the access to the buffer does not need any further synchronization and a single event can be of use to multiple readers. Readers feeding the same monitor synchronize the monitor event timeline via a synchronization variable storing the timestamp of the last event read, so the readers can determine which event should be read and sent to the runtime monitor first.

2.6.2 Uses for the Trace Data

Monitors output trace data usually for plain visualization of the execution flow. However, there are other more ambitious uses for this data.

Iyenghar *et al* [IWWP13] propose a framework where trace data is used to create a model (e.g.: UML interaction diagrams) of the system, which can then be used to generate test cases (model-based test cases) in real-time. The authors consider that these tests can help in regression testing and test coverage, while at the same time can act as an initial template for the tester to create more tests manually.

2.7 Existing Tools

This section introduces some of the most relevant monitoring tools being used in the embedded systems industry, and how the different operating systems (RTOS) capture execution data and make that data available to a monitoring tool. As detailed previously in section 2.2, a popular approach to capture system events is to instrument the system's software with code to capture execution data. The software instrumentation techniques vary from system to system, as different systems have different purposes and objectives.

2.7.1 Trace Compass

Trace Compass [Pol] is an open-source trace visualization tool developed by PolarSys, an Eclipse Industry Working group. It is based on the Eclipse framework, and available as a plug-in or as a Rich Client Platform (RCP) application.

The interface (see figure 2.3) consists of two timelines: one for processes and another for resources (such as CPU cores, or Interrupt Request (IRQ)s). As for statistics it only shows the percentage

of each event in the overall system event count. As for each event it shows:

- Timestamp;
- CPU core where it executed;
- Event type (e.g.: function call or return);
- Event contents (i.e.: call arguments or return value).

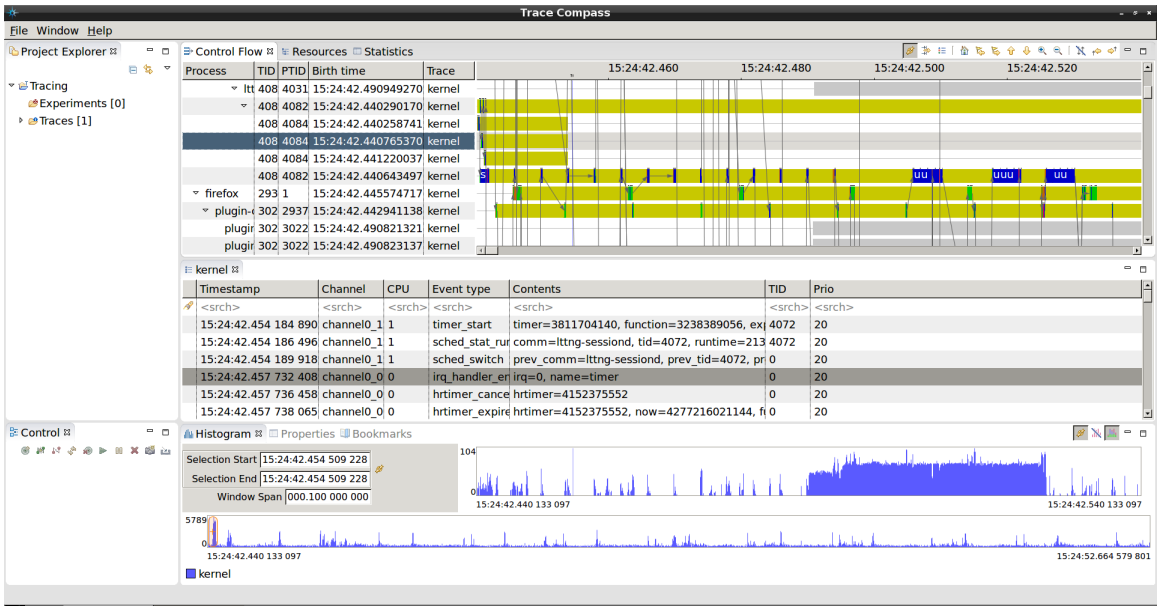

Figure 2.3: Printscreen of Trace compass showing a sample LTTng CTF trace.

2.7.2 Tracealizer

The main feature of Tracealizer's [Per] interface is the event timeline. It is possible to navigate through the various instances of each event (next and previous), as well as zoom (adjusting the timing resolution) and filter events. Event categories are called actors, of which each events is an instance of.
The interface (figure 2.4) provides information of each actor, such as:

- Instance start and finish clock ticks;
- Clock tick of the next execution;
- Who triggered its execution, and what else was triggered;
- Execution time (minimum, average and highest);
- CPU usage;
- What events this instance triggers.

There are also 20 other specialized views for blocking times, scheduling intensity, and others.

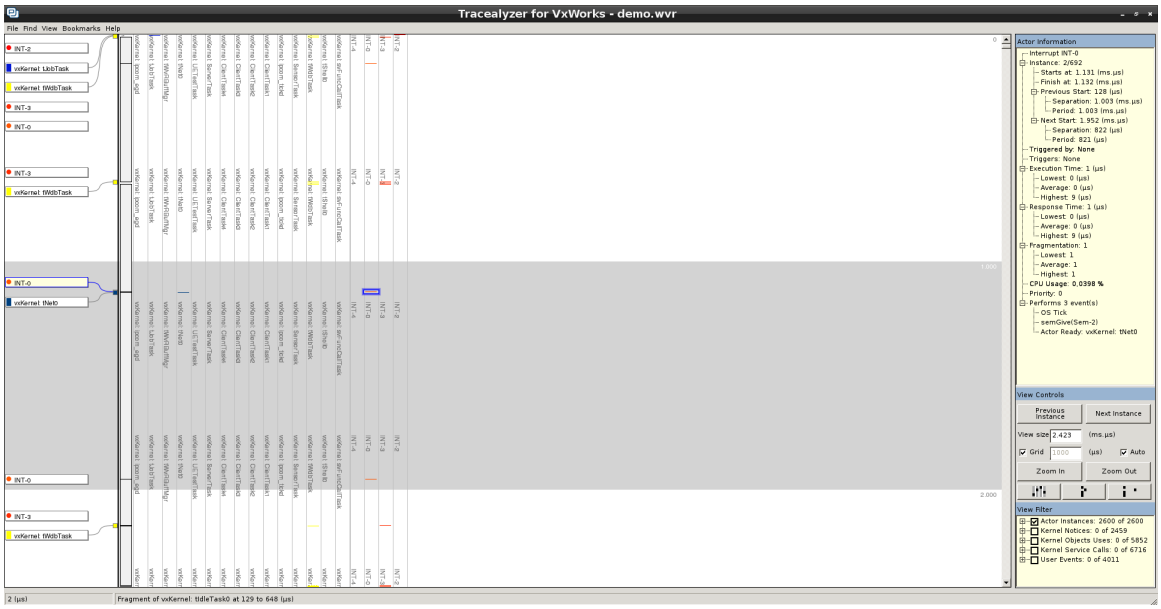

Figure 2.4: Printscreen of Tracealizer for VxWorks interface.

2.7.3 National Instruments

National Instruments [Ins] provides their users with access to system resource usage information. Their system gives programmatic access to a target system's available resources, allowing their user's software to change its behaviour if a given resource reaches a certain threshold. This also allows the creation and use of custom monitoring and logging tools, feeding on the CPU and memory usage information given by the system.
To aid the developers in the low-level analysis of the system, namely identifying memory allocation, analyzing thread activity as well as execution and timing data per CPU in the system, National Instruments provides a graphical trace viewer tool, the *Real-Time Trace Viewer*. This tool displays the percentage of CPU time spent executing threads by priority level, the percentage of idle CPU time and the percentage of CPU time devoted to Interrupt Service Routine (ISR)s.

2.7.4 ThreadX

Express Logic ThreadX RTOS [Loga] performance can be analyzed with intrusive or non-intrusive methods. The system provides two important variables:

- `_tx_thread_current_ptr` - pointer to the address of the currently running thread. If null the system is idle;
- `_tx_thread_system_state` - if non-zero an interrupt is being processed.

By monitoring only these two variables it is possible to monitor thread execution, idle time and interrupt processing in the whole system using the following rules:

1. thread execution starts when `_tx_thread_current_ptr` becomes non-NULL;
2. thread execution stops when `_tx_thread_current_ptr` becomes NULL;
3. if during a thread's execution `_tx_thread_system_state` becomes non-zero, the time during which is has a non-zero value is associated with interrupt processing;

4. when no thread is executing the system is in idle.

These variables can be monitored with a logic analyzer, meaning that this information can be retrieved without interfering with the system's execution (non-intrusive approach). A single thread can be monitored if its address is known. Each thread also has a counter which is incremented every time it is scheduled to execute, making it possible to detect situations of excessive preemption.
If a logic analyzer is not available it is also possible to instrument ThreadX scheduler, function return and interrupt management functions to capture the same data. Instrumentation implies the placement of a few assembly instructions, and the use of a dedicated high resolution timer. The estimated overhead impact is in the order of 5% per context switch or interrupt.
ThreadX also has built-in system tracing capabilities, activated by enabling a single flag (`EVENT_TRACE`). If tracing is enabled ThreadX will store execution data in a circular buffer. The application can control the size, location and contents of the traces, and may also insert custom events through an API. To analyze and view this information Express Logic provides the TraceX tool [Logb].
Trace data is sent to an host machine after execution (off-line monitoring) or during execution when a breakpoint is hit. The circular buffer stores up to a certain number of recent events, making them available for inspection in the event of a system malfunction or an user defined breakpoint. The following events can be traced in both single and multi-core systems:

- context switches;
- preemptions;
- system interrupts;
- application-specific (custom) events.

The events are labelled with an application time-stamp and identified with the active thread, so they can be displayed in the correct time sequence of events and properly identified. Tracing can be enabled or disabled dynamically to avoid filling the memory with events when the system is working correctly. TraceX also signals when a deterministic or non-deterministic priority inversion occurs.
For performance analysis, it also makes available:

- CPU usage;
- profiling information (time spent in each task, idle or processing interrupts;
- statistics (number of context switches, time slices, preemptions, interrupts, ...);
- thread stack usage.

TraceX is available for Windows, with a perpetual license for three developers costing $5,000.

2.7.5 Segger embOSView

Segger provides along with their RTOS embOS [SEGa] a profiling tool called embOSView [SEGb]. This tool communicates with the kernel via communication (Universal Asynchronous Receiver/-Transmitter (UART) or JTAG) ISRs installed in the system to collect timing information on the system's tasks, or CPU load, being the only instrumented code in the system. This means that if the profiling tool is not connected to the system the communication ISRs are never called, so they can be left on a production system without affecting its execution (except on the system size

footprint). For tracing they also provide a hardware module (J-link or J-trace), as an alternative to system instrumentation.

Segger also provides a free trace viewer tool called System View [SEGc] that works with their own trace format, which is saved in a binary package. The format specifies event categories as contexts. This allows an easier analysis of each event type, as it is possible to go to the next or previous event of that context easily, get data on that specific event (e.g.: Task Run: Runs for 6μs (1089 cycles)), data on all events of that context (frequency, minimum run time, maximum run time, ...) and corresponding terminal output (if applicable). The events can also be viewed in a timeline interface (figure 2.5) with adjustable timing resolution, and how much CPU each context is using.

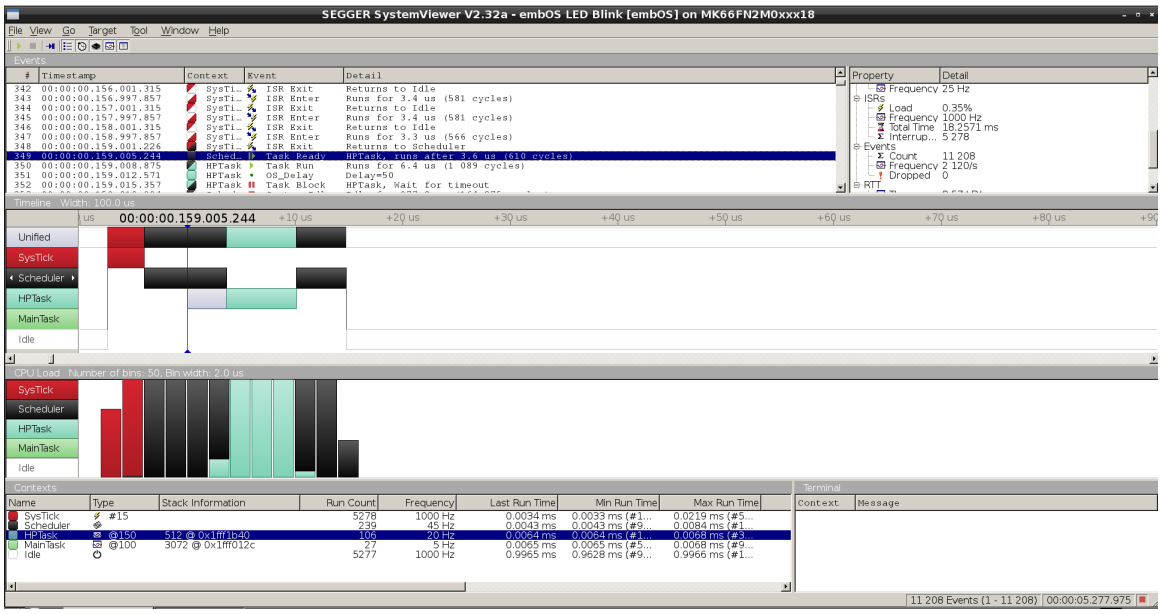

Figure 2.5: Printscreen of Segger System View.

2.7.6 Green Hills Software

Green Hills trace tool [Hilb] features two types of monitors:

- Hardware - trace capture from a hardware trace port;
- Software - capturing events from a system executing on a simulator.

For the hardware monitor [Hila] they provide an hardware device called the `SuperTrace Probe` which connects to a monitoring target via JTAG at 300 MHz speeds, capable of storing up to 4GigaByte (GB) of events with a timestamp resolution of 7.5 ns.
Events can then be viewed on a graphical interface as part of their development Integrated Development Environment (IDE).

2.7.7 RTEMS Trace

The RTEMS project provides a software based monitoring solution, targeting not only to its kernel, but also applications and third party packages [Cora]. Its architecture is based on a linker feature called *wrappers*, which can replace all calls to a given function at linkage time, calling instead a `__wrap_` symbol prefixed version of that function (e.g.: a wrapper function for

`rtems_semaphore_create` would be `__wrap_rtems_semaphore_create`). This new function is defined and provided to the linker (so it does not require modifications to the original source code), which will execute on behalf of the original function. This behaviour is perfect for code instrumentation, as this new function can record that the function was called (and any associated metadata, such as the input arguments or return value) and still invoke the original function via the `__real_` symbol prefixed version of the original function. In addition to the wrappers this tool also uses a separate module called the capture engine. This module can be configured to monitor only certain types of events, and contains a buffer where the events can be stored.

Using this linker feature the RTEMS Trace tool needs two inputs:

1. the functions to be monitored, such that they can be wrapped by the linker;
2. the wrapper function implementation for the monitored functions.

The RTEMS community uses an application (`rtems-tld`) to configure and compile the capture engine and instrumented code (the wrappers), and link it with the application to be monitored. Configuration and wrapper function implementations are provided as one or more `.ini` files. The trace data is retrieved either by printing to screen/console (e.g.: through a serial connection) or by writing to a file, which can be converted to CTF (using the babeltrace tool) and displayed with the Trace Compass tool (refer to section 2.7.1).

2.8 Discussion

A disadvantage of software monitors is that the instrumentation required has an impact on the system performance. The monitoring logic added to the system is usually of significant proportions, and the resulting overhead is easily unacceptable for real-time applications. On top of that, the monitoring harness is usually removed when the monitoring activities end, so the system being monitored is not the system being deployed.
Iyenghar *et al* [IWWP13] propose a solution for this in their paper:

- Use a generic monitoring routine with measurable overhead (e.g.: knowing before hand how many microseconds it takes to monitor a semaphore being obtained);
- Modular approach where the monitoring logic is independent of the communication interface between the target and host;
- Minimize the communication overhead between the target and host systems, by optimizing the encoding and decoding of the transmitted data.

This approach has a few problems for deeply embedded systems. They consider measurable overhead to be minimally intrusive, because the overhead is then considered to be bounded and for that reason it can be accounted for, for instance, in schedulability tests. Since the monitoring overhead is known this allows the monitoring harness to stay in the deployed system. However, it still means that every operation being monitored will take longer than necessary, and the memory and timing requirements of the system will increase. This is acknowledged in the paper, where the authors state that the inclusion of their monitor decreased the number of events the system could handle by a factor of 8.92 using a RS-232 interface.

For communication they propose that events should be sent only in the idle cycles of the CPU. This is also the approach used to send events to the TraceX tool running on the host platform in the ThreadX system. This however increases the monitor impact on the system's code, as it requires the idle thread to be instrumented to check and send events if there are any. A lower impact option would be to have the monitor create a task with the lowest priority possible with that same code to check the buffer and send the events. The only possible downside to this alternative is that the system must not use any task at this priority level, otherwise that task execution will compete with the monitor task.

Having the monitoring logic independent from the communication interface is important for embedded systems, as different systems may use different communication interfaces. Using a modular approach allows for more embedded platforms to be used with the monitor, as new communication interfaces can be easily added.
The format used to transfer data between the target and host platforms is one key area of software-based monitoring where time can be easily saved, by reducing the amount of data transferred.

Regarding the impact of software-based monitors on the monitored system, Iyenghar *et al* [IWWP13] consider that the use of on-chip monitoring alongside a software-based monitor could minimize the monitoring overhead.
For hardware monitors the problem is the lack on visibility on the system's software states. Already in 1981 Plattner *et al* [PN81] noted in their paper that commercial hardware monitors were starting to lose access to the complete sequence of states of a target process.

For Nelissen *et al* [NPP15] the problem with software monitors is that they are mostly implemented as sequential blocks of code, meaning that only one thread can be performing monitoring activities at a time. They consider run-time monitors where the monitors share the same machine as the monitored application. In this scenario a task (i.e.: a system/application thread) writes events to a system-wide global buffer, which is then accessed by the monitors (i.e.: monitoring threads performing run time verifications). In this situation only one thread can be either reading or writing to the buffer, meaning that a thread may be waiting for another task to write an event B to the buffer before it can write an event A. To avoid this they propose a monitoring framework where each event has its own buffer.

This problem does not apply, however, to systems where the monitors are running on a separate machine. In this scenario the events are transmitted to the monitors by a communication interface such as Transmission Control Protocol/Internet Protocol (TCP/IP), rather than having the monitors accessing the buffers directly. In this scenario it is likely that only one communication interface will be used, so it does not matter how much more buffers there are in the system as there is only one way out for the events.

As for the existing tools presented in section 2.7, it is clear that monitoring tools are an important piece of an operating system development toolkit. While most use software monitors, there are still some hardware monitors being developed and used (Green Hills and Segger for instance, although they also provide software monitors). As for the event visualization, the tracealizer tool is used by 20 major operating systems currently in use in the industry today, while trace compass is the open source alternative for open source projects such as Linux or RTEMS. Both these visualization tools make use of standard trace formats (refer to section 2.4) to decode the trace data, meaning

that the graphical part can be made system independent. This is the reason why most operating systems no longer provide a graphical viewer for their monitoring solutions: they just generate standardized trace data which can then be viewed on third-party interfaces.

Trace compass has less features than tracealizer, but since it is free and open source it would be advantageous to use it as the trace visualizer for the system.

2.9 Conclusions

A study of the literature shows that there is quite a mixture of concepts related to monitoring architectures, and what is the purpose of a monitoring tool. The literature focus mainly on the analysis of the system execution, and not so much on the retrieval of those traces in a way that does not impact the execution. For a real-time system running on a resource constrained embedded platform the capture side of monitoring is of utmost importance.

The concepts presented in this chapter will serve as a base for the next chapters where the design and implementation of the proposed monitoring tool will be detailed.

Chapter 3

Monitoring Tool Requirements

This chapter presents the analysis and specification of the software requirements for the developed monitoring tool, taking into account the presented state of the art and the thesis context within Edisoft. These requirements target the sparc leon3 architecture in single core (the multicore version of the tool is described in chapter 6), using SpaceWire (a serial protocol developed by ESA for space applications [Ageb]) as the communication channel and Edisoft's RTEMS Improvement version as the target system on top of a Gaisler's GR712RC development board.

3.1 Purpose

The purpose of the developed monitoring tool is to support the application developer with information collected from the execution of the application. The collected data encompasses thread scheduling, thread stack usage, resource usage, RTEMS configuration tables, generated interrupts and RTEMS API calls. The monitoring tool is made of two components running on different platforms: the stub and the host. Figure 3.1 illustrates how these components interact: the stub runs in the same platform as the application and RTEMS kernel, collecting information about the runtime execution, which is afterwards transmitted to the host.

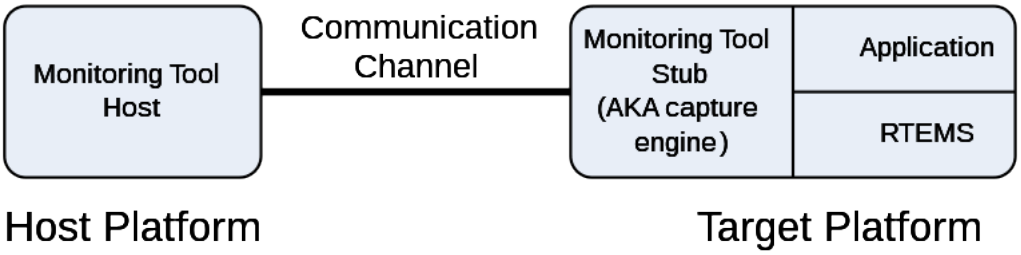

Figure 3.1: Monitoring Tool Platform.

The target platform, which integrates the stub component, requires a communication channel (in this case over SpaceWire, but the version described in chapter 6 uses serial - RS-232) and its memory and temporal requirements are directly related with the number of logs monitored. The monitoring tool interacts with both the RTEMS kernel and the application in order to monitor the events generated. This interaction creates a spatial and temporal overhead as it will take memory and CPU time to write the logs, in addition to the time it takes for the transmission of information from the target to the host.

Since the number of events received by the host can be very large (e.g.: if the stub is sending logs at 1000 Hertz (Hz), each log occupying 48 bytes, the minimum non-volatile memory size required to save all the information during 1 hour amounts to 165 MegaByte (MB)), a DataBase Management System (DBMS)(MySQL) is used to save and manage the data. The data may also be stored in CSV files as an alternative format.

3.2 Constraints

The Monitoring tool is limited by the communication channel speed and its computational parameters, and both the target and the host platforms require a SpaceWire interface in order to send/receive the trace data. The tool is also limited by the speed and memory capabilities of the target platform.

Due to these limitations it is important to offer the application developer flexibility in the configuration phase of the monitoring tool, such that it can be easily tailored to a given embedded platform. On top of this, the application being monitored may also be constrained: to capture interrupts the `rtems_interrupt_catch` primitive is used, meaning that the monitored application must not use this function for the interrupt vectors that the monitored tool was configured to monitor as it will replace the monitoring ISR placed by the monitoring tool. The monitored application may, however, use the `rtems_interrupt_catch` for interrupts vectors that the monitoring tool was not configured to monitor.

3.3 Host Logical Model Description

The main goal of the monitoring host is to receive the logs from the communication channel and store them for the user. There are two different storage alternatives:

1. MySQL Database - stores the logs into a MySQL database;
2. CSV file - stores the logs into a CSV file.

The log reception is performed by a specialized java thread, which will be continuously waiting for a message on the communication channel (see figure 3.2).

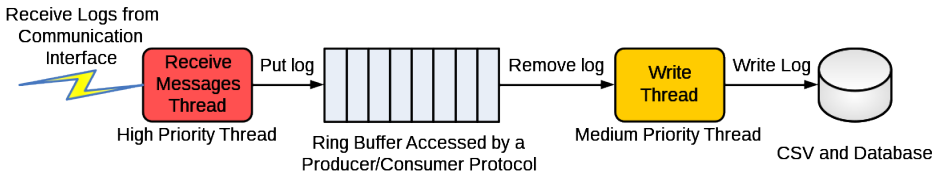

Figure 3.2: Monitoring Tool Host Functional Architecture.

As soon as the message is received, the corresponding log is saved in a ring buffer. This ring buffer will be implemented in RAM memory and will have a limited size (not user configurable). This ring buffer is accessed through a producer/consumer protocol (i.e.: if the buffer is full, the thread which is putting logs into the buffer is blocked; if the buffer is empty, the thread getting the logs is blocked).

The thread that has the highest priority (the Java API allows thread priority definition) exists to ensure that the messages received on the communication's channel buffer (controlled by the operating system - namely the driver controlling the channel) are moved as soon as possible to the monitoring host's ring buffer to avoid data loss over a communication channel's buffer overflow.

A medium priority thread continuously removes the logs from the ring buffer and writes them either to a database (MySQL) or to CSV files (see figure 3.2). This thread has a lower priority

than the `Receive Messages Thread` to lower interference. The size of the ring buffer must be typically large (∼1000 logs) to give enough margin for message bursts. The host can only handle relatively small amounts of data at a time (e.g.: 106 events).

3.4 Transmission Interface Logical Model Description

Data transmission is a very time consuming task, especially for the target system. To give more flexibility to the user and application, the transmission parameters are user configurable. There are two modes through which the information can be transmitted: on-line and off-line modes.

In the on-line mode, the logs are sent periodically by a monitoring tool task running in the target platform. The information is received by the host and is processed while the target is still running. In the off-line mode, the information is kept on the target system until the application ends execution (i.e.: when the `rtems_shutdown` directive is invoked), and only then the data is transmitted.

The on-line mode is not only suited for a live stream of events, but also to ensure that a long running application can be monitored throughout the whole execution, as the off-line mode would require larger amounts of memory to store a corresponding large amount of events. On the other hand the off-line mode has the advantage of less interference of the application, as the events are only sent at the end.

3.5 Specific Requirements and Constraints

This section presents the software requirements for the developed monitoring tool, in the context of the work performed at Edisoft. These requirements originate in the statement of work presented to Edisoft by ESA for the developed monitoring tool, and intend to specify (among all the possible alternatives for its implementation) a monitoring tool for Edisoft's RTEMS Improvement version as the target system on top of a Gaisler's GR712RC development board (sparc leon3 architecture, in single core configuration), with SpaceWire as the communication channel. These requirements do not apply to the multicore version presented in chapter 6, as it uses the arm architecture instead of sparc, and RS-232 instead of SpaceWire for communication.

The requirements presented in this section follow the structure defined in the table 3.1.

Requirement Identifier	This field contains the requirement's unique identifier.
Brief Description	This field contains the requirement's name.
Description	This field contains the full description of the requirement.

Table 3.1: Description of the Requirements Template.

The notation used for Software requirements analysis is UML. This notation was chosen due to its popularity, ease to use and high readability capabilities. The requirements within this chapter have been identified with the following label: `MT-SR-X-N`, which stands for:

- MT - Monitoring Tool;
- SR - Software Requirement;

- X - This identifier shall refer to the requirement identification and shall refer to the subsection where the requirement is defined;
- N - This identifier shall refer to the requirement number.

The X identifier can be divided into the following categories:

- FUNC - Functional requirements;
- PER - Performance requirements;
- TEST - Testability requirements;
- INST - Installation requirements;
- DITC - Design, Implementation and Test Constraints requirements;
- VAL - Verification and Validation requirements.

The combination of the X and N parameters shall define a unique identifier for each requirement, that is, two requirements cannot have the same X and N parameters.

3.5.1 Functional Requirements

The functional requirements for the monitoring tool are presented in tables 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 3.10, 3.11, 3.12, 3.13, 3.14, 3.15, 3.16, 3.17, 3.18, 3.19, 3.20 and 3.21.

Requirement Identifier	MT-SR-FUNC-0010
Brief Description	RTEMS Extension Manager Events Monitor
Description	The Monitoring tool shall be able to monitor events reported using the RTEMS User Extension Manager.

Table 3.2: Requirement MT-SR-FUNC-0010.

Requirement Identifier	MT-SR-FUNC-0020
Brief Description	Monitoring Other Events
Description	The Monitoring tool shall be able to monitor scheduling events other than those reported by the RTEMS User Extension Manager.

Table 3.3: Requirement MT-SR-FUNC-0020.

Requirement Identifier	MT-SR-FUNC-0030
Brief Description	Interrupts Monitor
Description	The Monitoring tool shall be able to monitor the interrupt generation for the LEON 3 (GR712RC).

Table 3.4: Requirement MT-SR-FUNC-0030.

Requirement Identifier	MT-SR-FUNC-0040
Brief Description	User Calls to RTEMS APIs Monitor
Description	The Monitoring tool shall be able to monitor the calls from the user specified RTEMS APIs that occur in the user's application.

Table 3.5: Requirement MT-SR-FUNC-0040.

Requirement Identifier	MT-SR-FUNC-0060
Brief Description	Stack Monitor
Description	The Monitoring tool shall be able to monitor the task stack usage at task context switch.

Table 3.6: Requirement MT-SR-FUNC-0060.

Requirement Identifier	MT-SR-FUNC-0070
Brief Description	RTEMS Configuration Monitor
Description	The Monitoring tool shall monitor the following RTEMS configuration tables defined by the user: • RTEMS Workspace address; • RTEMS Workspace size; • Number of device drivers; • Number of microseconds per clock tick; • Number of clock ticks per timeslice.

Table 3.7: Requirement MT-SR-FUNC-0070.

Requirement Identifier	MT-SR-FUNC-0080
Brief Description	Log Identification Data - Tasks, Semaphores and Messages Queues
Description	The Monitoring tool shall report tasks, semaphores and messages queues by their address, symbol and rtems id.

Table 3.8: Requirement MT-SR-FUNC-0080.

Requirement Identifier	MT-SR-FUNC-0085
Brief Description	Log Identification Data - Interrupts
Description	The Monitoring tool shall report interrupts by their vector in the Interrupt Vector Table.

Table 3.9: Requirement MT-SR-FUNC-0085.

Requirement Identifier	MT-SR-FUNC-0087
Brief Description	Application Debug Message
Description	The Monitoring tool shall provide a primitive for the application to log a custom message.

Table 3.10: Requirement MT-SR-FUNC-0087.

Requirement Identifier	MT-SR-FUNC-0090
Brief Description	Task Monitor
Description	The Monitoring tool shall identify when each task was executed, when it became ready, when it finished its processing and its priority over time.

Table 3.11: Requirement MT-SR-FUNC-0090.

Requirement Identifier	MT-SR-FUNC-0130
Brief Description	Types of Events to Monitor
Description	The Monitoring tool shall be able to enable/ disable event capturing by a set of user defined functions. It shall be able to filter the following events: • Task schedulability events (e.g. task create event) • Task stack events • RTEMS API calls (each manager): – Task Manager – Interrupt Manager – Clock Manager – Timer Manager – Semaphore Manager – Message Queue Manager – Event Manager – IO Manager – Error Manager – Rate Monotonic Manager – User Extension Manager

Table 3.12: Requirement MT-SR-FUNC-0130.

Requirement Identifier	MT-SR-FUNC-0140
Brief Description	Task Monitor Threshold Definition
Description	The user shall be able to define an upper and lower task priority threshold. The tool shall log the events generated by tasks within this limit (stack usage, task events).

Table 3.13: Requirement MT-SR-FUNC-0140.

Requirement Identifier	MT-SR-FUNC-0150
Brief Description	Log Timestamp
Description	The Monitoring tool shall provide time stamping capabilities for each log.

Table 3.14: Requirement MT-SR-FUNC-0150.

Requirement Identifier	MT-SR-FUNC-0170
Brief Description	Transfer Types
Description	The Monitoring tool shall support both on-line and offline modes for log transfer.

Table 3.15: Requirement MT-SR-FUNC-0170.

Requirement Identifier	MT-SR-FUNC-0180
Brief Description	On-Line Log Transfer
Description	In the on-line mode the logs shall be transmitted periodically through a periodic task (see MT-SR-DITC-0030).

Table 3.16: Requirement MT-SR-FUNC-0180.

Requirement Identifier	MT-SR-FUNC-0190
Brief Description	Off-Line Log Transfer
Description	In the off-line mode the logs shall be transmitted when the application ends (when the `rtems_shutdown` directive is called).

Table 3.17: Requirement MT-SR-FUNC-0190.

Requirement Identifier	MT-SR-FUNC-0195
Brief Description	Force Log Transfer
Description	A primitive shall be available in order to force the transfer of a target number of logs that are still stored in the stub.

Table 3.18: Requirement MT-SR-FUNC-0195.

Requirement Identifier	MT-SR-FUNC-0200
Brief Description	SpaceWire as a Transmission Channel
Description	The Monitoring tool shall transmit the logs from the target platform to the host platform through a SpW interface.

Table 3.19: Requirement MT-SR-FUNC-0200.

Requirement Identifier	MT-SR-FUNC-0210
Brief Description	Store Logs
Description	The Monitoring tool shall save the logs received from the target platform to database or CSV file in the host platform.

Table 3.20: Requirement MT-SR-FUNC-0210.

Requirement Identifier	MT-SR-FUNC-0220
Brief Description	Enable/ Disable Monitoring Tool
Description	It shall be possible to enable and disable the Monitoring tool in execution time.

Table 3.21: Requirement MT-SR-FUNC-0220.

3.5.2 Performance Requirements

The performance requirements for the monitoring tool are presented in tables 3.22 and 3.23.

Requirement Identifier	MT-SR-PER-0010
Brief Description	Maximum Temporal Overhead of an Event Saving
Description	The Monitoring tool shall add no more than 5% of temporal overhead to save an event.

Table 3.22: Requirement MT-SR-PER-0010.

Requirement Identifier	MT-SR-PER-0020
Brief Description	Maximum Temporal Overhead of an Event Transmission
Description	The Monitoring tool shall add no more than 5% of temporal overhead for each event transmitted.

Table 3.23: Requirement MT-SR-PER-0020.

3.5.3 Testability Requirements

The testability requirements for the monitoring tool are presented in tables 3.24, 3.25 and 3.26.

Requirement Identifier	MT-SR-TEST-0010
Brief Description	Test Naming Rule
Description	The test name shall be unique and follow this naming rule: mne_tt_nnnaa.ext, where: • mne: mnemonic code • tt: type of test • nnn: test number • aa: subtest number • ext: extension

Table 3.24: Requirement MT-SR-TEST-0010.

Requirement Identifier	MT-SR-TEST-0020
Brief Description	Execution Report Format
Description	The test suite shall generate the execution report in text format.

Table 3.25: Requirement MT-SR-TEST-0020.

Requirement Identifier	MT-SR-TEST-0030
Brief Description	Test pass/fail Criteria
Description	Test case shall pass successfully if all its steps are successful. One step is considered successful if the obtained output is equal to the expected output.

Table 3.26: Requirement MT-SR-TEST-0030.

3.5.4 Design, Implementation and Test Constraints

The design, implementation and test constraints for the monitoring tool are presented in tables 3.27, 3.28, 3.29, 3.30, 3.31, 3.32, 3.33, 3.34, 3.35 and 3.36.

Requirement Identifier	MT-SR-DITC-0020
Brief Description	Stub Programming Language
Description	The Monitoring tool stub shall be implemented in C language.

Table 3.27: Requirement MT-SR-DITC-0020.

Requirement Identifier	MT-SR-DITC-0030
Brief Description	Monitoring Tool Transmission Parameters Configuration
Description	The user shall be able to configure the following communication parameters (on the on-line mode): • Monitoring tool periodic task priority; • Monitoring tool periodic task period; • Number of messages to send per period.

Table 3.28: Requirement MT-SR-DITC-0030.

Requirement Identifier	MT-SR-DITC-0040
Brief Description	Stub Memory Buffer
Description	The memory buffer on the target system shall be implemented through a ring buffer. Each time a log is read, it shall be automatically deleted from the buffer. See MT-SR-DITC-0050.

Table 3.29: Requirement MT-SR-DITC-0040.

Requirement Identifier	MT-SR-DITC-0050
Brief Description	Ring Buffer Configuration
Description	It shall be possible to configure the ring buffer used by the target component of the Monitoring tool in the following aspects: • Buffer size; • Buffer start address; • Buffer overwrite settings when at full capacity: – Overwrite oldest logs; – Do not overwrite oldest logs, new logs are lost.

Table 3.30: Requirement MT-SR-DITC-0050.

Requirement Identifier	MT-SR-DITC-0060
Brief Description	Store Logs on MySQL Database
Description	The Monitoring tool on the host side shall save the logs to a MySQL database.

Table 3.31: Requirement MT-SR-DITC-0060.

Requirement Identifier	MT-SR-DITC-0070
Brief Description	Maximum Size of the Transmitted Messages
Description	The messages transmitted from the Monitoring tool stub to the host shall occupy a maximum of 61 bytes. Each message sent/received will correspond to a log.

Table 3.32: Requirement MT-SR-DITC-0070.

Requirement Identifier	MT-SR-DITC-0080
Brief Description	Interrupt Catch Constraint
Description	The `rtems_interrupt_catch` primitive shall not be used by the target application if interrupt events are being monitored.

Table 3.33: Requirement MT-SR-DITC-0080.

Requirement Identifier	MT-SR-DITC-0090
Brief Description	Timestamp Accuracy
Description	The Monitoring tool shall manage timestamps of events and timing statistics with the accuracy of a microsecond or better (target CPU clock cycle).

Table 3.34: Requirement MT-SR-DITC-0090.

Requirement Identifier	MT-SR-DITC-0100
Brief Description	Configure Host Address in the Stub
Description	The user shall be able to configure the SpW host address in compilation time.

Table 3.35: Requirement MT-SR-DITC-0100.

Requirement Identifier	MT-SR-DITC-0110
Brief Description	SpW Hardware Host Encapsulation
Description	The host SpW device shall be properly encapsulated in order to ease port to other environment.

Table 3.36: Requirement MT-SR-DITC-0110.

3.5.5 Installation Requirements

The single installation requirement for the monitoring tool is presented in table 3.37.

Requirement Identifier	MT-SR-INST-0010
Brief Description	Stub Installation
Description	The result of compilation and linking of the stub with the application shall be an executable binary file. The user shall copy this binary file to the memory of the final target.

Table 3.37: Requirement MT-SR-INST-0010.

3.5.6 Verification and Validation

The verification and validation requirements for the monitoring tool are presented in tables 3.38 and 3.39.

Requirement Identifier	MT-SR-VAL-0010
Brief Description	Target Board
Description	The Monitoring tool shall be tested with a GR712RC-BOARD.

Table 3.38: Requirement MT-SR-VAL-0010.

Requirement Identifier	MT-SR-VAL-0020
Brief Description	SpaceWire Interface
Description	A SpW interface is required in both the target and host systems in order to transfer the logged information.

Table 3.39: Requirement MT-SR-VAL-0020.

3.6 Conclusions

This chapter presented an overview of the developed monitoring tool and its requirements: functional, performance, testability, design, implementation and test constraints, installation and verification and validation. The following chapters use this specification to present the monitoring tool design 4, implementation A and testing 5.

Chapter 4

Monitoring Tool Design

This chapter describes the monitoring tool design, based on the requirements presented in the previous chapter, and follows the UML design method with the use of sequence diagrams to aid the system comprehension:

- for static design aspects, UML package and class diagrams are used;
- for dynamic design aspects, sequence diagrams are used.

The Monitoring tool is composed of two different components: stub and host.
The stub runs in the same platform as the application and the RTEMS kernel. The RTEMS version used/tested with the unicore version of the monitoring tool is the Edisoft's RTEMS Improvement (connecting with the host via SpaceWire).

The monitoring target is based on the Cobham-Gaisler GR712-RC development board (sparc/leon3 fault-tolerant dual-core) while the Monitoring host runs on a Windows 7 machine (SCOC3 Electrical Ground Support Equipment (EGSE)) lent to Edisoft by ESA, which contains a Teletel SpaceWire interface card. To operate the SpaceWire card a web service is used (Teletel iSAFT RunTime Environment (RTE)), acting as the driver.

4.1 Architectural Design

This section describes the Monitoring tool architecture. Figure 4.1 illustrates the relationship between the major components of the Monitoring tool.

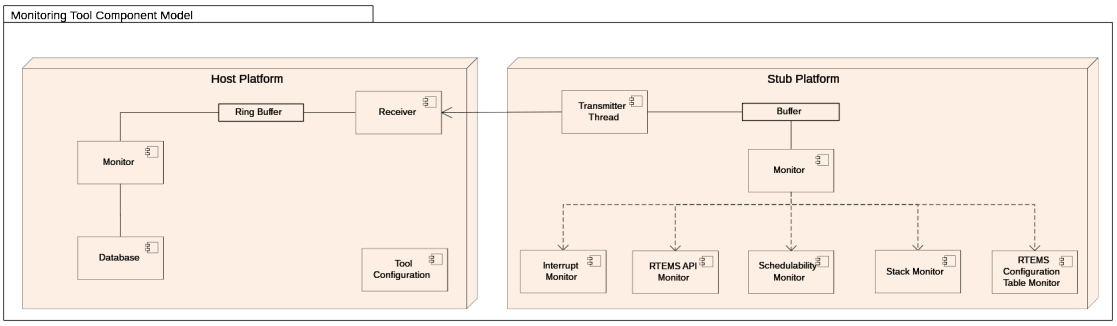

Figure 4.1: Relationship between the Monitoring Tool Major Components.

It is composed of two components: the Host platform and the Stub platform. The stub collects the information about the runtime execution environment to a local buffer (in this case only one buffer is used, as it is intended for a single core system. The version described in chapter 6 uses one buffer per core), which will be sent to the host platform:

- when the application execution terminates;

- periodically - according to a scheduling policy defined by the user (periodic task with user defined period and priority);
- at any time during execution, when ever the monitored application calls a primitive to flush a number of events.

The host node receives the collected information and temporarily saves it to a local ring buffer. The purpose of this buffer is to reduce the possibility of a buffer overflow at the communication channel device driver level, which is filled every time a new byte (or a number of bytes, if the device has an hardware buffer to diminish the CPU interruptions, such as serial communication boards with First In, First Out (FIFO) UART 16550A [Pro], which has 16 byte hardware FIFOs where the data is stored before interrupting the CPU - only one interruption every 16 bytes) arrives. Even if the driver has a configurable buffer size, their buffer is filled via a CPU interrupt, meaning that any data that is received by the communication channel is stored on the driver's buffer, however any application wanting to access that data is dependent on the system scheduler to have access to that resource. In that sense, and as depicted in figure 3.2, a high priority thread continuously fetches data from the device driver's buffer into the host node's buffer (which size is controlled by the host), from which a medium priority thread (as to reduce the interference with the higher priority thread) continuously checks the ring buffer to see if there are any new messages. If so, it removes them (to free more space in the buffer) and places them in the database or into a CSV file (the user shall choose between one of the two before stating the host).

The monitoring tool is not integrated within the RTEMS source code: the application developer may choose to use (or not) the Monitoring tool by linking the appropriate object files. This allows an efficient implementation of the tool since a change in the default parameters (e.g. task priority thresholds, etc) only requires a recompilation of the tool, instead of the RTEMS library or application. The linkage of the monitoring tool with the application and RTEMS library is performed through the use of "wrappers" (ld –wrap symbol) supported by the linker. This allows the monitoring tool to intercept specific RTEMS primitives, such as `rtems_initialize_early` (to startup the monitoring tool), `_Thread_Clear_state` (which can ready a task) or `rtems_task_create` (a RTEMS API call).

4.2 Stub Static Architecture

This section discusses the monitoring stub architecture. Figure 4.2 shows the class diagram of the Monitoring tool stub.
The stub is divided into the following components:

- Buffer;
- Task;
- Manager;
- API;
- InterruptInterceptor;
- DefaultConfiguration;
- SpwTransmitter;

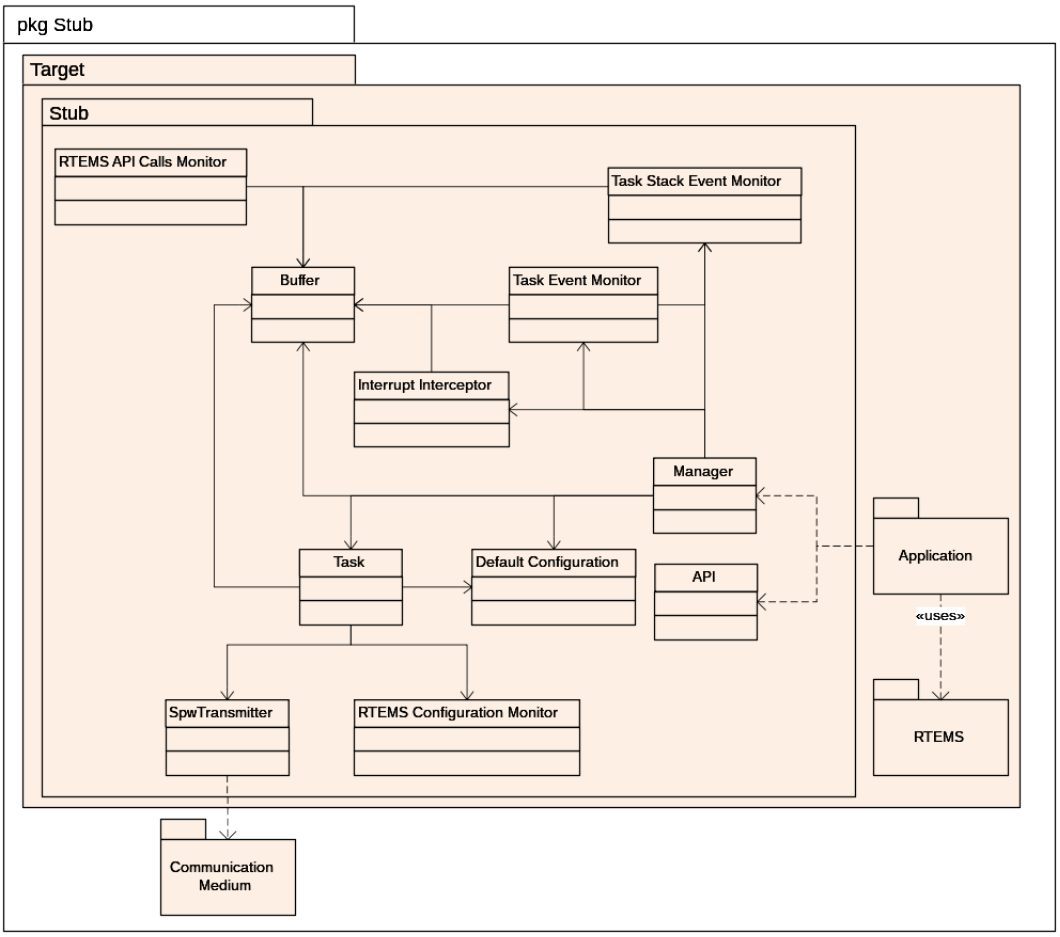

Figure 4.2: Monitoring Tool Stub Class Diagram.

- RTEMSAPICallsMonitor;
- TaskEventMonitor;
- TaskStackEventMonitor;
- RTEMSConfigurationMonitor.

4.2.1 Buffer

Description:

The Buffer component contains the events captured by the tool and that are ready to be sent (derived from requirement MT-SR-FUNC-0150 and MT-SR-FUNC-0170). It is composed of a ring buffer, accessed by multiple producers and one consumer (where one of the producers may be from an interrupt source). The dimension of the buffer is defined in the DefaultConfiguration component. Regarding the performance requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-PER-0010.

Regarding the design requirements, this architecture component has to conform to the constraints imposed by the requirements:

- MT-SR-DITC-0020;
- MT-SR-DITC-0040;
- MT-SR-DITC-0050;
- MT-SR-DITC-0070;
- MT-SR-DITC-0090.

Regarding the verification and validation requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-VAL-0010.

Interface with other Architecture components:

The Buffer component provides an interface (refer to table 4.1) which allows writing events on the buffer and sending the registered events from the buffer to the host.

Interface Signature	**Interface Purpose**	**Architectural Components that use the Interface**
ptrTimeline_event timeline_start_write_event(eventType type);	This function writes the initial fields of the event (time instant and type).	RTEMSAPICallsMonitor TaskEventMonitor TaskStackEventMonitor InterruptInterceptor
void timeline_commit_write_event();	Announces that the event has been written.	RTEMSAPICallsMonitor TaskEventMonitor TaskStackEventMonitor InterruptInterceptor
uint32_t timeline_send_events(uint32_t events2Flush)	Sends the events stored in the ring buffer through the communication interface.	Manager Task API
void timeline_send_message(uint8_t *message, unsigned int size);	Sends a message through the communication interface.	RTEMSConfigurationMonitor
void timeline_send_sync_message();	Sends the synchronization message through the communication interface.	Manager Task API

Table 4.1: Monitoring Stub Buffer Interface.

4.2.2 Task

Description:

The Task component represents a periodic task that sends the events contained in the Buffer to the SpwTransmitter. It has a user defined period, priority and number of messages to send within each period that can be defined in the DefaultConfiguration package (derived from requirements MT-SR-FUNC-0170 and MT-SR-FUNC-0180). Regarding the performance requirements, this architecture component has to conform to the constraints imposed by the requirements:

- MT-SR-PER-0020.

Regarding the design requirements, this architecture component has to conform to the constraints imposed by the requirements:

- MT-SR-DITC-0020;
- MT-SR-DITC-0030;
- MT-SR-DITC-0070.

Regarding the verification and validation requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-VAL-0010.

Interface with other Architecture components:

The Task component provides the periodic task (refer to table 4.2) needed to send the stored events to the host in online mode

Interface Signature	**Interface Purpose**	**Architectural Components that use the Interface**
void timeline_task(rtems_task_argument dummy)	The Monitoring task.	Manager

Table 4.2: Monitoring Stub Task Interface.

4.2.3 Manager

Description:

The Manager component manages the initialization and shutdown of the tool components. If the Monitoring tool is being used in offline mode then the manager sends all the application events stored in the Buffer component to the Host before the shutdown. In online mode it starts the task component to send the events periodically (derived from requirements MT-SR-FUNC-0170 and MT-SR-FUNC-0190). Regarding the performance requirements, this architecture component has to conform to the constraints imposed by the requirements:

- MT-SR-PER-0020.

Regarding the design requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-DITC-0020;
- MT-SR-DITC-0070.

Regarding the verification and validation requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-VAL-0010.

Interface Signature	Interface Purpose	Architectural Components that use the Interface
void timeline_initialize();	This function initializes the Monitoring tool.	RTEMS

Table 4.3: Monitoring Stub Manager Interface.

Interface with external Architecture components:

The Manager component provides an interface (refer to table 4.3) to RTEMS that is not used by the Monitoring tool itself. This interface allows the Monitoring tool to be initialized by RTEMS.

4.2.4 API

Description:

The API component provides a set of functionalities to the programmer through which the application can enable or disable the tool at runtime (derived from requirement MT-SR-FUNC-0220), force the transmission of a target number of messages (derived from requirement MT-SR-FUNC-0195) and allow the application to log custom debug messages (derived from requirement MT-SR-FUNC-0087). Regarding the performance requirements, this architecture component has to conform to the constraints imposed by the requirements:

- MT-SR-PER-0020.

Regarding the design requirements, this architecture component has to conform to the constraints imposed by the requirements:

- MT-SR-DITC-0020.

Regarding the verification and validation requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-VAL-0010.

Interface with external Architecture components:

The API component provides an interface (refer to table 4.4) to the application that is not used by the Monitoring tool itself. This interface allows the application to interact with the Monitoring tool.

4.2.5 InterruptInterceptor

Description:

The InterruptInterceptor component contains the interrupt interceptor to catch interrupt events. The interrupt types to monitor are defined in the DefaultConfiguration component (derived from requirements MT-SR-FUNC-0030 and MT-SR-FUNC-0085). Regarding the performance requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-PER-0010.

Interface Signature	Interface Purpose	Architectural Components that use the Interface
void monitoring_enable (int);	Enable or disable the monitoring activities during runtime.	Application
boolean timeline_write_user_event(uint32_t message)	Allows the application to log a custom message.	Application
int monitoring_flush(uint32_t total)	Sends a given number of logged messages to the monitoring host.	Application

Table 4.4: Monitoring Stub API Interface.

Regarding the design requirements, this architecture component has to conform to the constraints imposed by the requirements:

- MT-SR-DITC-0020;
- MT-SR-DITC-0080.

Regarding the verification and validation requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-VAL-0010.

Interface with other Architecture components:

The InterruptInterceptor component provides an interface (refer to table 4.5) to monitor interrupts.

Interface Signature	Interface Purpose	Architectural Components that use the Interface
void timeline_initialize_interrupts();	Initializes the interrupt event monitoring.	Manager

Table 4.5: Monitoring Stub Interrupt Interceptor Interface.

4.2.6 DefaultConfiguration

Description:

The DefaultConfiguration component contains the user defined default tool configuration parameters (derived from requirement MT-SR-FUNC-0130 and MT-SR-FUNC-0140).
Regarding the design requirements, this architecture component has to conform to the constraints imposed by the requirements:

- MT-SR-DITC-0020;
- MT-SR-DITC-0030;
- MT-SR-DITC-0050;
- MT-SR-DITC-0100.

Regarding the verification and validation requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-VAL-0010.

Interface with other Architecture components:

The DefaultConfiguration component provides the default Monitoring tool configuration. This component allows including or removing each filter/ monitoring to satisfy the user needs.

4.2.7 SpwTransmitter

Description:

The SpwTransmitter component contains the needed SpW transmitter functionalities (derived from requirement MT-SR-FUNC-0200). Regarding the performance requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-PER-0020.

Regarding the design requirements, this architecture component has to conform to the constraints imposed by the requirements:

- MT-SR-DITC-0020;
- MT-SR-DITC-0070;
- MT-SR-DITC-0100.

Regarding the verification and validation requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-VAL-0010;
- MT-SR-VAL-0020.

Interface with other Architecture components:

The Transmitter component provides the interface (refer to table 4.6) which allows transmitting the collected information through SpaceWire to the host.

Interface Signature	**Interface Purpose**	**Architectural Components that use the Interface**
int monitoring_spw_connect();	Connects to the tool host.	Manager
void monitoring_write_spw(uint8_t vector[] , unsigned int dim);	Writes a message to the host.	Buffer

Table 4.6: Monitoring Stub SpW Interface.

4.2.8 RTEMSAPICallsMonitor

Description:

The RTEMSAPICallsMonitor component contains the wrappers to log the RTEMS API calls (derived from requirements MT-SR-FUNC-0010, MT-SR-FUNC-0020, MT-SR-FUNC-0040 and MT-SR-FUNC-0080). The RTEMS API creates an interface for task, interrupt, clock, timer, semaphore, message queue, event management, I/O, fatal error, rate monotonic and user extension managers. Regarding the performance requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-PER-0010.

Regarding the design requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-DITC-0020.

Regarding the verification and validation requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-VAL-0010.

Interface with other Architecture components:

The RTEMSAPICallsMonitor component provides the wrappers for the RTEMS API managers. For each manager there are a set of wrappers used by the target application which use the interfaces MT-ADD-0001 and MT-ADD-0005 from the Buffer component to register the logs.

4.2.9 TaskEventMonitor

Description:

The TaskEventMonitor component contains the wrappers to log the task events (derived from requirements MT-SR-FUNC-0080, MT-SR-FUNC-0090 and MT-SR-FUNC-0140). Regarding the performance requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-PER-0010.

Regarding the design requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-DITC-0020.

Regarding the verification and validation requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-VAL-0010.

Interface with external Architecture components:

The TaskEventMonitor component provides an interface (refer to table 4.7) to RTEMS that is not used by the Monitoring tool itself. This interface allows RTEMS to log task events.

Interface Signature	Interface Purpose	Architectural Components that use the Interface
boolean timeline_task_event (rtems_tcb *current_task , rtems_tcb *new_task);	Stores a task event for the task. The event can be creating, starting, deleting or switch.	RTEMS

Table 4.7: Monitoring Stub Task Event Monitor Interface.

4.2.10 TaskStackEventMonitor

Description:

The TaskStackEventMonitor component contains the wrappers to log the task stack events (derived from requirement MT-SR-FUNC-0060 and MT-SR-FUNC-0140). Regarding the performance requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-PER-0010.

Regarding the design requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-DITC-0020.

Regarding the verification and validation requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-VAL-0010.

Interface with other Architecture components:

The TaskStackEventMonitor component provides an interface (refer to table 4.8) to monitor task stacks.

Interface Signature	Interface Purpose	Architectural Components that use the Interface
void timeline_init_stack(rtems_tcb *thread);	Initializes the task stack filling it with a predefined pattern.	TaskEventMonitor
void timeline_write_stack_event(rtems_tcb *thread);	Writes a stack event to the buffer from the parameter thread.	TaskEventMonitor

Table 4.8: Monitoring Stub Task Stack Event Monitor Interface.

4.2.11 RTEMSConfigurationMonitor

Description:

The RTEMSConfigurationMonitor component contains the wrappers to log the RTEMS Configuration tables (derived from requirement MT-SR-FUNC-0070). Regarding the performance requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-PER-0010.

Regarding the design requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-DITC-0020.

Regarding the verification and validation requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-VAL-0010.

Interface with other Architecture components:

The RTEMSConfigurationMonitor component provides an interface (refer to table 4.9) to retrieve the RTEMS Configuration table information.

Interface Signature	**Interface Purpose**	**Architectural Components that use the Interface**
void timeline_send_configuration_message()	Sends the configuration message through the communication interface.	Manager Task API

Table 4.9: Monitoring Stub RTEMS Configuration Monitor Interface.

4.3 Host

This section discusses the host architecture. Figure 4.3 illustrates the class diagram of the host side of the Monitoring tool.
As depicted, the host is divided into the following main components:

- Receiver;
- Database;
- Main;
- Util.

4.3.1 Receiver

Description:

The Receiver component contains the classes to connect the host to the stub using the communication medium (SpaceWire) and receive raw events which contain the information about each event monitored by the stub (derived from requirement MT-SR-FUNC-0200).
Regarding the design requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-DITC-0070;
- MT-SR-DITC-0110;

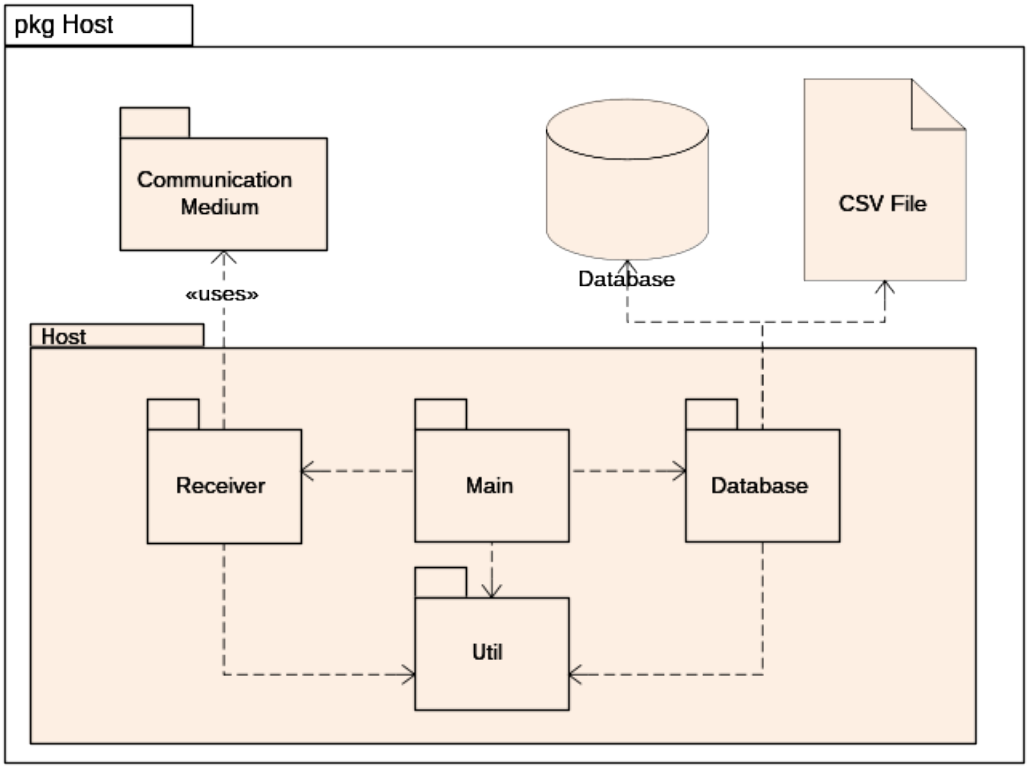

Figure 4.3: Monitoring Tool Host Class Diagram.

Regarding the verification and validation requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-VAL-0020.

Interface with other Architecture components:

The Receiver component provides an interface (refer to table 4.10) to retrieve the monitored data gathered with the stub. This data will be supplied to the Database component so it can be stored either in a database or a CSV file.

Interface Signature	**Interface Purpose**	**Architectural Components that use the Interface**
public class SpWReceiver;	Class that reads the received data from the SpaceWire communication interface.	Main

Table 4.10: Monitoring Host Receiver Interface.

4.3.2 Database

Description:

The Database component contains the classes to connect to the database, including creating, deleting and accessing the database, and to write data in the CSV files, including creating the files.

Each execution is stored on a separate database or CSV file directory (derived from requirement MT-SR-FUNC-0210).

Regarding the design requirements, this architecture component has to conform to the constraints imposed by the requirement:

- MT-SR-DITC-0060.

Interface with other Architecture components:

The Database component provides an interface (refer to table 4.11) to manage the storage mediums (database or CSV files) and provides an interface to store the received information.

Interface Signature	**Interface Purpose**	**Architectural Components that use the Interface**
public class DatabaseWriter;	This class contains the management facilities to access the database. It defines the names of the database tables and has functions to create and connect to the database and to create tables	Main
public class CsvWritter;	This class contains the management facilities to access the CSV files. It defines the names of the CSV files and has functions to create directories and files, including writing to the files.	Main

Table 4.11: Monitoring Host Database Interface.

4.3.3 Main

Description:

The Main component cannot be directly traced to a requirement, but is needed to connect all the other components: it parses the configuration for the host and starts the Receiver and Database services.

Interface with other Architecture components:

The Main component provides an entry point to start the Monitoring tool host, so there are not any architectural components that use this interface.

4.3.4 Util

Description:

The Util component cannot be directly traced to a requirement, but is needed because it contains auxiliary methods and classes used by other components.

Interface with other Architecture components:

The Util component provides an interface (refer to table 4.12) with several useful functions such as the ring buffer class which stores the events temporarily (until they are inserted into the database or CSV file). It also makes available a set of raw types to store the events retrieved by the Receiver component.

Interface Signature	**Interface Purpose**	**Architectural Components that use the Interface**
public SynchronousRingBuffer(int capacity);	Create a ring buffer to store the retrieved data.	Main
public void putObject(Object obj);	Places a new object inside the ring buffer.	Receiver
public Object removeObject();	Removes an object from the ring buffer.	Database
public static short unsignedByteToShort(byte input);	Converts an unsigned byte to a signed short.	Receiver

Table 4.12: Monitoring Host Util Interface.

4.4 Dynamic Architecture - Monitoring Tool Stub

This section describes the dynamic architecture of the Monitoring tool stub, which introduces some temporal overhead in the following stages:

- during the initialization phase;
- when it is saving the information about an event;
- when it is sending the collected information to the host machine.

The sequence diagram in figure 4.4 describes how the tool is initialized.

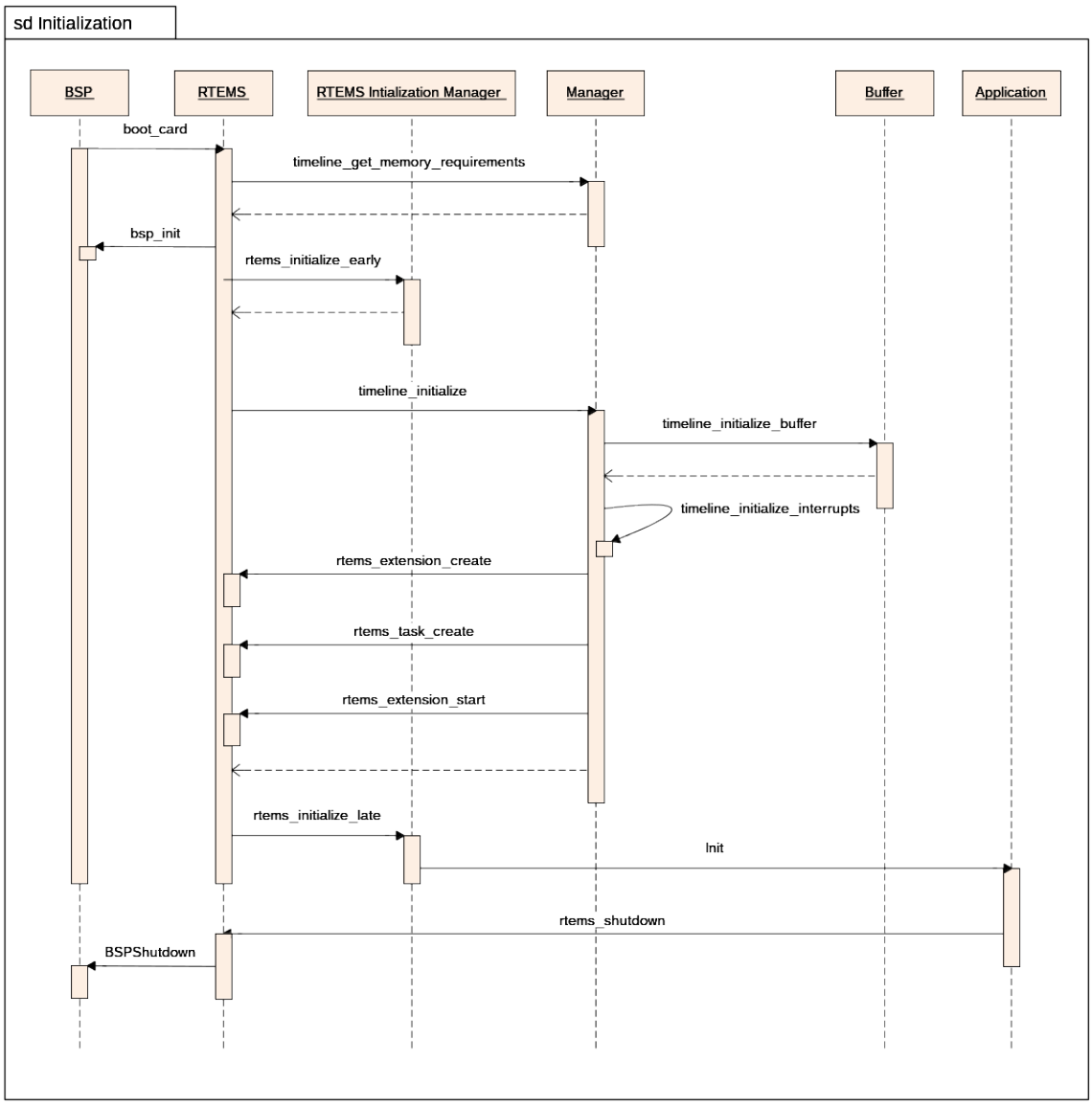

Figure 4.4: Monitoring Tool Initialization Sequence Diagram.

The following subsections describe how and when the logging is performed, how the data is transmitted to the host platform and how the clock stamping is performed in order to get a greater accuracy.

4.4.1 Logging

This subsection is subdivided into subsections that further describe how the logging is performed and how it affects the application at a temporal level. The "filter event" condition (and similars) describe the condition that is analyzed in order to save or not the event. If "filter event" is true then the event is not saved. If "filter event" is false, then the event is saved. The same is valid for "filter task" and others.

4.4.1.1 Interrupt Generation Logging

This subsection discusses how an interrupt event is logged by the monitoring tool. As depicted in figure 4.5, when a monitored interrupt occurs the RTEMS kernel calls a function which corresponds to an ISR interceptor. If the interrupt vector is configured to be monitored, the interrupt event will be recorded into the monitoring tool buffer twice: when the application ISR starts executing and when the application ISR finishes executing (creating a slight temporal overhead during interrupt processing).

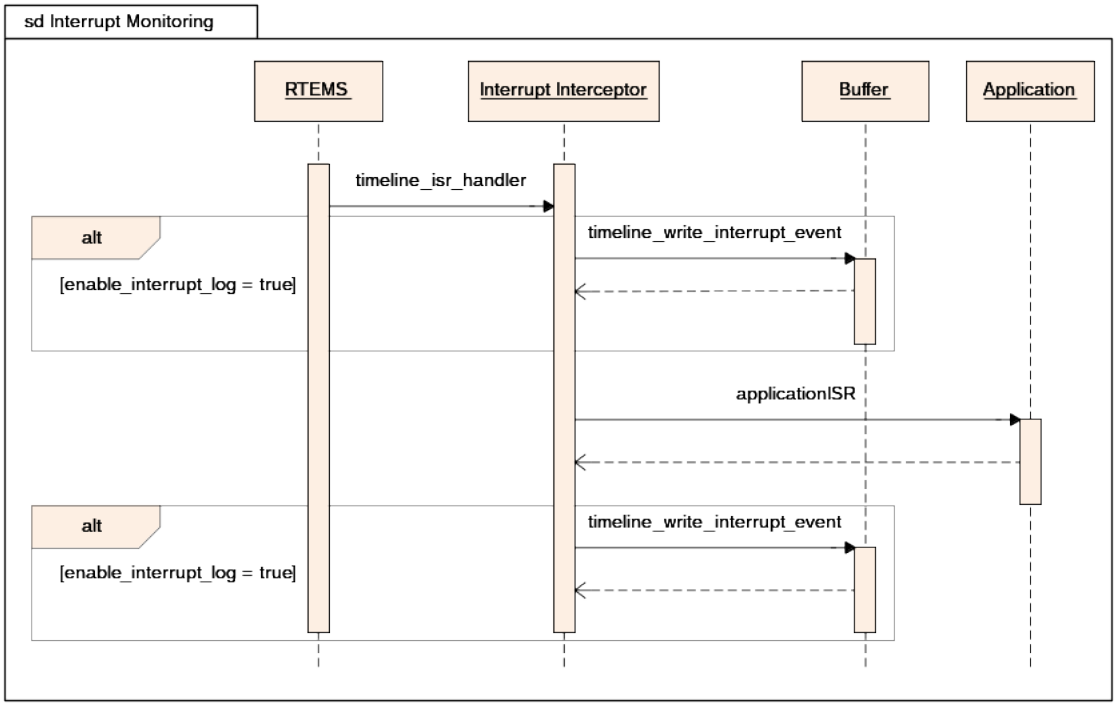

Figure 4.5: Monitoring Tool Interrupt Logging Sequence Diagram.

This interrupt wrapping mechanism is possible through the `rtems_interrupt_catch` directive, which allows the tool to replace the current interrupt vector ISR handler with a new one (the monitoring tool ISR interceptor). The monitoring tool ISR logs the interrupt event and calls the original ISR handler that was replaced. This gives more flexibility to the monitoring tool user by making the interrupt monitoring transparent to the application code, but does not work if the application installs a new ISR (or calls the `rtems_interrupt_catch` directive) for an interrupt vector that is being monitored, as it would replace the ISR interceptor and then the monitoring tool would be unable to log those events (an usually seldom occurrence since the establishment of application ISR occurs during device driver initialization, which is performed prior to the initialization of the monitoring tool).

4.4.1.2 Task Scheduling Logging

The Task scheduling logging is performed when a task changes state. This can occur during a task dispatching routine (the heir task switches to the executing state while the current task switches to the ready/blocked/suspended/timed state), by a RTEMS API call (e.g. a `rtems_semaphore_release` can unblock another task) or by a timer expiration (a task expediting event). Figure 4.6 illustrates the task state transitions monitored by the Monitoring tool.

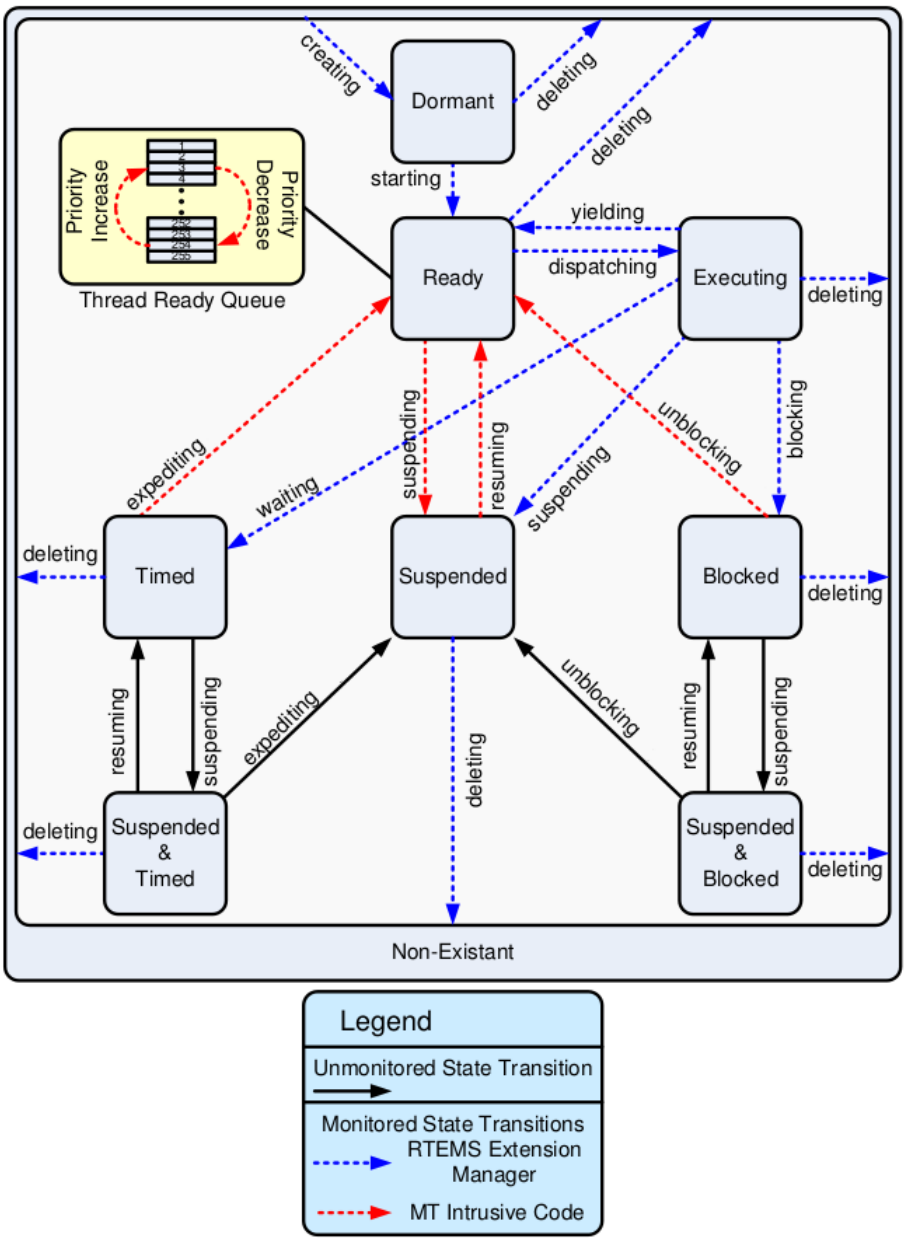

Figure 4.6: Monitored RTEMS Task State Transitions.

Table 4.13 illustrates the mapping between the RTEMS task states and the reduced set of states represented in figure 4.6. Figure 4.7 represents the Task Scheduling sequence diagram.

RTEMS task state (as defined in states.h)	**Monitoring tool task state**
STATES_READY	"Ready"
STATES_DORMANT	"Dormant"
STATES_SUSPENDED	"Suspended" or "Blocked and Suspended"
STATES_TRANSIENT	Not implemented
STATES_DELAYING	"Timed" or "Timed and Suspended"
STATES_WAITING_FOR_TIME	"Timed" or "Timed and Suspended"
STATES_WAITING_FOR_BUFFER	"Blocked" or "Blocked and Suspended"
STATES_WAITING_FOR_SEGMENT	"Blocked" or "Blocked and Suspended"
STATES_WAITING_FOR_MESSAGE	"Blocked" or "Blocked and Suspended"
STATES_WAITING_FOR_EVENT	"Blocked" or "Blocked and Suspended"
STATES_WAITING_FOR_SEMAPHORE	"Blocked" or "Blocked and Suspended"
STATES_WAITING_FOR_MUTEX	"Blocked" or "Blocked and Suspended"
STATES_WAITING_FOR_CONDITION_VARIABLE	"Blocked" or "Blocked and Suspended"
STATES_WAITING_FOR_JOIN_AT_EXIT	"Blocked" or "Blocked and Suspended"
STATES_WAITING_FOR_RPC_REPLY	"Blocked" or "Blocked and Suspended"
STATES_WAITING_FOR_PERIOD	"Timed" or "Timed and Suspended"
STATES_WAITING_FOR_SIGNAL	"Blocked" or "Blocked and Suspended"

Table 4.13: Mapping between RTEMS and Monitoring Tool Task States.

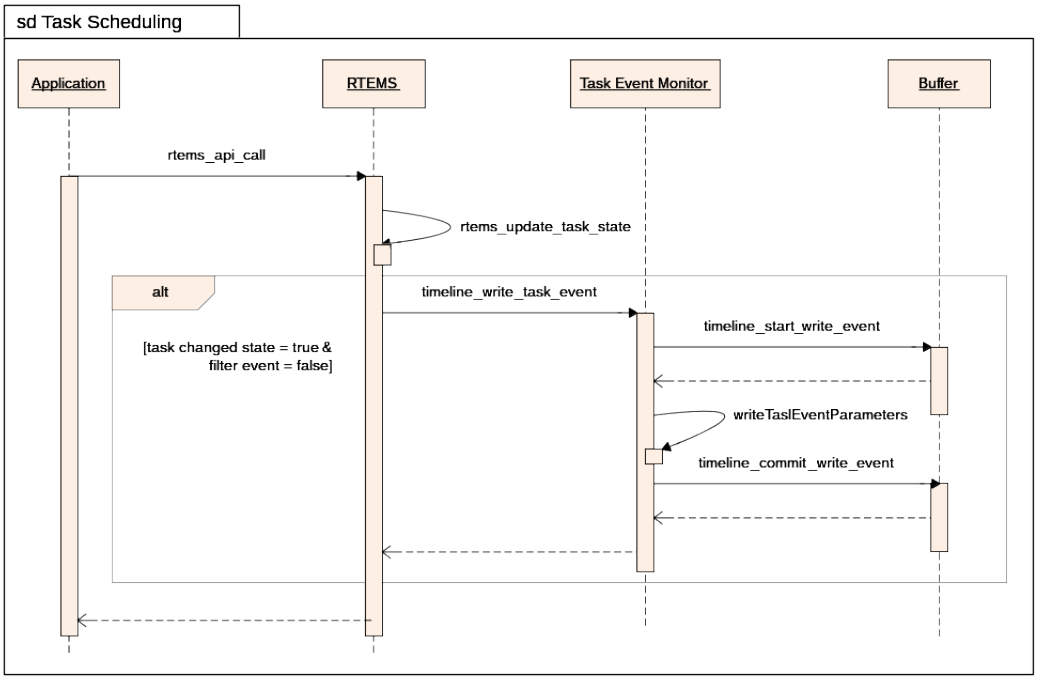

Figure 4.7: Monitoring Tool Task Scheduling Logging Sequence Diagram.

4.4.1.3 RTEMS API Call Logging

The RTEMS API call logging is performed in the context of the calling task. When a RTEMS API call is performed, it calls the RTEMSAPICallMonitor as illustrated in figure 4.8 and records the event of when the call was performed. Similarly, when the function ends, the RTEMSAPICall-Monitor is also addressed to record the event of when the RTEMS API call finished.

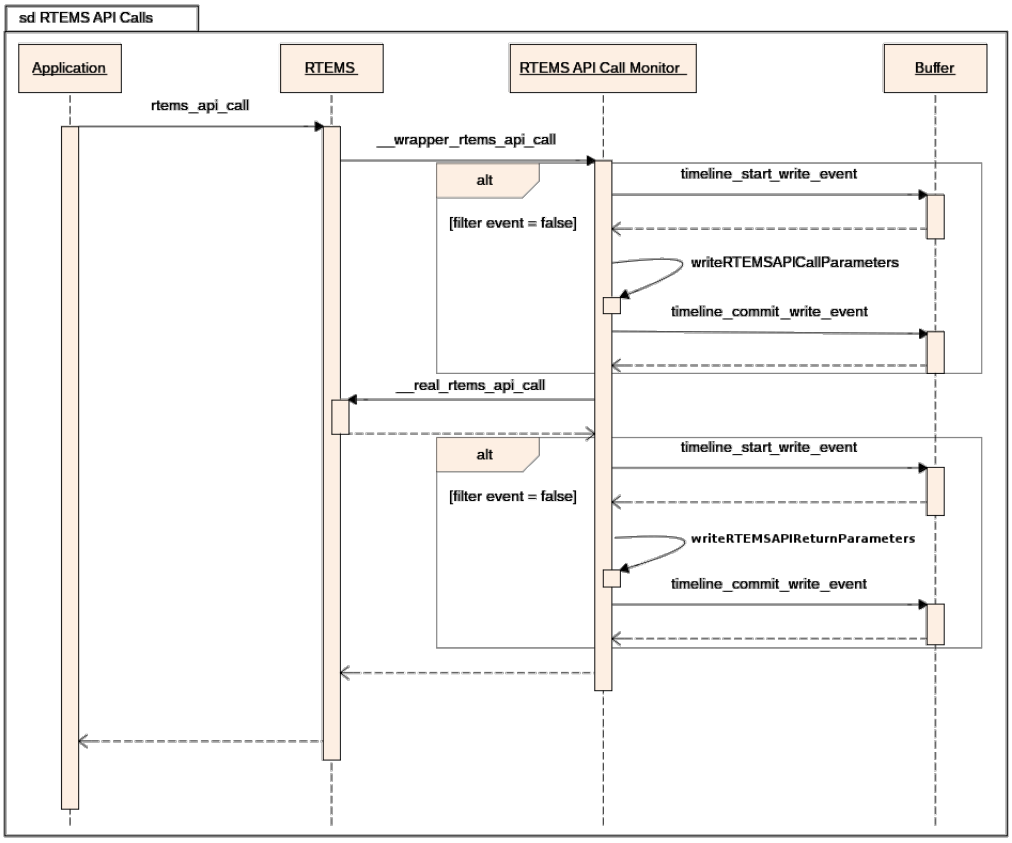

Figure 4.8: Monitoring Tool RTEMS API Call Logging Sequence Diagram.

4.4.1.4 Task Stack Usage Logging

The task stack usage logging is performed during a task context switch as illustrated in figure 4.9. The maximum stack usage of the current task (the task that is going to relinquish the CPU) is determined (starting at the higher or lower stack memory address - depending if the stack grows up or down - and finding the first word that does not match the stack pattern) and logged. Only the maximum stack usage is possible to calculate since the stack pattern is replaced as the stack grows.

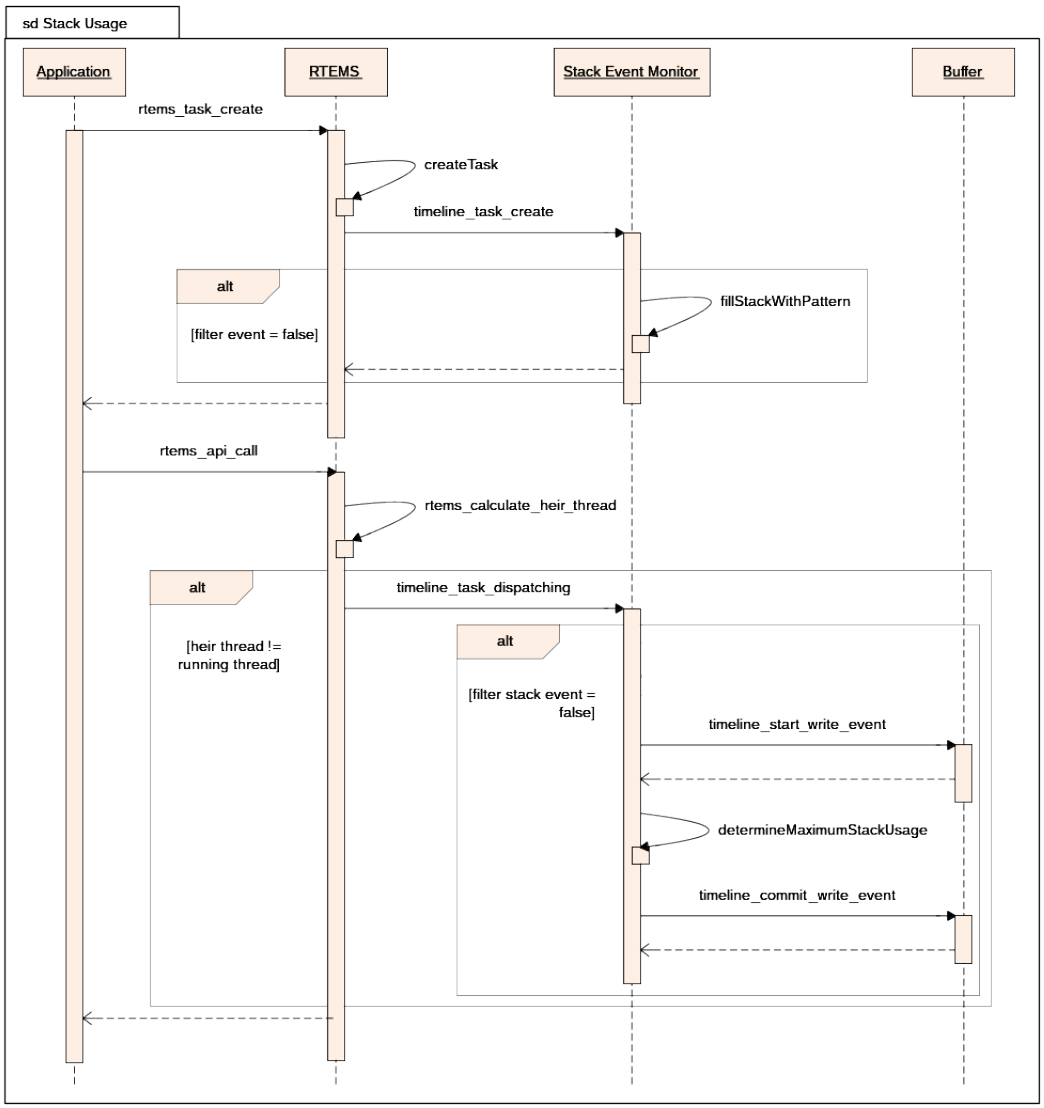

Figure 4.9: Monitoring Tool Stack Usage Logging Sequence Diagram.

4.4.1.5 RTEMS Configuration Logging

The RTEMS configuration is only logged a single time per execution (right after the synchronization message), as the RTEMS configuration does not change during runtime. Configuration includes RTEMS workspace (the memory region used to allocate space for objects created by RTEMS, such as semaphores, tasks, queues, ... [On-15]) start address, size, number of loaded drivers, number of

microseconds per clock tick and number of clock ticks per timeslice (the message format used to store the configuration is shown as event type 6 in figure 4.15).

4.4.2 Log Transmission

The log transmission has two modes: online and offline. In both modes it is also possible for the application to force the transmission of a number of logs at any time. The sequence diagram in figure 4.10 shows the execution flow in case online or offline modes are enabled.

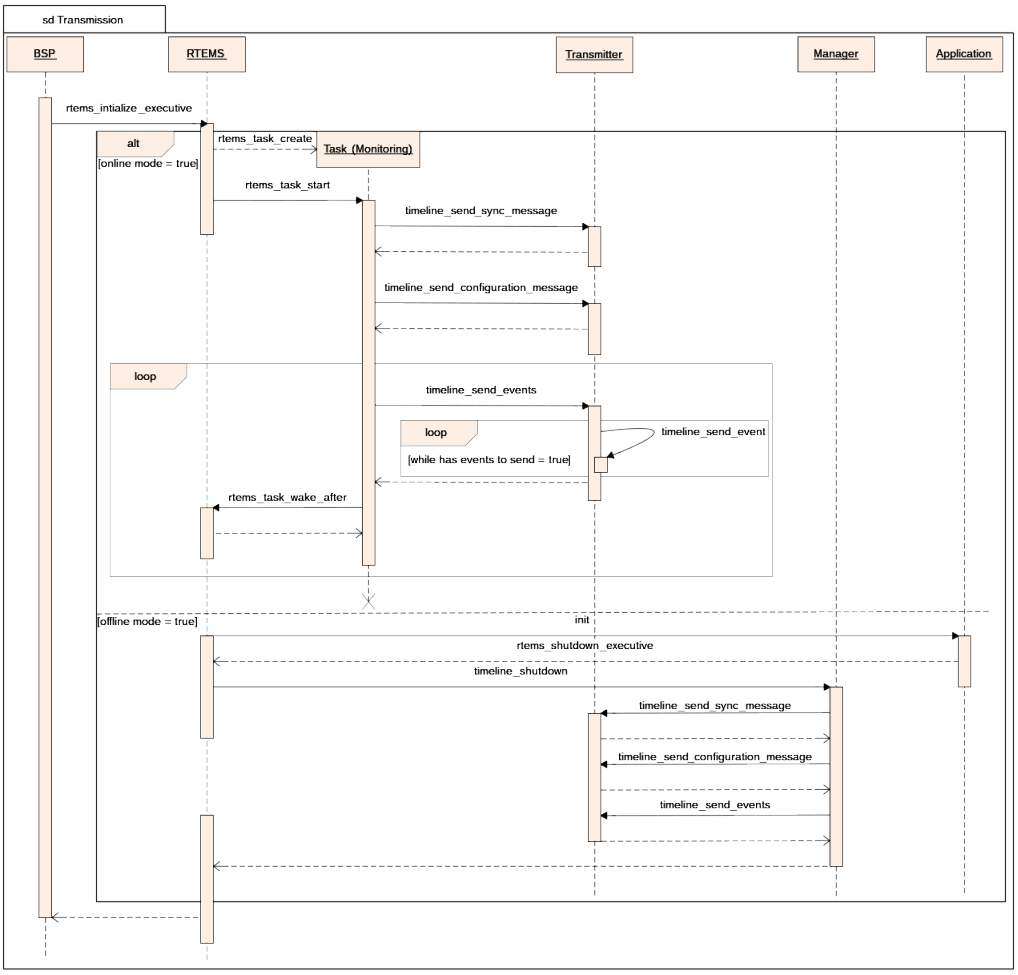

Figure 4.10: Monitoring Tool Event Transmission Sequence Diagram.

In the online mode the tool creates a task that periodically sends the logs through the communication medium. The user can define the priority, period and the total number of logs to send per period of the additional task created and used in online mode. In the offline mode the `rtems_shutdown` primitive transmits all the collected events through the communication medium after disabling further events to be monitored, and prior to calling the real `rtems_shutdown` primitive. On both modes the application can force the transmission of logs by simply calling an available primitive to transmit a target number of logs to the host.

4.4.3 Clock Update

Each event is timestamped with the clock tick and microsecond of when the monitored event occurred. The source for this is a secondary timer initialized before the drivers (i.e.: in the

monitoring stub wrapper for the `rtems_IO_Initialize_all_drivers`), which is reset to zero and reloaded every second, creating a 1 second clock tick. The ticks and microseconds elapsed at any time during execution are updated via the clock update mechanism, which is described in the sequence diagram shown in figure 4.11.

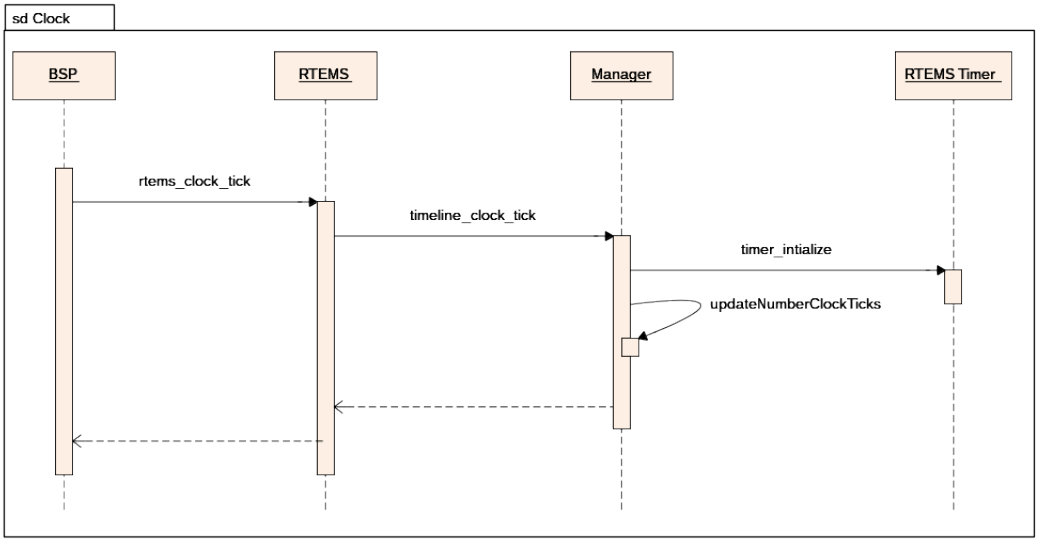

Figure 4.11: Monitoring Tool Clock Update Sequence Diagram.

The clock microseconds reset to zero when a clock tick occurs by calling the `timer_initialize` function. This function is inside the RTEMS Board Support Package (BSP) in the Timer component (not the RTEMS Timer Manager). The Manager component returns the number of microseconds elapsed since the last `timer_initialize` call. Apart from resetting the microseconds, the Manager component also updates its own clock tick tracking (RTEMS keeps the number of elapsed clock ticks in an unsigned integer).

4.5 Dynamic Architecture - RTEMS Monitoring Tool Host

This section describes the dynamic architecture of the Monitoring tool host, and is divided into two main sections that describe the dynamic sequence of the receiver and database components.

4.5.1 Receiver

This subsection describes the receiver component illustrated in figure 4.12.
A synchronization message is sent at the beginning to allow the receiver to synchronize, if the Monitoring stub is configured to operate in online mode, or right before `rtems_executive_shutdown` in offline mode. After the receiver has synchronized it can start receiving useful information. A configuration message is sent after the synchronization message so the system knows some vital information (e.g. microseconds per tick). While the system is in the synchronized state the remaining messages are processed and placed in the Ring Buffer. Regardless of the stub transfer mode the application can always force the transmission of logs at any time during execution through the Monitoring API. The Monitoring API always sends a synchronization message to ensure that the host is ready to receive the information, and tries to send a configuration message. In any case only one configuration message per application execution is sent to the host, right after the first

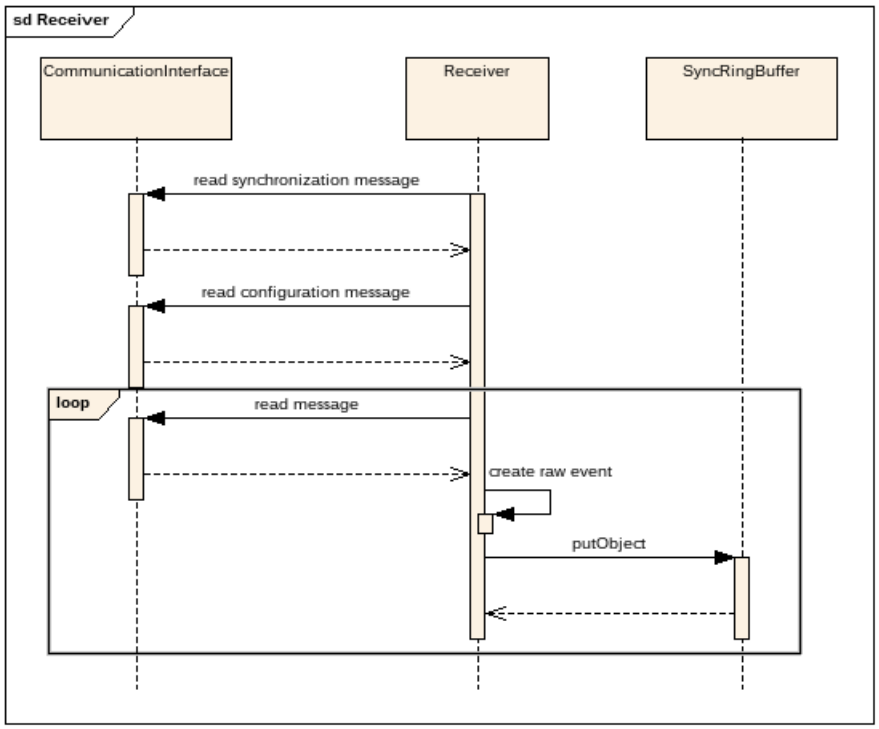

Figure 4.12: Monitoring Tool Receiver Sequence Diagram.

synchronization message.

4.5.2 Database

This subsection describes the database component illustrated in figure 4.13.

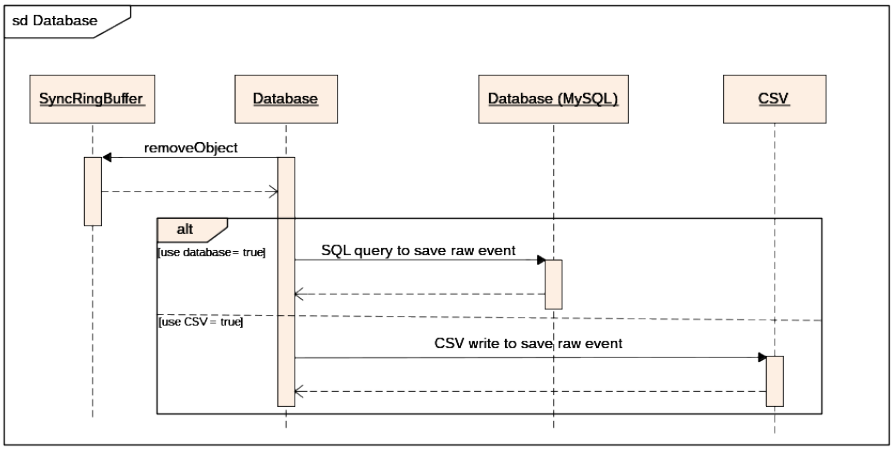

Figure 4.13: Monitoring Tool Database Sequence Diagram.

The Monitoring tool database is continuously waiting for new messages to be placed in the ring buffer. When a new message is placed, the database component either creates a Structured Query Language (SQL) statement to save the message to the database, or writes a new line in the respective CSV file.

4.6 Communication Interface

As shown in section 4.1, the stub needs to transmit the collected information to the host node. This communication is performed through SpaceWire. The packets sent with the collected information vary in size and format between the several events being reported. Figure 4.14 shows the format of a message: it is composed by a header containing the event type and by a body. The event type is represented in with a single byte while the body has a variable length according to the event being transmitted.

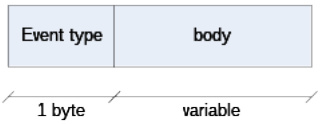

Figure 4.14: Message Format.

Figure 4.15 shows the possible types of events and associated messages reported by the Monitoring tool.

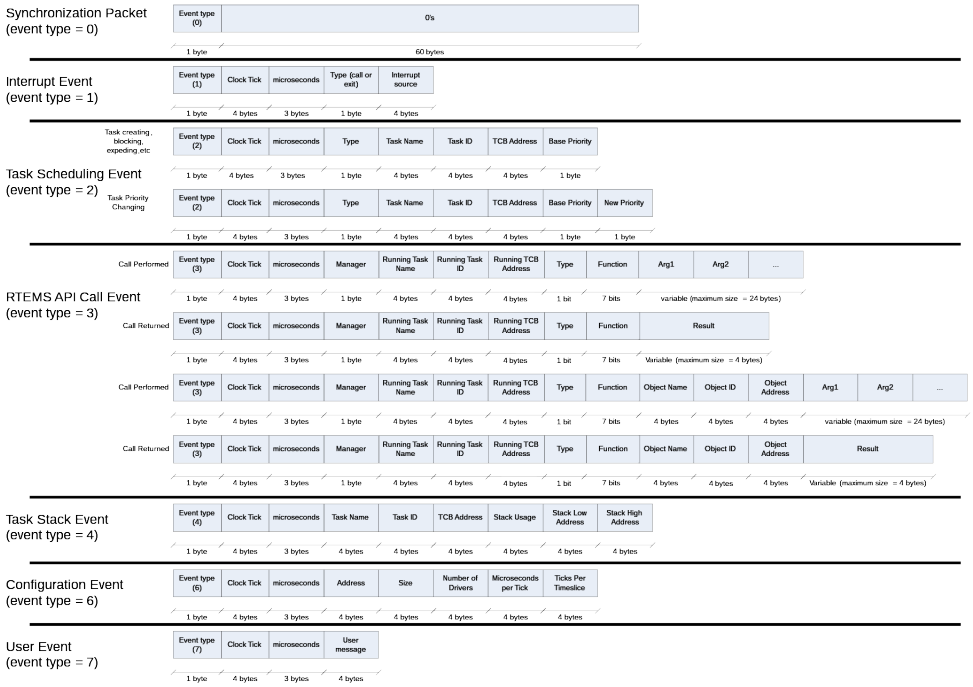

Figure 4.15: Message Formats for Different Types of Events.

The synchronization packet is used to synchronize the transmitter and the receiver during the system initialization phase. Thus, only after the synchronization packet has been received does the host start saving the messages. This packet has the maximum dimension so that it is not possible for another message (or series of messages) to have the same content, as all the bits on the synchronization message are 0 (e.g: the other message types have at least a non-zero event type code at the beginning).

Apart from the synchronization packet, all other messages contain the clock tick and microsecond when the event occurred. Since the microsecond is stored in a 3 byte buffer, the clock tick cannot be greater than 224 microseconds (∼16 seconds), which is a very slow clock tick for the majority of systems. The number of clock ticks elapsed is saved by RTEMS in an unsigned integer (4 bytes)

which allows tracking up to 232 clock ticks. For space applications this is a small number (e.g., for a clock tick of 1ms, the tool can keep track of 49 days before the clock is reset).

The interrupt event message additionally contains a 1 byte field with the type of interrupt event (ISR started or finished) and a 4-byte field with the interrupt source (number of the generated interrupt in the Interrupt Vector Table). The Task Scheduling event message contains the task identification (task name, task RTEMS ID and Task Control Block (TCB) address) and the type of event as described in Table 4.14. If the event is due to a task priority change, an additional field is inserted: new priority.

Number	Type
0	Task creating
1	Task starting
2	Task dispatching
3	Task yielding
4	Task expediting
5	Task waiting
6	Task blocking
7	Task unblocking
8	Task deleting
9	Task resuming
10	Task suspending
11	Task priority changing

Table 4.14: Different Types of Scheduling Events.

The RTEMS API call event message is divided into two types: an event that marks when the function was called and another when the function returned. The type of event is represented in the "type" field (1 bit: 0 for the function call; 1 for the function return). Both types of messages contain a field with the invoked RTEMS Manager, another field with the function, three fields containing the calling task identification (if the call was made inside an ISR then these values are all set to zero) and, for some specific RTEMS Managers (semaphore, message queue, timer, rate monotonic and extension), three fields containing the calling task identification.

The association of manager and function number with the RTEMS API primitive is represented in tables 4.15 and 4.16. The function call event contains the arguments passed by the application to RTEMS. The number of bytes necessary to represent all the arguments in the worst case is 24. The return event is associated with a typically smaller message, since it only sends the return code of the primitive (typically a `rtems_status_code` which is 1-byte long). However, since the `rtems_interrupt_disable`, `rtems_interrupt_enable`,`rtems_interrupt_is_in_progress`, `rtems_interrupt_is_masked`, `rtems_mask_interrupt`, `rtems_unmask_interrupt`, `rtems_interrupt_flash` and `rtems_error_report` are implemented as macros, the tool cannot capture these calls.

The Task Stack event message contains the task identifier and the maximum stack usage of that task (see figure 4.15 for more detail). The Configuration event message contains the address of the RTEMS Workspace, the RTEMS Workspace maximum size, number of drivers, number of microseconds per clock tick and the number of ticks per timeslice. The User event message can be used by the application developer. The application can send four bytes in the message body.

Manager		Primitive	
Number	RTEMS Manager	Number	Primitive
0	RTEMS Task Manager	0	*rtems_task_create*
		1	*rtems_task_ident*
		2	*rtems_task_start*
		3	*rtems_task_restart*
		4	*rtems_task_delete*
		5	*rtems_task_suspend*
		6	*rtems_task_resume*
		7	*rtems_task_is_suspended*
		8	*rtems_task_set_priority*
		9	*rtems_task_mode*
		10	*rtems_task_get_note*
		11	*rtems_task_set_note*
		12	*rtems_task_wake_after*
		13	*rtems_task_wake_when*
		14	*rtems_task_variable_add*
		15	*rtems_task_variable_get*
		16	*rtems_task_variable_delete*
1	RTEMS Interrupt Manager	0	*rtems_interrupt_catch*
2	RTEMS Clock Manager	0	*rtems_clock_set*
		1	*rtems_clock_get*
		2	*rtems_clock_tick*
		3	*rtems_clock_set_nanoseconds_extension*
		4	*rtems_clock_get_uptime*
3	RTEMS Timer Manager	0	*rtems_timer_create*
		1	*rtems_timer_ident*
		2	*rtems_timer_cancel*
		3	*rtems_timer_delete*
		4	*rtems_timer_fire_after*
		5	*rtems_timer_fire_when*
		6	*rtems_timer_reset*
		7	*rtems_timer_initiate_server*
		8	*rtems_timer_server_fire_after*
		9	*rtems_timer_server_fire_when*
4	RTEMS Semaphore Manager	0	*rtems_semaphore_create*
		1	*rtems_semaphore_ident*
		2	*rtems_semaphore_delete*
		3	*rtems_semaphore_obtain*
		4	*rtems_semaphore_release*
		5	*rtems_semaphore_flush*

Table 4.15: Mapping between RTEMS Manager and Primitive Number Assignment (part 1).

Manager		Primitive	
Number	RTEMS Manager	Number	Primitive
5	RTEMS Message Queue Manager	0	*rtems_message_queue_create*
		1	*rtems_message_queue_ident*
		2	*rtems_message_queue_delete*
		3	*rtems_message_queue_send*
		4	*rtems_message_queue_urgent*
		5	*rtems_message_queue_broadcast*
		6	*rtems_message_queue_receive*
		7	*rtems_message_queue_get_number_pending*
		8	*rtems_message_queue_flush*
6	RTEMS Event Manager	0	*rtems_event_send*
		1	*rtems_event_receive*
7	RTEMS I/O Manager	0	*rtems_io_register_driver*
		1	*rtems_io_initialize*
		2	*rtems_io_open*
		3	*rtems_io_close*
		4	*rtems_io_read*
		5	*rtems_io_write*
		6	*rtems_io_control*
8	RTEMS Fatal Error Manager	0	*rtems_fatal_error_occurred*
		1	*rtems_error_get_latest_non_fatal_by_offset*
		2	*rtems_error_get_latest_fatal_by_offset*
9	RTEMS Rate Monotonic Manager	0	*rtems_rate_monotonic_create*
		1	*rtems_rate_monotonic_ident*
		2	*rtems_rate_monotonic_cancel*
		3	*rtems_rate_monotonic_delete*
		4	*rtems_rate_monotonic_period*
		5	*rtems_rate_monotonic_get_status*
		6	*rtems_rate_monotonic_deadline*
		7	*rtems_rate_monotonic_get_deadline_state*
		8	*rtems_rate_monotonic_execution_time*
10	RTEMS Extension Manager	0	*rtems_extension_create*
		1	*rtems_extension_ident*
		2	*rtems_extension_delete*

Table 4.16: Mapping between RTEMS Manager and Primitive Number Assignment (part 2).

4.7 Conclusions

This chapter presented the monitoring tool design, based on the requirements presented in the previous chapter (chapter 3), outlining the monitoring tool architecture and describing its implementation. As a result, the next chapter will present the monitoring tool testsuite.

Chapter 5

Monitoring Tool Testsuite

This chapter presents the testsuite used to test the developed monitoring tool. The tests are divided into two categories:

- General Purpose Tests - test general functionality of the monitoring tool (i.e.: all the requirements except performance);
- Performance Tests - test the performance requirements of the monitoring tool (refer to section 3.5.2).

5.1 Testing

5.1.1 Test Platform Configuration

This section describes how the target and host platforms that compose the Monitoring tool must be configured in order to run the validation tests, as well as the platform that will build the test application and monitoring stub to run in the target platform.

Target Platform

The considered Target platform was the Gaisler-Research GR712RC Board, which must be running the test application together with the monitoring stub. It should be connected to the Host platform through a SpaceWire link connected to core 0 of GR712.

Host Platform

The considered Host platform was the EGSE-SCOC3 computer, which features the iSAFT RTE web services. In addition, the Host platform must satisfy the following requirements:

- Java Java Runtime Environment (JRE) 1.8;
- MySQL Server 5.6.

The iSAFT RTE service must be running, and the host should be connected to the Target platform through a SpaceWire link connected to the `SpWPort0` of EGSE-SCOC3.

Build Platform

The build platform is responsible for compiling RTEMS, the RTEMS application and monitoring stub. A desktop computer running Linux (Debian Lenny) was used. The communication between the build platform and the target platform is done using GRMON. Refer to section A.4 for how to execute an application on the target platform.

5.2 Validation Test Definition

This section contains the description of the tests used to generate the validation test reports. The tests have been identified with the label `val_XX_YYYZZ`, which stands for:

- val - Validation Tests;
- `XX` - Two digit number to identify the type of test;
- `YYY` - Three digit number to identify the test;
- `ZZ` - Two digit number to identify the subtest.

The `XX` identifier can be divided into the following categories:

- 10 - General Purpose;
- 20 - Performance.

The `YYY` and `ZZ` identifiers are divided into different categories, depending on the `XX` identifier. The combination of the `XX`, `YYY` and `ZZ` parameters shall define a unique identifier for each validation test, that is, two validation tests cannot have the same `XX`, `YYY` and `ZZ` parameters. According to the requirement `MT-SR-TEST-0010`, the test name shall follow the naming rule: The test name of the each executable is unique and follow the naming rule `mne_tt_nnnaa.ext`, where:

- `mne` corresponds to the mnemonic code of the test;
- `tt` corresponds to the type of test. The `tt` component corresponds to the `XX` number defined above;
- `nnn` corresponds to test number. The `nnn` component corresponds to the `YYY` number defined above;
- `aa` corresponds to the subtest number. The `aa` component corresponds to the `ZZ` number defined above;
- `ext` corresponds to the file extension (e.g. “exe”).

5.2.1 General Purpose Tests

The General Purpose Tests test all but the performance requirements of the Monitoring Tool. To generate a single test report four applications must be used, namely:

- `MAX_APP` - the test application which execution will be traced by the monitoring tool. It performs all the calls that the monitoring tool is capable of tracing;
- Monitoring Stub - the monitoring stub which will record the test application execution. It should be linked with the `MAX_APP` test application;
- Monitoring Host - the monitoring host which will receive the trace data from the stub and save it either on CSV files or on a MySQL database;
- `LOG_TEST` - tests the trace data saved by the monitoring host to ensure that the data received is valid (i.e.: the trace of a function has the expected number of fields, and that each field is within the valid values or byte range). It tests trace data saved either on CSV files or on a MySQL database.

When the monitoring stub is linked with the MAX_APP application it should be executed on the target platform, and at the same time the monitoring host should be running on the host platform. When the MAX_APP has ended its execution and the monitoring host has received all the trace data, that data is then used as input for the LOG_TEST which will test each trace line and generate a complete test report. The Monitoring Testsuite uses three Monitoring Stub configurations, each of them being an individual validation test. Each test should be run two times, such that the monitoring host can record the trace data in both CSV and database storage mediums. A test case is only complete when the monitoring host stores the received trace data, and the test report is only generated when the trace data is given to the LOG_TEST application.

5.2.1.1 Test Case Specification

The General Purpose Tests have been identified with the label val_10_YYYZZ, which stands for:

- val - Validation Tests;
- YYY - Three digit number to identify the test;
- ZZ - Two digit number to identify the subtest.

The YYY identifier can be divided into the following categories:

- 010 - Monitoring Stub in offline mode with a priority filter for task schedulability events between 70 and 110;
- 020 - Monitoring Stub in offline mode with filters for RTEMS API, task scheduling and task stack events;
- 030 - Monitoring Stub in online mode without filters.

The ZZ identifier can be divided in the following categories:

- 10 - Monitoring Host recording trace data in CSV files;
- 20 - Monitoring Host recording trace data in MySQL database.

5.2.1.2 Executing the Tests

Each test has to be executed manually. The procedure to compile an application along with a monitoring stub is described in section A.3.
The application used in the Monitoring Testsuite is always the MAX_APP. This application is monitored by the monitoring stub with the configuration defined for each test case. After compiling the MAX_APP application, enter on one of the validation test directory and proceed as stated in section A.1.6.
In the host platform the monitoring host must be running RTEMSMonitoringHost.jar (found in the directory Host/dist), configured to store the trace data in CSV files for the subtests number 10 or in the MySQL database for the subtests number 20. To generate the test reports see section 5.2.1.3. Since the test reports are too long (e.g.: the reports for val_10_01010 and val_10_03020 are over 340000 lines), appendix B presents a snippet of the trace data and the resulting test report for the val_10_01010 test.

5.2.1.3 Test Report Generation

This section describes the steps required to generate a test report from the execution of a test case. After the execution of a test case the resulting output is either a directory with CSV files or a new MySQL database containing the trace data of the monitored application. The `LOG_TEST` application (found in the directory `LOG_TEST/dist`) verifies each trace line and generates a test report evaluating if the data received and stored is valid for the given test case.
The command in listing 5.1 should be run to generate the test report for a given test case (the `<configuration>` field should be replaced with the desired configuration, described below).

```
java -jar LOG_TEST.jar <configuration>
```

Listing 5.1: Starting the `LOG_TEST` application to generate the general purpose test reports.

When launching the test report generator it should be stated the trace data source type (CSV or database), the directory where the report should be written to and the test case to which the trace data corresponds. If the trace data source is a database then the database Uniform Resource Locator (URL), user name and password must also be given, otherwise if the trace data source is CSV then its directory should be provided. Configuration of the test report generator is done by supplying one following arguments in the command line, replacing the fields between `< >` with the corresponding value:

- Using MySQL database:
  - `-db <url> <username> <password> <test_report_dir> <test_case_name>`
- Using CSV files:
  - `-csv <csv_directory> <test_report_dir> <test_case_name>`

Where the `test_report_dir` is the directory where the test report will be saved, and `test_case_name` is the name of the test case which corresponds to the given trace data. In the case of the database the URL field must point to the database containing the trace data of the test case given in `test_case_name`. Similarly if using CSV files the `csv_directory` must be the path to a directory containing the CSV files which resulted from the execution of the given test case in `test_case_name`.
Note that when testing the trace data from CSV files the subtest number is always 10, and when testing trace data from the MySQL database the subtest number is always 20.

5.2.2 Performance Tests

The Performance Tests test the performance requirements of the Monitoring Tool. These tests measure the monitoring stub performance while saving events on memory and while sending events to the monitoring host through SpaceWire.

A test scenario was devised to replicate a sample application that may be subject to monitoring. The considered scenario is as follows: a task acquires a semaphore via `rtems_semaphore_obtain`, and waits for an event with `rtems_event_receive` causing a context switch and a stack save. This scenario is defined to take 10 milliseconds.

The performance tests call directly the functions on the monitoring stub which save the respective events on memory and that send the events to the monitoring host directly. The actual functions

on RTEMS are not called or considered. To measure the time spent to save or send an event, each is saved or sent 1000 times such that the final time is the average of all the iterations. The time is measured with the leon3 timer 0.
To generate a single test report three applications must be used, namely:

- `DUMMY_APP` - dummy application with a single task containing only a call to `rtems_shutdown_executive`. This application exists only so the monitoring stub can be compiled;
- Monitoring Stub - the monitoring stub which will be linked with the `DUMMY_APP`. In this case the monitoring stub will not execute, but must be compiled as the performance tests will call directives on the Stub. This monitoring stub is in the `DUMMY_APP` directory;
- A performance test.

The `DUMMY_APP` and the monitoring stub in the same directory should be compiled as described in the procedure to compile an application along with a monitoring stub in section A.3, and only then can the performance tests (`val_20_01010` and `val_20_02010`) be compiled and executed.

5.2.2.1 Test Case Specification

The Performance Tests have been identified with the label `val_20_YYY10`, which stands for:

- val - Validation Tests;
- `YYY` - Three digit number to identify the test.

The `YYY` identifier can be divided into the following categories:

- 010 - Performance test to measure the time spent saving a monitored event;
- 020 - Performance test to measure the time spent sending a monitored event via SpaceWire.

5.2.2.2 Executing the Tests

Each test has to be executed manually. The `DUMMY_APP` directory contains both the dummy application and the monitoring stub. Once these have been compiled the performance tests can be compiled and executed.
To compile the performance tests enter on one of the validation test directories (`val_20_01010` or `val_20_02010`) and run the `make` command.
The test is now compiled, and can be loaded and executed on the GR712RC Board by following the instructions in section 5.1.1.
In the host platform the monitoring host must be running `RTEMSMonitoringHost.jar` (found in the directory `Host/dist`). In this case it does not matter if the trace data is saved on CSV or on the database, as the host is only required to activate the SpaceWire link. Since the performance tests never send a synchronization message to the monitoring host, all messages sent by the tests will be ignored. To generate the test reports see sections 5.2.1.3 and 5.2.2.3.

5.2.2.3 Test Report Generation

The test report is printed on the GRMON console, after the test has been loaded and executed as described in section 5.1.1. The test report for the performance tests can be seen in appendix C.

5.3 Conclusions

The tests show that the monitoring tool is able to correctly capture the execution of a given application, and transfer that information to a monitoring host for storage and later analysis. The requirements specified in chapter 3 were validated, and all tests have passed. Note that not all requirements can be tested with software, meaning that they have to be manually validated. This is the case of requirements `MT-SR-TEST-0010`, `MT-SR-TEST-0020`, `MT-SR-TEST-0030`, `MT-SR-DITC-0020`, `MT-SR-DITC-0050`, `MT-SR-DITC-0080`, `MT-SR-DITC-0110` and `MT-SR-VAL-0010`.

The source code for the developed Monitoring Tool and the full test reports can be found on the Compact Disc (CD) delivered with this dissertation.

The next chapter will describe how this monitoring tool can work in a SMP configuration, and what was tried to accomplish it.

Chapter 6

Multicore

This chapter gives a brief presentation of the challenges presented by a multicore system, how it is implemented in RTEMS and how the monitoring tool has been changed to trace an application execution running in a SMP configuration.

6.1 Multicore in Space

Following the increase in popularity of multicore systems in Real-Time Embedded Systems, the European Space Agency has been analyzing the possibilities and problems of multicore systems in space applications. Cobham-Gaisler develops the processors and boards used by ESA on their space missions, and is currently developing a quad-core LEON4 processor to add to the current dual-core LEON3 GR712RC Board as a space ready (radiation tolerant) multicore platform.

The benefits that the space sector may derive from multicore systems have been analyzed in the report of a recent (2011) activity [MP11]. It reports benefits for scientific payload data processing applications, such as image processing and compression, which leads to savings in telemetry downlink operations, as it reduces the amount of time spent transferring data to earth. It is also noted that by increasing the number of cores it is possible to reduce the CPU frequencies, leading to lower power consumption.

6.2 Multicore Challenges

Updating an existing single-core system for multi core raises some synchronization concerns. A single-core OS can easily achieve task-level mutual exclusion by setting the running task as non-preemtible (since only one task can execute at a given time, and that task can not be preempted by any another task regardless of priority, the task is guaranteed to execute in mutual exclusion concerning other tasks - it still could be preempted by an ISR), however in a multicore setting this is no longer the case, as tasks running on other cores can execute without having to preempt this task.

As for interrupt handling, in a single-core system an ISR is guaranteed to execute in system-wide mutual exclusion (unless the OS allows nested interrupt handling). On a multicore environment a single ISR executes only on a single core, meaning that an ISR may execute concurrently with a task or even another ISR running on another core.

Another issue with multicore systems is cache coherency among private caches. Each individual core usually has one or two private (L1) caches, for instructions and data. These caches have to be coherent among all the system's cores, otherwise there is the risk of one CPU using an old value already updated by another core. Some hardware architectures include one or more shared cache levels (L2, L3, ...).

Coherence is obtained through cache snooping, where the CPUs (namely their cache controller) listens (or sniffs) the system bus for information on private cache updates. When a cached value is updated (marked as "dirty"), the cache controller invalidates that address on all the other CPU caches. This means that when another CPU tries to access that address on their cache it will result on a read miss, so the cache controller will send a request on the system bus for that address. Every cache controller check their cache and if they have a value marked as dirty on that address they update its state to "valid" and sends a copy to the requesting CPU. Cache coherence is critical but can have significant impact on the system bus bandwidth, which may be accentuated by certain execution patterns.

6.3 RTEMS SMP implementation

This section gives a brief presentation on the SMP implementation in RTEMS 4.11.

6.3.1 Source Code Separation

All the SMP specific code in RTEMS is encapsulated within `RTEMS_SMP` define guards, which are only active if the flag `-enable-experimental-smp` is used when building RTEMS.

6.3.2 SMP Applications

SMP-enabled applications must also configure the RTEMS configuration table with at least the options shown in listing 6.1, where `<MAXIMUM_PROCESSORS>` must be replaced with the maximum number of cores that the application wishes to use (note that the hardware or the BSP may not be able to provide the requested amount of processors).

```
#define CONFIGURE_SMP_APPLICATION
#define CONFIGURE_SMP <MAXIMUM_PROCESSORS>
```

Listing 6.1: RTEMS SMP configuration table options for applications.

6.3.3 Interrupt Processing

In a SMP configuration an interrupt may be broadcasted to all cores, or mapped to a single core (interrupt affinity).

Since every core will handle interrupts, the interrupt handling routines that exist on RTEMS, such as `rtems_interrupt_disable` and `rtems_interrupt_enable`, have been replaced (if used in a SMP setting) with a local version: `rtems_interrupt_local_disable` and `rtems_interrupt_local_enable`, so only interrupts on the core executing the function are disabled/enabled.

6.3.4 Multicore Scheduling

There are three main multicore scheduling approaches:

- Partitioned Scheduling - Each task executes on a fixed processor (no task migration);
- Semi-partitioned Scheduling - Each task (or task instance) executes on a fixed processor, but may migrate (whole task, of just an instance) to another;
- Global Scheduling - Any task can execute and migrate to any processor .

The first two approaches use static task assignment, while global scheduling feature dynamic task assignment.

6.3.5 RTEMS SMP Scheduling

RTEMS currently features three priority based SMP schedulers [On-15]:

- Simple SMP;
- Priority SMP;
- Priority Affinity SMP.

Note that all RTEMS SMP schedulers ignore the task preemption model (i.e.: a task can not be configured as non-preemptible).

6.3.5.1 Simple SMP Scheduler

The Simple SMP scheduler is a non-deterministic algorithm (an implementation of Global Job-Level Fixed Priority scheduler) which favours replacing threads that are preemptible and have executed the longest when given the choice to replace one of two threads of equal priority on different cores.

6.3.5.2 Priority SMP Scheduler

The Priority SMP scheduler is a deterministic algorithm based on the Global Fixed Task-Priority Pre-emptive scheduler. It uses one ready chain per priority level to ensure constant time insert operations.

6.3.5.3 Priority Affinity SMP Scheduler

This scheduler is an extension of the Priority SMP Scheduler (refer to 6.3.5.2) that adds thread to core affinity support. Affinity is achieved by using CPU sets (same as in the Linux Kernel), which are 32-bit bitmasks where a bit set means that the corresponding CPU is part of the set. Each thread/task has a CPU set, which if used for affinity defines the CPU cores where that task can execute.

6.3.6 CPU Core Data and Communication

Due to the sequential nature of the basic system initialization one processor has a special role: the processor executing the `boot_card()` function, called the boot processor. All other processors are called secondary.

Each CPU core in RTEMS is assigned a data structure (`Per_CPU_Control`) to hold each core current state. This structure is used independently of the number of cores, but in a SMP setting the structure has additional fields to state if the core has been started (at start up only one core is active, the others have to be powered-up by RTEMS), to retrieve the scheduler context (i.e.: how many cores the scheduler instance owns) and even a message field (32-bit unsigned integer) that can be used for inter-CPU synchronization.
Since this data structure is in shared memory, visible to all cores, a given core can atomically store (using the C11 standard atomic operations) a message on the receiving core data structure, and

notify the core that a message was sent by forcing an external interrupt (currently IRQ 14 for the sparc BSP) on the interrupt controller. Besides sending messages, a core can also notify another core for it to run its thread dispatcher, which may be the case when a thread migrates between cores (so the thread's new CPU can evaluate if it can execute right away).

Upon the reception of an Inter-Processor Interrupt (IPI) interrupt an ISR is called which

6.3.7 Resource sharing among cores - MrsP

MrsP is a lock-based resource control protocol [BW13]. It is a generalization of the priority ceiling protocol featuring:

- FIFO ordering;
- busy wait at ceiling priority;
- helping mechanism.

Each core receives a local ceiling for each global resource, while each global resource is given a FIFO list. If a task requests a given resource it will spin (busy wait with a spinlock) at that resource's local ceiling, and in the event that other tasks also require access to the resource then they will also still at their local ceiling and their requests are handled in a FIFO order.

It also uses a "helping protocol". The general idea behind a helping protocol is that a task waiting for a resource will execute the resource's critical section on behalf of another waiting task. If a task holding a resource is preempted by a local higher priority task, the task holding the resource can migrate to a CPU containing a task waiting for that same resource. When migrated the task holding the resource will have a higher priority than the task waiting for the resource.
RTEMS uses a dynamic list for the FIFO, where each task is assigned a node. Each scheduler node has three thread pointers:

- owner - the task owning the node;
- user - if the task owning the node helps a task, this pointer will point to the helped task;
- idle - active owners will lend their own node to an idle thread in case they execute currently using another node or in case they perform a blocking operation;
- accepts_help - if the owner task is helping another task (help is always for tasks running on another core), then this pointer will point to the task that will be running on the owner task CPU while it is providing help.

Currently the only resource managed by MrsP is semaphores, named MrsP semaphores.

6.4 Monitoring Tool in Multicore

This section describes how the developed monitoring tool was changed to work with RTEMS 4.11 in SMP configuration.

6.4.1 The Target Platform

Since Edisoft had to return the GR712 board to ESA, the multicore part of the monitoring tool had to be based on another target. Since there was no suitable multicore hardware available to use with RTEMS SMP (one option was the Raspberry Pi 2, but the SMP support for it is still a work in progress), the `realview-pbx-a9-qemu` BSP was chosen, as it is a BSP that runs on the QEMU emulator, and its SMP support in RTEMS is already somewhat mature (it was one of the first SMP BSPs). The Realview PBX A9 board is an ARM reference board Field-Programmable Gate Array (FPGA), with a A9 dual-core processor. The `-qemu` sufix on the BSP name means that this particular BSP is configured for the emulated version of this board provided by QEMU.

There was, however, the need to add a few missing peripherals to the this BSP:

- support for a secondary timer, to be used by the monitoring tool;
- support for a secondary UART, for the transmission of trace data from the target to the host platform (refer to section 6.4.2).

In addition to this new code, the RTEMS configuration table definition was changed to remove the `const` qualifier, so the monitoring tool could access and update it with its own resource requirements.

6.4.2 Communication Channel

Since the new target platform had no SpaceWire connectivity, a new communication channel was required. The choice was to use a RS-232 serial link to connect the target platform with the monitoring host, through QEMU (QEMU acting here as an intermediary node between the two platforms).
Since the monitoring host was written in Java, the RXTX library [Lib] was chosen to handle the serial communications.

As for the target, the Termios (the UNIX API for terminal I/O operations) implementation in RTEMS 4.11 was updated to remove the `ONLCR` flag, which precedes every byte that has the value 10 (which translates to the character `'\n'` - newline - in the American Standard Code for Information Interchange (ASCII) table) with a new byte with the value 13 (the character `'\r'` - carriage return), and the `OPOST` flag, which enables text output post-processing. Since the monitoring tool transmits binary data, these transformations alter the data being sent and had to be removed.

6.4.3 Changes to the Monitoring Stub

As RTEMS 4.11 is a number of versions more recent than RTEMS 4.8 (Edisoft's RTEMS Improvement is a fork of RTEMS 4.8), there were a number of changes that the monitoring tool had to take into account, apart from the SMP specific code.
As for the multicore changes, only two changes were needed to the monitoring tool stub so it could monitor events in a SMP configuration:

- Event buffer - a single event buffer is not suitable in a SMP configuration, as it would create a bottleneck since every core would have to wait for their turn to store their events;

- Event message data - because a SMP environment implies two or more CPU cores, each event has to be tagged with the CPU core where it occurred to properly identify its source.

Having one event buffer per core instead of a global buffer works best, as each core does not have to wait for the other cores to store its events (therefore avoiding synchronization issues). However, this does not ensure the chronological order of events, as each buffer will store events on their own pace, depending on the tasks they are executing. Do note that even if a global buffer were to be used the chronological order would not be ensured, as tasks with higher priority would get to write their events on the buffer faster.

Having multiple buffers also does not affect the memory requirements: if each core is assigned a ring buffer with 1 MB, then a global buffer servicing n cores would require n MB.

The bottleneck is now the event transmission, as there is only one interface out of the system. The way the monitoring tool deals with this depends on the transmission mode being used:

- online mode - if e is the number of events to send on each period, and c is the number of cores on the system, then $\frac{e}{c}$ events are sent from each core buffer until the total e events have been sent. For this to properly work e has to be equal or greater than c otherwise the events on the buffer of the last core(s) would only be sent at the end;
- offline mode - each core's buffer is emptied, one at a time (i.e.: all events in core 1 buffer are sent, then all events on core 2 buffer are sent, and so on).

As for the event message data, a single byte-field was added to every event message after the microsecond field (refer to figure 4.14) containing the CPU core index where the event has occurred.

6.4.4 Changes to the Monitoring Host

The only change required on the monitoring host was to accommodate the new `core` byte-field in every event message that the stub now produces, due to the update in the message protocol (as described in section 6.4.3).

6.4.5 Implementation

As detailed in section 6.4.1, QEMU was used to emulate the Realview PBX A9 platform. Listing 6.2 presents how an SMP application can be executed on QEMU, where `-serial mon:/dev/pts/4` may be replaced by any other pseudo-terminal where the application standard output can be shown, and `-serial /dev/ttyD0` is the secondary serial port where the monitoring stub will output the execution data.

```
qemu-system-arm -no-reboot -net none -nographic -M realview-pbx-a9 -m 256M -
    smp 2 -kernel o-optimize/mtApp.exe -serial mon:/dev/pts/4 -serial /dev/
    ttyD0
```

Listing 6.2: Running SMP application on QEMU.

Since the monitoring host requires a connection to be open at all times, and QEMU only keeps the serial ports up while it is running, there was a need for a way to have a serial port always up on the system. The solution found was to create a loopback between two pseudo-terminals, created with the `socat` utility (refer to listing 6.3). This creates two pseudo-terminals . As a pseudo-terminal

does not depend on an actual hardware connection, the monitoring host can be waiting for data on that port indefinitely. While QEMU executes the application, the monitoring stub feeds that port with the execution data which can then be processed by the monitoring host.

```
socat -d -d pty,raw,echo=0,onlcr=0 pty,raw,echo=0,onlcr=0
```

Listing 6.3: Looback between two pseudo-terminals.

The monitoring host is configured to receive data on the `/dev/ttyD1` port, so one of the pseudo-terminals created by `socat` must match that name. This was done for both pseudo-terminals using symbolic links, as described in listing 6.4, where `xx` and `yy` must be replaced with the pseudo-terminal numbers returned by `socat`.

```
ln -s /dev/pts/xx /dev/ttyD0
ln -s /dev/pts/yy /dev/ttyD1
```

Listing 6.4: Creating a symbolic link to the pseudo-terminals.

Even though the changes required on the monitoring tool to operate in a SMP configuration are small, there were transmission problems between the monitoring stub and the host and as such the multicore version of the tool could not be properly tested, as data would be lost in the transmission. Replacing the SpaceWire communication with RS-232 (as described in section 6.4.2), and placing QEMU between the stub and the host added extra layers of unexpected complexity. The QEMU source code had to be changed, as it hardcodes the termios flags used by any serial ports given to it (by default it treats all serial communication as text/lines) and sets a hardcoded baudrate, replacing any previous configuration of the ports. Because it does all of this silently, it was tricky to detect what was happening. In the end there was not enough development time to resolve the problem.

6.5 Conclusions

Initial tests with the extension of the tool for multicore systems show that it is able to monitor the execution of an application, and tracing back the events to the CPU core where they have occurred.

The SMP version of the tool is in the `MON_TOOL_SMP` directory in the delivered CD, as well as the patches for RTEMS 4.11 and QEMU source-codes. The tool configuration and compilation is the same as described in appendix A.

The next chapter presents the conclusions and future work to be done on the developed monitoring tool.

Chapter 7

Conclusions and Future Work

In this dissertation a state of the art in monitoring and software execution tracing, a set of requirements and an architecture for a monitoring tool were presented, as well as an implementation for both single and multicore systems in both sparc and arm architectures.

The tests developed and performed show that the monitoring tool is able to correctly capture the execution of a given application (with varying degrees of verbosity), and transfer that information to a monitoring host for storage and later analysis, while also validating the requirements presented in chapter 3. Initial results from the multicore version show that it is also viable for such systems.

The results of this work are relevant for future improvements of the developed monitoring tool at Edisoft, in the context of future projects with ESA.

7.1 Future Work

The developed monitoring tool can obviously be improved and extended. One possible improvement might be to monitor only a specific event instance, instead of all events of a certain type (e.g.: only monitor the stack usage of a specific task, instead of monitoring the stack usage of all tasks on the system).

A graphical display tool for the execution trace data based on tracecompass or tracealizer would be a great next step, as it would be much easier to view and interpret the data that the monitoring tool outputs.

Regarding the tool usage, currently the monitoring tool is composed of two distinct components (the host and the stub) that are configured and operated by the user as two independent applications. A more refined version of the monitoring tool could have the host machine compiling the monitored application, configuring and linking the stub to the application, sending the executable to the target machine and capturing the execution events. Ideally the monitoring tool graphical timeline could even be integrated within an IDE for debugging convenience.

As for the event capture, which is currently done by wrapping the calls at linkage time, the wrappers could be automatically generated based on debugging data (e.g.: DWARF debugging data) to retrieve function signatures, instead of manually defining the wrapper. This could allow an easier port of the tool across RTEMS versions, and easily monitor more API functions, BSP specific features or even application defined functions.

In terms of architecture, it would be nice to include in the tool support for hardware tracers (such as the Debug Support Unit (DSU) included in the Gaisler's GR712 board), to attest the benefit (or lack of) an hybrid monitoring architecture.

Bibliography

[Agea] European Space Agency. Mil-std-1553. http://www.esa.int/Our_Activities/Space_Engineering_Technology/Onboard_Computer_and_Data_Handling/Mil-STD-1553. Last access 2 of October 2016. 1

[Ageb] European Space Agency. Spacewire. http://spacewire.esa.int/. Last access 2 of October 2016. 1, 19

[BW13] A. Burns and A. J. Wellings. A schedulability compatible multiprocessor resource sharing protocol – mrsp. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 282–291, July 2013. 68

[Cora] OAR Corporation. Rtems tracing. https://devel.rtems.org/wiki/Developer/Tracing. Last access 2 of October 2016. 15

[Corb] Oracle Corporation. Java se runtime environment. http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html. Last access 2 of October 2016. 77

[Corc] Oracle Corporation. Mysql community server. http://dev.mysql.com/downloads/mysql/. Last access 2 of October 2016. 77

[DGR04] N. Delgado, A.Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *Software Engineering, IEEE Transactions on*, 30(12):859–872, Dec 2004. 5, 6, 8

[Eff] EfficiOS. Common trace format (ctf). https://www.efficios.com/ctf. Last access 2 of October 2016. 9

[Hila] Green Hills. Processor probes. http://www.ghs.com/products/debugdevices.html. Last access 2 of October 2016. vii, 15

[Hilb] Green Hills. Time machine. http://www.ghs.com/products/timemachine.html. Last access 2 of October 2016. 15

[Hil05] Nat Hillary. Measuring performance for real-time systems. *Freescale Semiconductor, November*, 2005. 9

[HR02] Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 342–356. Springer, 2002. 4, 6, 7

[Ins] Texas Instruments. Real-time system performance monitoring. http://www.ni.com/white-paper/52225/en/. Last access 2 of October 2016. 13

[IWWP13] P. Iyenghar, J. Wuebbelmann, C. Westerkamp, and E. Pulvermueller. Model-based test case generation by reusing models from runtime monitoring of deeply embedded systems. *Embedded Systems Letters, IEEE*, 5(3):38–41, Sept 2013. vii, xvii, 5, 6, 9, 10, 11, 16, 17

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. Available from: `http://doi.acm.org/10.1145/1629575.1629596`. 7

[Lib] RXTX Library. Rxtx. `http://rxtx.qbang.org/`. Last access 2 of October 2016. 69

[Loga] Express Logic. Threadx performance analysis. `http://rtos.com/articles/18834`. Last access 2 of October 2016. vii, 13

[Logb] Express Logic. Tracex. `http://rtos.com/products/tracex`. Last access 2 of October 2016. 14

[MP11] Vincent Lefftz Mathieu Patte. System impact of distributed multi core systems (sidms) final report. `http://microelectronics.esa.int/ngmp/SIDMS_Final_Report.pdf`, November 2011. Last access 2 of October 2016. 65

[NPP15] Geoffrey Nelissen, David Pereira, and Luis Miguel Pinho. A novel run-time monitoring architecture for safe and efficient inline monitoring. In *Reliable Software Technologies–Ada-Europe 2015*, pages 66–82. Springer, 2015. 7, 8, 11, 17

[On-15] On-Line Applications Research Corporation. *RTEMS C User's Guide*, 4.10.99.0 edition, July 2015. 50, 67

[Per] Percepio. Tracealizer. `http://percepio.com/tz/`. Last access 2 of October 2016. vii, 12

[PN81] B. Plattner and J. Nievergelt. Special feature: Monitoring program execution: A survey. *Computer*, 14(11):76–93, Nov 1981. vii, 6, 17

[Pol] Polarsys. Trace compass solution. `https://www.polarsys.org/solutions/tracecompass`. Last access 2 of October 2016. vii, 11

[Pro] The Linux Documentation Project. What are uarts? how do they affect performance? `http://tldp.org/HOWTO/Serial-HOWTO-18.html`. Last access 2 of October 2016. 32

[SEGa] SEGGER. embos. `https://www.segger.com/embos.html`. Last access 2 of October 2016. 14

[SEGb] SEGGER. embossegger microcontroller. `https://community.arm.com/docs/DOC-3651`. Last access 2 of October 2016. 14

[SEGc] SEGGER. Segger systemview. `https://www.segger.com/systemview.html`. Last access 2 of October 2016. vii, 15

[Sho04] Mohammed El Shobaki. On-chip monitoring for non-intrusive hardware/software observability. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-120/2004-1-SE, September 2004. Available from: `http://www.es.mdh.se/publications/631-`. 4

[WH07] C. Watterson and D. Heffernan. Runtime verification and monitoring of embedded systems. *Software, IET*, 1(5):172–179, October 2007. 6, 7, 8, 11

Appendix A

Monitoring Tool User Manual

A.1 Monitoring Tool Installation

This section describes how to install the RTEMS Monitoring tool. It is a pre-requisite for the installation that the user detains knowledge in Linux commands and the compilation process of RTEMS. It is also assumed that the user already has RTEMS Improvement completely installed and working.

A.1.1 Installing the host tools

This section describes how to obtain and install the tools required for the Monitoring tool host in the Host platform. Both tools (MySQL [Corc] and JRE [Corb]) can be found in their respective websites.

A.1.2 Monitoring Tool

This section describes how to obtain and unarchive the Monitoring tool. For future reference, it is assumed that the Monitoring tool files are in the `$RIMP_ROOT/archive` directory. The variable `$RIMP_ROOT` should be set as defined in listing A.1.

```
export RIMP_ROOT=''/home/rtems/''
```

Listing A.1: Setting the `$RIMP_ROOT` variable.

A.1.2.1 Unarchiving the Monitoring source

The files obtained in the previous step are compressed and need to be unarchived in order to be used. It is assumed that the Monitoring tool will be unzipped to the `$RIMP_ROOT/build` directory. To do this, type the commands shown in listing A.2.

```
cd $RIMP_ROOT/build
unzip ../archive/MON_TOOL.zip
```

Listing A.2: Commands to unarchive the monitoring tool.

Doing this will create the directory `MON_TOOL`. Inside this directory you will find the Stub directory (which contains the stub files) and the Host directory (which contains the host files). The file `RTEMSMonitoringHost.jar` (the host program) can be found in the Host/dist directory, which must be copied to the EGSE SCOC3 computer.

A.1.3 Monitoring Host

This section explains the process to configure and execute the host. It is assumed that the `RTEMSMonitoringHost.jar` file is already extracted in the `$RIMP_ROOT/build/MON_TOOL` directory.

A.1.4 Executing the Monitoring Host

The host will retrieve the information provided by the application (compiled with the stub). Run the command in listing A.3 to execute the Monitoring host (the iSAFT RTE service must be already running).

```
java -jar RTEMSMonitoringHost.jar <Configuration>
```

Listing A.3: Command to execute the monitoring host.

When launching the RTEMS Monitoring Host, there are at least two parameters that need to be specified in the case that a MySQL database is used: a database username and the corresponding password to write the retrieved data. It is also possible to change the name of the CSV file where the data is written. After the host has been started, the stub can be executed and the communication established.

A.1.5 Host Configuration

Configuration of the Monitoring host is done by supplying the arguments shown in table A.1.

Parameter Name	**Description**
-db <url> <username> <password>	Indicates that the results shall be saved on a MySQL database
-csv <csv_directory>	Indication that the results shall be saved on CSV files

Table A.1: Monitoring Host Configuration.

The fields between < > must be replaced with the corresponding information. While the stub is transmitting, the Monitoring host will save the results either in the MySQL database or CSV files depending on which parameter was used. If the data is being saved into a database, the user and password to access it are needed, as well as the database URL. Each execution of the user application will result on a new database, with one table per event type. If CSV files are being used then each execution of the user application will result on a new directory containing the CSV files, one file per event type. In both formats each execution is labeled with a timestamp from the machine running the Monitoring host.

A.1.6 Monitoring Stub

This section explains the process to configure and execute the stub. It is assumed that the Stub folder is already extracted in the `$RIMP_ROOT/build/MON_TOOL` directory.

A.1.6.1 Stub API

The Monitoring tool provides an API with three functions that can be used by the target application. These provide the possibility to force the buffer transmission, enable or disable the Monitoring tool in runtime and log a custom user message. They can be easily integrated in a RTEMS application by adding the header `monitoring_api.h` (refer to listing A.4) to the target application and including the files `monitoring_api.h` and `monitoring_api.c` in the application Makefile. These two files contain a stub implementation of each API function in order for the programmer to compile the application successfully. When the result its compiled with the Monitoring tool stub files, the real functions will be called.

```
#include ``monitoring_api.h''
```

Listing A.4: Include file that contains the prototypes of the original functions.

monitoring_flush

This function provide means to the application to force the transmission of the buffer (totally or partially). When calling this function a parameter is requested: if zero the function will try to transmit the entire buffer, otherwise it will try to transmit the total number of events requested. Note that when the buffer contains less logs than the parameter supplied, it is the same as calling the function with the value zero. The input parameters are shown in table A.2. Listing A.4 shows the required headers that contains the prototypes of the functions to include in the application, and listing A.5 presents an usage example.

Member	Type	Description
total	uint32_t	The total logs to transmit. If zero, sends all logs in the buffer.

Table A.2: Monitoring Stub API: monitoring_flush parameters.

```
monitoring_flush (7);
```

Listing A.5: Example showing how to force the transmission of seven logs.

Return Value:

- Integer - total number of events transmitted.

timeline_user_event

This function provide means to the application to log a specific user message. When calling this function a parameter with the user message is requested. The input parameters are shown in table A.3. Listing A.4 shows the required headers that contains the prototypes of the original functions to include in the application, and listing A.6 presents an usage example.

Member	Type	Description
message	uint32_t	The message to log.

Table A.3: Monitoring Stub API: timeline_user_event parameters.

```
timeline_write_user_event (23);
```

Listing A.6: Example showing how to log a specific message.

Return Value:

- 0 - successfully logged;
- 1 - error logging the message.

monitoring_enable

This function provides means to the application to enable and disable the Monitoring tool by the target application in execution time. When setting this variable a value is requested: if zero

the tool will stop its monitoring capabilities, if one, it enables the monitoring capabilities. Note that this function either enables all the previous monitoring capabilities or disables everything, it cannot disable one custom function call (see section A.1.7 to enable or disable custom calls). The input parameters are shown in table A.4. Listing A.4 shows the required headers that contains the variable definition to include in the application, and listing A.7 presents an usage example.

Member	Type	Description
monitoring_enable	int	0 - Disables the log functionality
		1 - Enables the log functionality

Table A.4: Monitoring Stub API: monitoring_enable parameters.

```
//<application code>

monitoring_enable (0);    /* Disable the tool */

//<application code >

monitoring_enable (1);    /* Enable the tool */

//<application code >
```

Listing A.7: Example showing how to enable and disable the tool monitoring capabilities.

A.1.7 Stub Configuration

Configuration of the Monitoring stub is done by modifying the parameters in the timeline_user_configuration.h header file. This file can be found in the Stub folder. Note that any changes to this file implies the Stub recompilation.

A.1.7.1 User Configuration

The timeline_user_configuration.h file allows the developer to select which events are monitored and to set a few configurations to the Monitoring tool. There are several features that can be modified in this file. The following sub sections contains the user configurable parameters. Note that changing any of the items described implies a new recompilation of the stub so the changes can take effect. To recompile the stub perform as explained in section A.4.

A.1.7.2 Tool Configuration

Tables A.5 and A.6 present the generic tool configuration parameters available to the user.

A.1.7.3 RTEMS API: Task Manager

Tables A.7 and A.8 present the available Task Manager API functions to monitor.

A.1.7.4 RTEMS API: Interrupt Manager

Table A.9 presents the available Interrupt Manager API functions to monitor.

Parameter Name	**Description**	**Parameter Range**
TIMELINE_VERBOSE_MODE	Displays debug messages in the target platform STDOUT	0 - Disable
		1 - Enable
TIMELINE_BUFFER_OVERRIDE	Override the oldest data when the buffer is full. If not, new events will be lost	0 - Disable
		1 - Enable
TIMELINE_TASK_DEFAULT_PRIORITY	The RTEMS Monitoring tool task priority	1.. 255
TIMELINE_TASK_DEFAULT_PERIOD	The RTEMS Monitoring tool task period (given in system clock ticks)	Integer range
TIMELINE_TASK_DEFAULT_NUMBER_MESSAGES_SEND	The RTEMS Monitoring tool number of messages to send per period (only used in on-line mode)	Integer range
TIMELINE_MONITOR_INTERRUPTS	Interrupts are monitored	0 - Disable
		1 - Enable
TIMELINE_TASK_SCHEDULABILITY_DEFAULT_HIGH_PRIORITY	A task schedulability is monitored if its initial priority is between these two thresholds	1.. 255
TIMELINE_TASK_SCHEDULABILITY_DEFAULT_LOW_PRIORITY		1.. 255
TIMELINE_TASK_STACK_DEFAULT_HIGH_PRIORITY		1.. 255
TIMELINE_TASK_STACK_DEFAULT_LOW_PRIORITY		1.. 255
TIMELINE_DEFAULT_API_HIGH_PRIORITY	A RTEMS API call is monitored if the task that performed the call original priority is between these two thresholds	1.. 255
TIMELINE_DEFAULT_API_LOW_PRIORITY		1.. 255
TIMELINE_DEFAULT_API_INTERRUPT_TRACE	Monitor RTEMS API calls that were performed inside an ISR	0 - Disable
		1 - Enable

Table A.5: Monitoring Stub Configuration Parameters (1/2).

Parameter Name	Description	Parameter Range
TIMELINE_ON_LINE_MODE	The Monitoring on-line mode is used. Note that the off-line mode must not be active	0 - Disable
		1 - Enable
TIMELINE_OFF_LINE_MODE	The Monitoring off-line mode is used. Note that the on-line mode must not be active	0 - Disable
		1 - Enable
TIMELINE_SPW_HOST_ADDRESS	The Monitoring SpW host address	0.. 255
TIMELINE_SPW_MAJOR_NUMBER	The major number to register the driver (make sure it is available)	0..NUMBER_MAXIMUM_OF_DRIVERS
TIMELINE_SPW_MINOR_NUMBER	The minor number to use to send messages	0.. 5

Table A.6: Monitoring Stub Configuration Parameters (2/2).

A.1.7.5 RTEMS API: Clock Manager

Table A.10 presents the available Clock Manager API functions to monitor.

A.1.7.6 RTEMS API: Timer Manager

Table A.11 presents the available Timer Manager API functions to monitor.

A.1.7.7 RTEMS API: Semaphore Manager

Table A.12 presents the available Semaphore Manager API functions to monitor.

A.1.7.8 RTEMS API: Message Queue Manager

Table A.13 presents the available Message Queue Manager API functions to monitor.

A.1.7.9 RTEMS API: Event Manager

Table A.14 presents the available Event Manager API functions to monitor.

A.1.7.10 RTEMS API: I/O Manager

Table A.15 presents the available I/O Manager API functions to monitor.

A.1.7.11 RTEMS API: Error Manager

Table A.16 presents the available Error Manager API functions to monitor.

A.1.7.12 RTEMS API: Rate Monotonic Manager

Table A.17 presents the available Rate Monotonic Manager API functions to monitor.

Parameter Name	**Description**	**Parameter Range**
TIMELINE_DEFAULT_API_TASK_CREATE_TRACE	Monitor rtems_task_create function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TASK_IDENT_TRACE	Monitor rtems_task_ident function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TASK_START_TRACE	Monitor rtems_task_start function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TASK_RESTART_TRACE	Monitor rtems_task_restart function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TASK_DELETE_TRACE	Monitor rtems_task_delete function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TASK_SUSPEND_TRACE	Monitor rtems_task_suspend function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TASK_RESUME_TRACE	Monitor rtems_task_resume function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TASK_IS-_SUSPENDED_TRACE	Monitor rtems_task_is-_suspended function	0 - Disable
		1 - Enable

Table A.7: Monitoring Stub Configuration: Task Manager (1/2).

Parameter Name	Description	Parameter Range
TIMELINE_DEFAULT_API_TASK_SET-_PRIORITY_TRACE	Monitor rtems_task-_set_priority function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TASK_MODE_TRACE	Monitor rtems_task_mode function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TASK_GET_NOTE_TRACE	Monitor rtems_task_get_note function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TASK_SET_NOTE_TRACE	Monitor rtems_task_set_note function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TASK_WAKE-_AFTER_TRACE	Monitor rtems_task-_wake_after function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TASK_WAKE-_WHEN_TRACE	Monitor rtems_task-_wake_when function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TASK_VARIABLE-_ADD_TRACE	Monitor rtems_task_variable-_add function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TASK_VARIABLE-_GET_TRACE	Monitor rtems_task_variable-_get function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TASK_VARIABLE-_DELETE_TRACE	Monitor rtems_task_variable-_delete function	0 - Disable
		1 - Enable

Table A.8: Monitoring Stub Configuration: Task Manager (2/2).

Parameter Name	Description	Parameter Range
TIMELINE_DEFAULT_API_INTERRUPT-_CATCH_TRACE	Monitor rtems_interrupt-_catch function	0 - Disable
		1 - Enable

Table A.9: Monitoring Stub Configuration: Interrupt Manager.

Parameter Name	Description	Parameter Range
TIMELINE_DEFAULT_API_CLOCK_SET_TRACE	Monitor rtems_clock_set function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_CLOCK_GET_TRACE	Monitor rtems_clock_get function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_CLOCK_SET-_NANOSECONDS_EXTENSION_TRACE	Monitor rtems_clock_set-_nanoseconds-_extension function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_CLOCK_GET-_UPTIME_TRACE	Monitor rtems_clock-_get_uptime function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_CLOCK_TICK_TRACE	Monitor rtems_clock_tick function	0 - Disable
		1 - Enable

Table A.10: Monitoring Stub Configuration: Clock Manager.

Parameter Name	Description	Parameter Range
TIMELINE_DEFAULT_API_TIMER_CREATE_TRACE	Monitor rtems_timer_create function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TIMER_IDENT_TRACE	Monitor rtems_timer_ident function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TIMER_CANCEL_TRACE	Monitor rtems_timer_cancel function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_TIMER_DELETE_TRACE	Monitor rtems_timer_delete function	0 - Disable
		1 - Enable

Table A.11: Monitoring Stub Configuration: Timer Manager.

Parameter Name	Description	Parameter Range
TIMELINE_DEFAULT_API_SEMAPHORE-_CREATE_TRACE	Monitor rtems_semaphore-_create function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_SEMAPHORE-_IDENT_TRACE	Monitor rtems_semaphore-_ident function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_SEMAPHORE-_DELETE_TRACE	Monitor rtems_semaphore-_delete function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_SEMAPHORE-_OBTAIN_TRACE	Monitor rtems_semaphore-_obtain function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_SEMAPHORE-_RELEASE_TRACE	Monitor rtems_semaphore-_release function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_SEMAPHORE-_FLUSH_TRACE	Monitor rtems_semaphore-_flush function	0 - Disable
		1 - Enable

Table A.12: Monitoring Stub Configuration: Semaphore Manager.

A.1.7.13 RTEMS API: User Extension Manager

Table A.18 presents the available User Extension Manager API functions to monitor.

A.2 Compiling an Application

The process to compile an application to be monitored is the same as for any RTEMS application. The only care to be taken is that the object code should not be deleted after the application is compiled (as the monitoring tool stub will link with those object files) and the warnings shown in section A.5.

A.3 Linking the RTEMS Monitoring Stub with the Application

After configuring the RTEMS Monitoring tool, it must be linked with the application object files to produce the final executable. This can be done using the Makefile in the RTEMS Monitoring tool stub. Table A.19 shows the parameters needed to link the compiled application with the RTEMS Monitoring tool.
With the Monitoring stub Makefile configured, the command `make` can be executed in the RTEMS Monitoring tool stub folder. The executable produced by this step can be found in the `o-optimize` folder inside the Monitoring tool stub folder. It can now be loaded into the target platform and

Parameter Name	Description	Parameter Range
TIMELINE_DEFAULT_API_MESSAGE_QUEUE-_CREATE_TRACE	Monitor rtems_message-_queue_create function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_MESSAGE_QUEUE-_IDENT_TRACE	Monitor rtems_message-_queue_ident function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_MESSAGE_QUEUE-_DELETE_TRACE	Monitor rtems_message-_queue_delete function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_MESSAGE_QUEUE-_SEND_TRACE	Monitor rtems_message-_queue_send function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_MESSAGE_QUEUE-_URGENT_TRACE	Monitor rtems_message-_queue_urgent function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_MESSAGE_QUEUE-_BROADCAST_TRACE	Monitor rtems_message-_queue_broadcast function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_MESSAGE_QUEUE-_RECEIVE_TRACE	Monitor rtems_message-_queue_receive function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_MESSAGE_QUEUE-_GET_NUMBER_PENDING_TRACE	Monitor rtems_message-_queue_get-_number_pending function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_MESSAGE_QUEUE-_FLUSH_TRACE	Monitor rtems_message-_queue_flush function	0 - Disable
		1 - Enable

Table A.13: Monitoring Stub Configuration: Queue Manager.

Parameter Name	Description	Parameter Range
TIMELINE_DEFAULT_API_EVENT_SEND_TRACE	Monitor rtems_event_send function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_EVENT_RECEIVE_TRACE	Monitor rtems_event_receive function	0 - Disable
		1 - Enable

Table A.14: Monitoring Stub Configuration: Event Manager.

Parameter Name	Description	Parameter Range
TIMELINE_DEFAULT_API_IO_REGISTER-_DRIVER_TRACE	Monitor rtems_io_register-_driver function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_IO_INITIALIZE_TRACE	Monitor rtems_io_initialize function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_IO_OPEN_TRACE	Monitor rtems_io_open function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_IO_CLOSE_TRACE	Monitor rtems_io_close function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_IO_READ_TRACE	Monitor rtems_io_read function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_IO_WRITE_TRACE	Monitor rtems_io_write function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_IO_CONTROL_TRACE	Monitor rtems_io_control function	0 - Disable
		1 - Enable

Table A.15: Monitoring Stub Configuration: I/O Manager.

Parameter Name	Description	Parameter Range
TIMELINE_DEFAULT_API_FATAL_ERROR-_OCCURRED_TRACE	Monitor rtems_fatal_error-_occurred function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_ERROR_GET_LATEST-_NON_FATAL_BY_OFFSET_TRACE	Monitor rtems_error_get-_latest_non_fatal-_by_offset function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_ERROR_GET_LATEST-_FATAL_BY_OFFSET_TRACE	Monitor rtems_error_get-_latest_fatal-_by_offset function	0 - Disable
		1 - Enable

Table A.16: Monitoring Stub Configuration: Error Manager.

executed. Note that the Monitoring host should be running before the monitored application is started, so it can capture the execution events.

A.4 Stub Compilation

In order to compile the target application with the Monitoring stub, the user needs to copy the Stub folder to the target application folder. Assuming that the target application folder is `$RIMP_ROOT/progr1` the command in listing A.8 should be run.

```
cp -r $RIMP_ROOT/build/MON_TOOL/Stub $RIMP_ROOT/progr1/
```

Listing A.8: Copying the Monitoring Stub inside the monitored application directory.

The next step is to compile the target program, usually the program has a makefile and the command `make` is enough to compile the target application. Before continuing, ensure that the folder `o-optimize` contains the application executable. Note that this folder is automatically created after a successful compilation of the target application. After the program is successfully compiled it should be linked with the monitoring stub wrappers using the makefile that is supplied inside the Stub folder. This makefile is similar to one used when compiling RTEMS, with the modifications presented in section A.3. If the target application is already compiled and the stub Makefile configured, the commands in listing A.9 are required to compile the application with the stub.

```
cd $RIMP_ROOT/Programs/progr1/Stub
make
```

Listing A.9: Compiling the monitoring stub, and linking with the monitored application object code.

After the compilation process is completed, the application `$RIMP_ROOT/Programs/progr1/Stub/mtApp.exe` (assuming the name chosen for the executable is `mtApp`) can be executed in the target host machine. To deploy and execute the application in the GR712RC-BOARD execute the command in listing A.10, and when inside the GRMON shell execute the commands in listing A.11.

Parameter Name	Description	Parameter Range
TIMELINE_DEFAULT_API_RATE_MONOTONIC-_CREATE_TRACE	Monitor rtems_rate-_monotonic_create function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_RATE_MONOTONIC-_IDENT_TRACE	Monitor rtems_rate-_monotonic_ident function	0 - Disable
		1 - Enable
IMELINE_DEFAULT_API_RATE_MONOTONIC-_CANCEL_TRACE	Monitor rtems_rate-_monotonic_cancel function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_RATE_MONOTONIC-_DELETE_TRACE	Monitor rtems_rate-_monotonic_delete function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_RATE_MONOTONIC-_PERIOD_TRACE	Monitor rtems_rate-_monotonic_period function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_RATE_MONOTONIC-_GET_STATUS_TRACE	Monitor rtems_rate-_monotonic_get-_status function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_RATE_MONOTONIC-_DEADLINE_TRACE	Monitor rtems_rate-_monotonic_deadline function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_RATE_MONOTONIC-_GET_DEADLINE_STATE_TRACE	Monitor rtems_rate-_monotonic_get-_deadline_state function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_RATE_MONOTONIC-_EXECUTION_TIME_TRACE	Monitor rtems_rate-_monotonic-_execution_time function	0 - Disable
		1 - Enable

Table A.17: Monitoring Stub Configuration: Rate Monotonic Manager.

Parameter Name	Description	Parameter Range
TIMELINE_DEFAULT_API_EXTENSION-_CREATE_TRACE	Monitor rtems_extension-_create function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_EXTENSION-_IDENT_TRACE	Monitor rtems_extension-_ident function	0 - Disable
		1 - Enable
TIMELINE_DEFAULT_API_EXTENSION-_DELETE_TRACE	Monitor rtems_extension-_delete function	0 - Disable
		1 - Enable

Table A.18: Monitoring Stub Configuration: User Extension Manager.

Parameter Name	Description
SAMPLE	The name for the final executable containing the application, RTEMS and RTEMS Monitoring stub, as seen in figure 3.1.
APPLICATION_OBJECT_FILES_PATH	The application object files directory. Corresponds to the directory where the application files are located at. For example: `$RIMP_ROOT/Programs/progr1/o-optimize`.
MANAGERS	The managers for the RTEMS Monitoring tool (should be set to all).
LIBS	The libraries used. Should be equal to the libraries used by the application.
RTEMS_MAKEFILE_PATH	The RTEMS makefile path directory. Corresponds to the directory where RTEMS was installed.
TIMELINE_COMPILER_ARGS	The functions to be wrapped (monitored).

Table A.19: Monitoring Stub compilation parameters.

```
grmon -nb -nswb -abaud 115200 -ftdi -gr712 -u
```

Listing A.10: Starting GRMON.

```
load o-optimize/mtApp.exe
run
```

Listing A.11: Loading and executing the monitored application linked with the monitoring stub in GRMON.

The application should now be running, and the output appearing on the screen. To exit the GRMON shell type `quit`.

A.5 Warnings

All the logs collected by the target application are sent to the host through SpaceWire. The monitoring tool takes care of the SpW driver registration and initialization meaning that the programmer should not call the `grspw_register` function in the target application because it will register the driver for the second time (and erase all the configurations performed by the Monitoring tool). Taking into account this restriction, it is offered to the programmer the possibility to configure the major and minor numbers of the SpW driver and core to be used by the Monitoring tool. Note that the major number attribution is always dependent of the function `rtems_io_register_driver`.

After the Monitoring tool is initialized, the SpW driver is usable and registered. The target application can then use the all the SpW cores except for the one set in the configuration file. Also note that since the monitoring tool only uses 3-bytes to store an event microsecond, the clock tick cannot be greater than 224 microseconds ($\sim$16 seconds).

Appendix B

General Purpose Test Results

Listings B.1, B.2, B.3, B.4, B.5 and B.6 show a snippet of the trace data from the test `val_10_01010` for the different event types (each listing only shows at most 5 lines of the trace, as the complete trace would be too long), and listing B.7 shows a snippet of the corresponding test results (only the first line of each event line is shown, as the full test report has 348659 lines). Since the test reports are so long, only the test report for the test `val_10_01010` is shown here.

```
3,0,198928,2,1430860064,167772161,1077558752,0,0,NULL,NULL,NULL,1077604780,
    NULL,NULL,NULL,NULL,NULL,NULL
3,0,199014,2,1430860064,167772161,1077558752,1,0,NULL,NULL,NULL,NULL,NULL,
    NULL,NULL,NULL,NULL,0
3,0,199048,2,1430860064,167772161,1077558752,0,4,NULL,NULL,NULL,1077604732,
    NULL,NULL,NULL,NULL,NULL,NULL
3,0,199085,2,1430860064,167772161,1077558752,1,4,NULL,NULL,NULL,NULL,NULL,
    NULL,NULL,NULL,NULL,0
3,0,199118,1,1430860064,167772161,1077558752,0,0,NULL,NULL,NULL
    ,1073835604,0,1077604728,NULL,NULL,NULL,NULL
```

Listing B.1: Snippet of the trace data with the first 5 RTEMS API events for the general purpose test `val_10_01010`.

```
6,0,0,1077537496,349480,1,10000,50
```

Listing B.2: Trace data with the configuration events for the general purpose test `val_10_01010`.

```
2,0,201155,0,1936028781,167772163,1077559600,97,NULL
2,0,202573,1,1936028781,167772163,1077559600,97,NULL
2,0,202928,10,1936028781,167772163,1077559600,97,NULL
2,0,203025,9,1936028781,167772163,1077559600,97,NULL
2,0,203676,2,1936028781,167772163,1077559600,97,NULL
```

Listing B.3: Snippet of the trace data with the first 5 task events for the general purpose test `val_10_01010`.

```
1,0,198813,0,24
1,0,198876,1,24
1,0,200174,0,24
1,0,200238,1,24
1,0,210150,0,24
```

Listing B.4: Snippet of the trace data with the first 5 interrupt events for the general purpose test `val_10_01010`.

```
7,0,199770,56
7,1,1611590,1
7,1,1612985,2
```

```
7,1,1614361,3
7,1,1620560,4
```

Listing B.5: Snippet of the trace data with the first 5 user events for the general purpose test `val_10_01010`.

```
4,0,203633,1430860064,167772161,1077558752,4112,1077600920,1077605032
4,0,203842,1936028781,167772163,1077559600,1180,1077613704,1077621912
4,0,613555,1430860064,167772161,1077558752,4112,1077600920,1077605032
4,0,613651,1414090053,167772186,1077569352,924,1077800968,1077805080
4,0,615276,1918989419,167772185,1077568928,1180,1077792456,1077800664
```

Listing B.6: Snippet of the trace data with the first 5 stack events for the general purpose test `val_10_01010`.

```
**************************************************
*                                                *
*       Test report for test case val_10_01010   *
*                                                *
**************************************************
Testing Log Line: 2, 0, 201155, 0, 1936028781, 167772163, 1077559600, 97,
    NULL
Test step 10
Step message: Testing if event type code is correct
Expected output: 2
Actual output: 2
Step result: Pass

Test step 20
Step message: Testing if the clock tick is within the correct byte size
Expected output: at most 4 bytes
Actual output: 1 bytes
Step result: Pass

Test step 30
Step message: Testing if the microseconds is within the correct byte size
Expected output: at most 3 bytes
Actual output: 3 bytes
Step result: Pass

Test step 40
Step message: Testing if the number of fields is correct
Expected output: 9
Actual output: 9
Step result: Pass

Test step 50
Step message: Testing if task event type is correct
Expected output: between 0 and 11
```

```
Actual output: 0
Step result: Pass

Test step 60
Step message: Testing if the task name is within the correct byte size
Expected output: at most 4 bytes
Actual output: 4 bytes
Step result: Pass

Test step 70
Step message: Testing if the task id is within the correct byte size
Expected output: at most 4 bytes
Actual output: 4 bytes
Step result: Pass

Test step 80
Step message: Testing if the TCB address is within the correct byte size
Expected output: at most 4 bytes
Actual output: 4 bytes
Step result: Pass

Test step 90
Step message: Testing if the task priority fits the test priority window
    filter
Expected output: between 70 and 110
Actual output: 97
Step result: Pass

Test step 100
Step message: Testing that the new priority field is not defined
Expected output: null
Actual output: null
Step result: Pass


(...)


Testing Log Line: 3, 0, 198928, 2, 1430860064, 167772161, 1077558752, 0, 0,
    NULL, NULL, NULL, 1077604780, NULL, NULL, NULL, NULL, NULL, NULL
Test step 104110
Step message: Testing if event type code is correct
Expected output: 3
Actual output: 3
Step result: Pass

Test step 104120
```

```
Step message: Testing if the clock tick is within the correct byte size
Expected output: at most 4 bytes
Actual output: 1 bytes
Step result: Pass

Test step 104130
Step message: Testing if the microseconds is within the correct byte size
Expected output: at most 3 bytes
Actual output: 3 bytes
Step result: Pass

Test step 104140
Step message: Testing if the number of fields is correct
Expected output: 19
Actual output: 19
Step result: Pass

Test step 104150
Step message: Testing if the API manager is valid
Expected output: between 0 and 10
Actual output: 2
Step result: Pass

Test step 104160
Step message: Testing if the API manager function is valid
Expected output: between 0 and 4
Actual output: 0
Step result: Pass

Test step 104170
Step message: Testing that the object name field is not defined
Expected output: null
Actual output: null
Step result: Pass

Test step 104180
Step message: Testing that the object id field is not defined
Expected output: null
Actual output: null
Step result: Pass

Test step 104190
Step message: Testing that the object address field is not defined
Expected output: null
Actual output: null
Step result: Pass
```

```
Test step 104200
Step message: Testing that the returning task name is within the correct
    byte size
Expected output: at most 4 bytes
Actual output: 4 bytes
Step result: Pass

Test step 104210
Step message: Testing that the returning task id is within the correct byte
    size
Expected output: at most 4 bytes
Actual output: 4 bytes
Step result: Pass

Test step 104220
Step message: Testing that the returning TCB address is within the correct
    byte size
Expected output: at most 4 bytes
Actual output: 4 bytes
Step result: Pass

Test step 104230
Step message: Testing if the call type is valid
Expected output: between 0 and 1
Actual output: 0
Step result: Pass

Test step 104240
Step message: Testing if the argument 12 is within the correct byte size
Expected output: at most 4 bytes
Actual output: 4 bytes
Step result: Pass

Test step 104250
Step message: Testing that the unused argument field 13 is not defined
Expected output: null
Actual output: null
Step result: Pass

Test step 104260
Step message: Testing that the unused argument field 14 is not defined
Expected output: null
Actual output: null
Step result: Pass

Test step 104270
Step message: Testing that the unused argument field 15 is not defined
```

```
Expected output: null
Actual output: null
Step result: Pass

Test step 104280
Step message: Testing that the unused argument field 16 is not defined
Expected output: null
Actual output: null
Step result: Pass

Test step 104290
Step message: Testing that the unused argument field 17 is not defined
Expected output: null
Actual output: null
Step result: Pass

Test step 104300
Step message: Testing that the return value is not defined
Expected output: null
Actual output: null
Step result: Pass


(...)


Testing Log Line: 7, 0, 199770, 56
Test step 416510
Step message: Testing if event type code is correct
Expected output: 7
Actual output: 7
Step result: Pass

Test step 416520
Step message: Testing if the clock tick is within the correct byte size
Expected output: at most 4 bytes
Actual output: 1 bytes
Step result: Pass

Test step 416530
Step message: Testing if the microseconds is within the correct byte size
Expected output: at most 3 bytes
Actual output: 3 bytes
Step result: Pass

Test step 416540
Step message: Testing if the number of fields is correct
```

```
Expected output: 4
Actual output: 4
Step result: Pass

Test step 416550
Step message: Testing that the user message is within the correct byte size
Expected output: at most 4 bytes
Actual output: 1 bytes
Step result: Pass


(...)


Testing Log Line: 4, 0, 203633, 1430860064, 167772161, 1077558752, 4112,
    1077600920, 1077605032
Test step 425560
Step message: Testing if event type code is correct
Expected output: 4
Actual output: 4
Step result: Pass

Test step 425570
Step message: Testing if the clock tick is within the correct byte size
Expected output: at most 4 bytes
Actual output: 1 bytes
Step result: Pass

Test step 425580
Step message: Testing if the microseconds is within the correct byte size
Expected output: at most 3 bytes
Actual output: 3 bytes
Step result: Pass

Test step 425590
Step message: Testing if the number of fields is correct
Expected output: 9
Actual output: 9
Step result: Pass

Test step 425600
Step message: Testing that the task name is within the correct byte size
Expected output: at most 4 bytes
Actual output: 4 bytes
Step result: Pass

Test step 425610
```

```
Step message: Testing that the task id is within the correct byte size
Expected output: at most 4 bytes
Actual output: 4 bytes
Step result: Pass

Test step 425620
Step message: Testing that the TCB address is within the correct byte size
Expected output: at most 4 bytes
Actual output: 4 bytes
Step result: Pass

Test step 425630
Step message: Testing that the stack usage is within the correct byte size
Expected output: at most 4 bytes
Actual output: 2 bytes
Step result: Pass

Test step 425640
Step message: Testing that the stack low address is within the correct byte
    size
Expected output: at most 4 bytes
Actual output: 4 bytes
Step result: Pass

Test step 425650
Step message: Testing that the stack high address is within the correct byte
     size
Expected output: at most 4 bytes
Actual output: 4 bytes
Step result: Pass


(...)


Testing Log Line: 1, 0, 198813, 0, 24
Test step 447360
Step message: Testing if event type code is correct
Expected output: 1
Actual output: 1
Step result: Pass

Test step 447370
Step message: Testing if the clock tick is within the correct byte size
Expected output: at most 4 bytes
Actual output: 1 bytes
Step result: Pass
```

```
Test step 447380
Step message: Testing if the microseconds is within the correct byte size
Expected output: at most 3 bytes
Actual output: 3 bytes
Step result: Pass

Test step 447390
Step message: Testing if the number of fields is correct
Expected output: 5
Actual output: 5
Step result: Pass

Test step 447400
Step message: Testing interrupt event type
Expected output: between 0 and 1
Actual output: 0
Step result: Pass

Test step 447410
Step message: Testing if the interrupt source is within the correct byte
    size
Expected output: at most 4 bytes
Actual output: 1 bytes
Step result: Pass


(...)


Testing Log Line: 6, 0, 0, 1077537496, 349480, 1, 10000, 50
Test step 572520
Step message: Testing if event type code is correct
Expected output: 6
Actual output: 6
Step result: Pass

Test step 572530
Step message: Testing if the clock tick is within the correct byte size
Expected output: at most 4 bytes
Actual output: 1 bytes
Step result: Pass

Test step 572540
Step message: Testing if the microseconds is within the correct byte size
Expected output: at most 3 bytes
Actual output: 1 bytes
```

```
Step result: Pass

Test step 572550
Step message: Testing if the number of fields is correct
Expected output: 8
Actual output: 8
Step result: Pass

Test step 572560
Step message: Testing that the address is within the correct byte size
Expected output: at most 4 bytes
Actual output: 4 bytes
Step result: Pass

Test step 572570
Step message: Testing that the size is within the correct byte size
Expected output: at most 4 bytes
Actual output: 3 bytes
Step result: Pass

Test step 572580
Step message: Testing that the number of drivers is within the correct byte
    size
Expected output: at most 4 bytes
Actual output: 1 bytes
Step result: Pass

Test step 572590
Step message: Testing that the microseconds per tick is within the correct
    byte size
Expected output: at most 4 bytes
Actual output: 2 bytes
Step result: Pass

Test step 572600
Step message: Testing that the ticks per timeslice is within the correct
    byte size
Expected output: at most 4 bytes
Actual output: 1 bytes
Step result: Pass

************************************************
*                                              *
*                  Result: Passed              *
*                                              *
************************************************
```

Listing B.7: Snippet of the test report for the general purpose test `val_10_01010`.

Appendix C

Performance Tests Results

Listing C.1 show the test report for the performance test `val_20_01010` and listing C.2 shows the test report for the performance test `val_20_02010`.

```
***********************************************
*                                             *
*    Test report for test case val_20_01010   *
*                                             *
***********************************************

Test scenario description:

A task acquires a semaphore via rtems_semaphore_obtain
and waits for an event with rtems_event_receive,
causing a context switch and a stack save.
This scenario is defined to take 10 miliseconds to complete.

***********************************************
*                                             *
*            Starting test scenario           *
*                                             *
***********************************************

Saving call event to rtems_semaphore_obtain
Average time (out of 1000 iterations) required to save a call to
    rtems_semaphore_obtain: 30 microseconds

Saving return event from rtems_semaphore_obtain
Average time (out of 1000 iterations) required to save a return from
    rtems_semaphore_obtain: 28 microseconds

Saving call event to rtems_event_receive
Average time (out of 1000 iterations) required to save a call to
    rtems_event_receive: 28 microseconds

Saving context switch event
Average time (out of 1000 iterations) required to save a context switch
    event: 20 microseconds

Saving stack event
Average time (out of 1000 iterations) required to save a stack event: 331
    microseconds
```

```
Total time spent saving events: 437 microseconds

***********************************************
*                                             *
*              Ending test scenario           *
*                                             *
***********************************************

Step 10
Step message: Testing if the event saving time has a maximum overhead of 5%
Expected output: at most 500 microseconds
Actual output: 437 microseconds
Result: Passed

***********************************************
*                                             *
*                 Result: Passed              *
*                                             *
***********************************************
```

Listing C.1: Test report for the performance test `val_20_01010`.

```
***********************************************
*                                             *
*    Test report for test case val_20_02010   *
*                                             *
***********************************************

Test scenario description:

A task acquires a semaphore via rtems_semaphore_obtain
and waits for an event with rtems_event_receive,
causing a context switch and a stack save.
This scenario is defined to take 10 miliseconds to complete.

***********************************************
*                                             *
*             Starting test scenario          *
*                                             *
***********************************************

Sending call event to rtems_semaphore_obtain
Average time (out of 1000 iterations) required to send a call to
    rtems_semaphore_obtain: 86 microseconds

Sending return event from rtems_semaphore_obtain
```

```
Average time (out of 1000 iterations) required to send a return from
    rtems_semaphore_obtain: 85 microseconds

Sending call event to rtems_event_receive
Average time (out of 1000 iterations) required to send a call to
    rtems_event_receive: 90 microseconds

Sending context switch event
Average time (out of 1000 iterations) required to send a context switch
    event: 77 microseconds

Sending stack event
Average time (out of 1000 iterations) required to send a stack event: 95
    microseconds

Total time spent sending events: 433 microseconds

**********************************************
*                                            *
*              Ending test scenario          *
*                                            *
**********************************************

Step 10
Step message: Testing if the event sending time has a maximum overhead of 5%
Expected output: at most 500 microseconds
Actual output: 433 microseconds
Result: Passed

**********************************************
*                                            *
*                 Result: Passed             *
*                                            *
**********************************************
```

Listing C.2: Test report for the performance test `val_20_02010`.